Технология параллельного программирования OpenMP

Бахтин Владимир Александрович
к.ф.-м.н., зав. сектором Института прикладной математики им М.В.Келдыша РАН
ассистент кафедры системного программирования факультета вычислительной математики и кибернетики Московского университета им. М.В. Ломоносова

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы

Современные направления развития параллельных вычислительных систем

- Тенденции развития современных процессоров
- Существующие подходы для создания параллельных программ
- Что такое OpenMP?

Тенденции развития современных процессоров

В течение нескольких десятилетий развитие ЭВМ сопровождалось удвоением их быстродействия каждые 1.5-2 года. Это обеспечивалось и повышением тактовой частоты и совершенствованием архитектуры (параллельное и конвейерное выполнение команд).

Узким местом стала оперативная память. Знаменитый закон Мура, так хорошо работающий для процессоров, совершенно не применим для памяти, где скорости доступа удваиваются в лучшем случае каждые 6 лет.

Совершенствовались системы кэш-памяти, увеличивался объем, усложнялись алгоритмы ее использования.

Для процессора Intel Itanium:

Latency to L1: 1-2 cycles

Latency to L2: 5 - 7 cycles

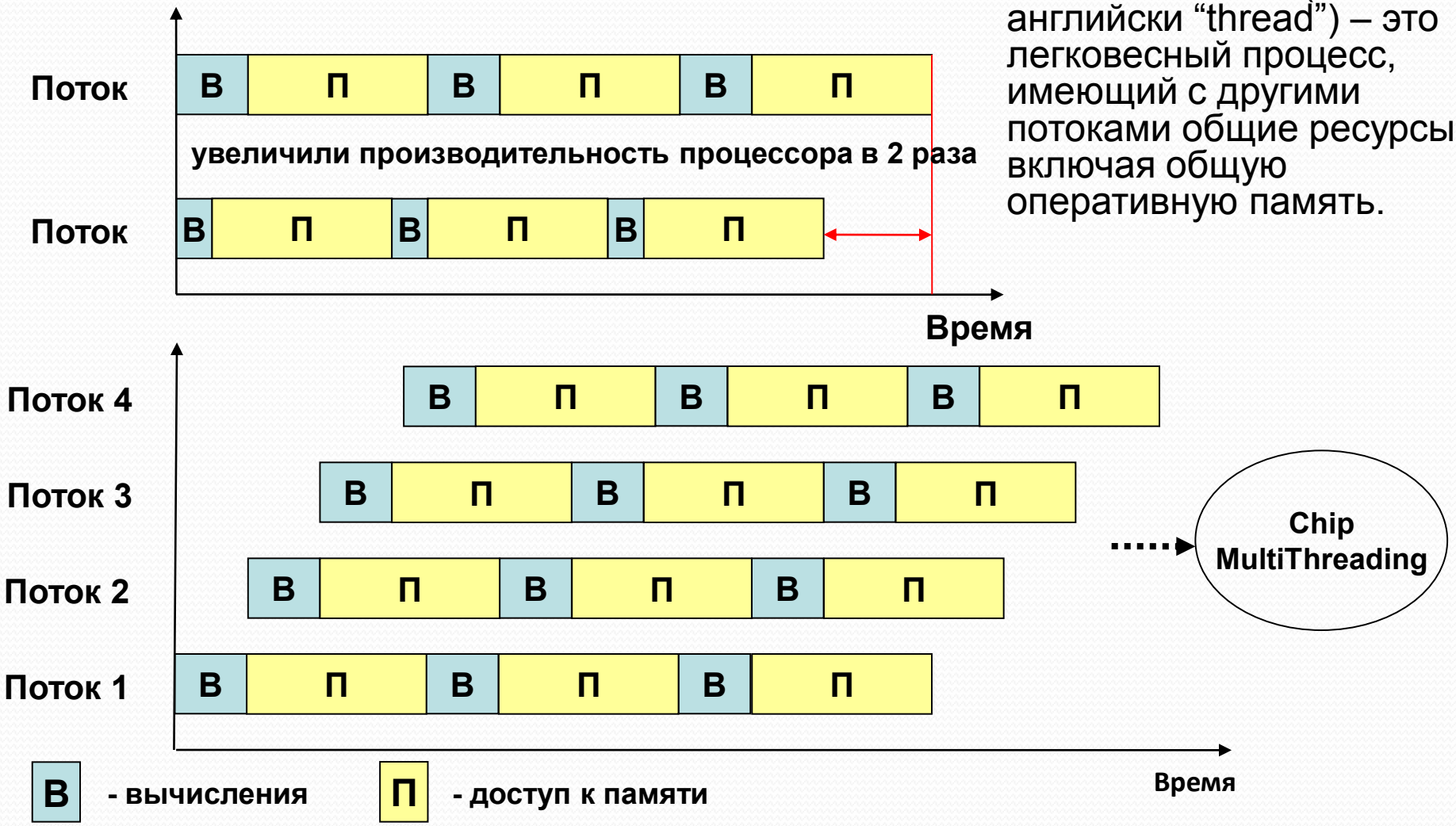
Latency to L3: 12 - 21 cycles

Latency to memory: 180 – 225 cycles

Важным параметром становится - **GUPS** (Giga Updates Per Second)

Тенденции развития современных процессоров

Поток или **нить** (по-английски “thread”) – это легковесный процесс, имеющий с другими потоками общие ресурсы, включая общую оперативную память.



Тенденции развития современных процессоров

Суперкомпьютер Jaguar Cray XT5-HE Opteron Six Core 2.6 GHz

- Пиковая производительность - 2331 TFlop/s
- Число ядер в системе — 224 162
- Производительность на Linpack - 1759 TFlop/s (75.4% от пиковой)
- Энергопотребление комплекса - **6950.60 кВт**

Важным параметром становится – **Power Efficency (Megaflops/watt)**

Как добиться максимальной производительности на Ватт => Chip MultiProcessing, многоядерность.

Тенденции развития современных процессоров

AMD Opteron серии 6200

6284 SE 16 ядер @ 2,7 ГГц, 16 МБ L3 Cache

6220 8 ядер @ 3,0 ГГц, 16 МБ L3 Cache

6204 4 ядра @ 3,3 ГГц, 16 МБ L3 Cache

встроенный контроллер памяти (4 канала памяти DDR3)

4 канала «точка-точка» с использованием HyperTransort 3.0

Тенденции развития современных процессоров

Intel Xeon серии E5

2690 8 ядер @ 2,9 ГГц, 16 нитей, 20 МБ L3 Cache

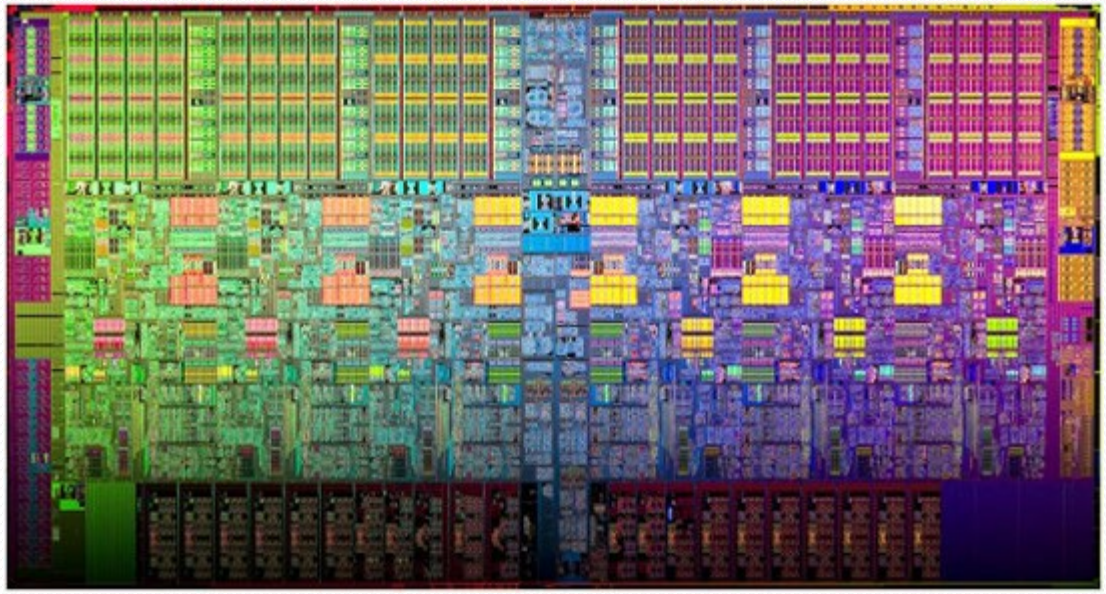
2643 4 ядра @ 3,5 ГГц, 8 нитей, 10 МБ L3 Cache

Intel® Turbo Boost

Intel® Hyper-Threading

Intel® QuickPath

Intel® Intelligent Power



Тенденции развития современных процессоров

Intel Core i7-3960X Extreme Edition

3,3 ГГц (3,9 ГГц)

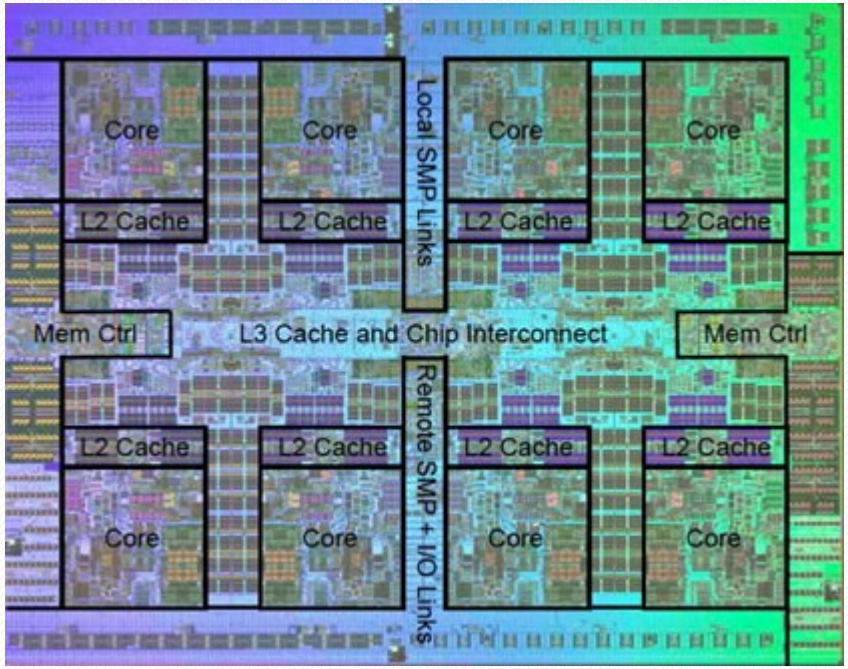
- 6 ядер
- 12 потоков с технологией Intel Hyper-Threading
- 15 МБ кэш-памяти Intel Smart Cache
- встроенный контроллер памяти (4 канала памяти DDR3 1066/1333/1600 МГц)
- технология Intel QuickPath Interconnect

Тенденции развития современных процессоров

Intel Itanium 9350 (Tukwila) 1,73 ГГц

- 4 ядра
- 8 потоков с технологией Intel Hyper-Threading
- 24 МБ L3 кэш-памяти
- технология Intel QuickPath Interconnect
- технология Intel Turbo Boost

Тенденции развития современных процессоров



IBM Power7

- 3,5 - 4,0 ГГц
- 8 ядер x 4 нити Simultaneuos MultiThreading
- L1 64КБ
- L2 256 КБ
- L3 32 МБ
- встроенный контроллер памяти

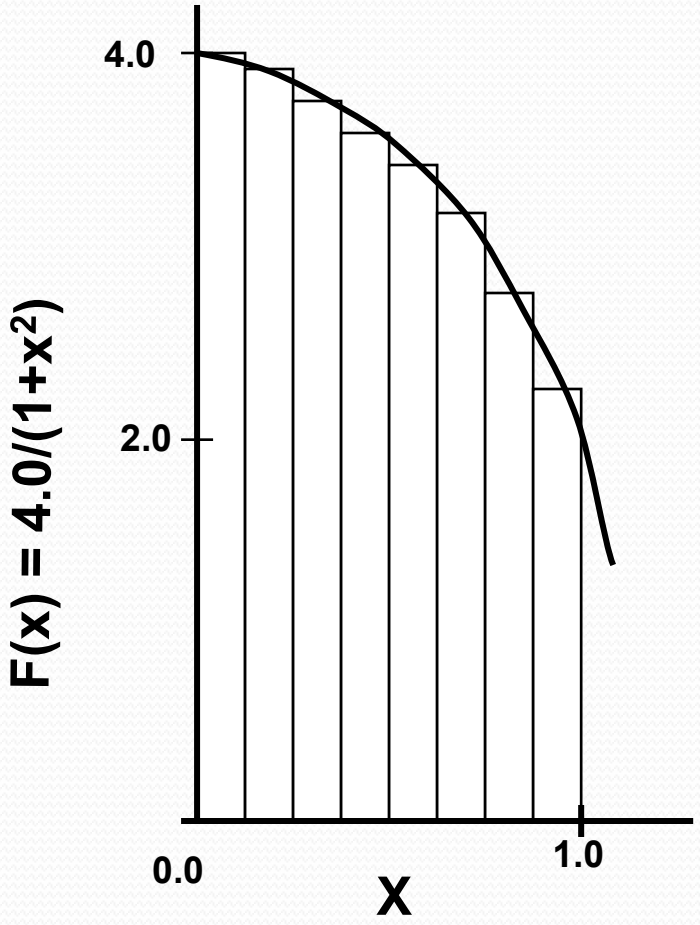
Тенденции развития современных процессоров

- Темпы уменьшения латентности памяти гораздо ниже темпов ускорения процессоров + прогресс в технологии изготовления кристаллов => CMT (Chip MultiThreading)
- Опережающий рост потребления энергии при росте тактовой частоты + прогресс в технологии изготовления кристаллов => CMP (Chip MultiProcessing, многоядерность)
- И то и другое требует более глубокого распараллеливания для эффективного использования аппаратуры

Тенденции развития современных процессоров

- Автоматическое / автоматизированное распараллеливание
- Библиотеки нитей
  - Win32 API
  - POSIX
- Библиотеки передачи сообщений
  - MPI
- OpenMP

Вычисление числа π



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

Мы можем аппроксимировать интеграл как сумму прямоугольников:

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

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

Вычисление числа π. Последовательная программа

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

Автоматическое распараллеливание

Polaris, CAPO, WPP, SUIF, VAST/Parallel, OSCAR, Intel/OpenMP, ParaWise

```
icc -parallel pi.c
pi.c(8): (col. 5) remark: LOOP WAS AUTO-PARALLELIZED.
pi.c(8): (col. 5) remark: LOOP WAS VECTORIZED.
pi.c(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.

Автоматизированное распараллеливание

Intel/GAP (Guided Auto-Parallel), CAPTools/ParaWise, BERT77, FORGE Magic/DM, ДВОР (Диалоговый Высокоуровневый Оптимизирующий Распараллеливатель), САПФОР (Система Автоматизации Параллелизации ФОРтран программ)

```
for (i=0; i<n; i++) {
    if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
    if (A[i] > 1) {A[i] += b;}
}
```

```
icc -guide -parallel test.cpp
```

Автоматизированное распараллеливание

test.cpp(49): remark #30521: (PAR) Loop at line 49 cannot be parallelized due to conditional assignment(s) into the following variable(s): b. This loop will be parallelized if the variable(s) become unconditionally initialized at the top of every iteration. [VERIFY] Make sure that the value(s) of the variable(s) read in any iteration of the loop must have been written earlier in the same iteration.

test.cpp(49): remark #30525: (PAR) If the trip count of the loop at line 49 is greater than 188, then use "#pragma loop count min(188)" to parallelize this loop. [VERIFY] Make sure that the loop has a minimum of 188 iterations.

```
#pragma loop count min (188)
for (i=0; i<n; i++) {
    b = A[i];
    if (A[i] > 0) {A[i] = 1 / A[i];}
    if (A[i] > 1) {A[i] += b;}
}
```

Вычисление числа π с использованием Win32 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 is approximately %.16f”, pi);
}
```

Вычисление числа π с использованием Win32 API

```
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);
}
```

Взаимное исключение критических интервалов

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

Thread0: pi = pi + val; && Thread1: pi = pi + val;

Время	Thread 0	Thread 1
1	LOAD R1,pi	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R1,pi
4		LOAD R2,val
5		ADD R1,R2
6		STORE R1,pi
7	STORE R1,pi	

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

Вычисление числа π с использованием MPI

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char *argv[])
{
    int n =100000, myid, numprocs, i;
   double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     h   = 1.0 / (double) n;
    sum = 0.0;
```

Вычисление числа π с использованием MPI

```
    for (i = myid + 1; i <= n; i += numprocs)
   {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf("pi is approximately %.16f”, pi);
    MPI_Finalize();
    return 0;
}
```

Вычисление числа π с использованием OpenMP

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

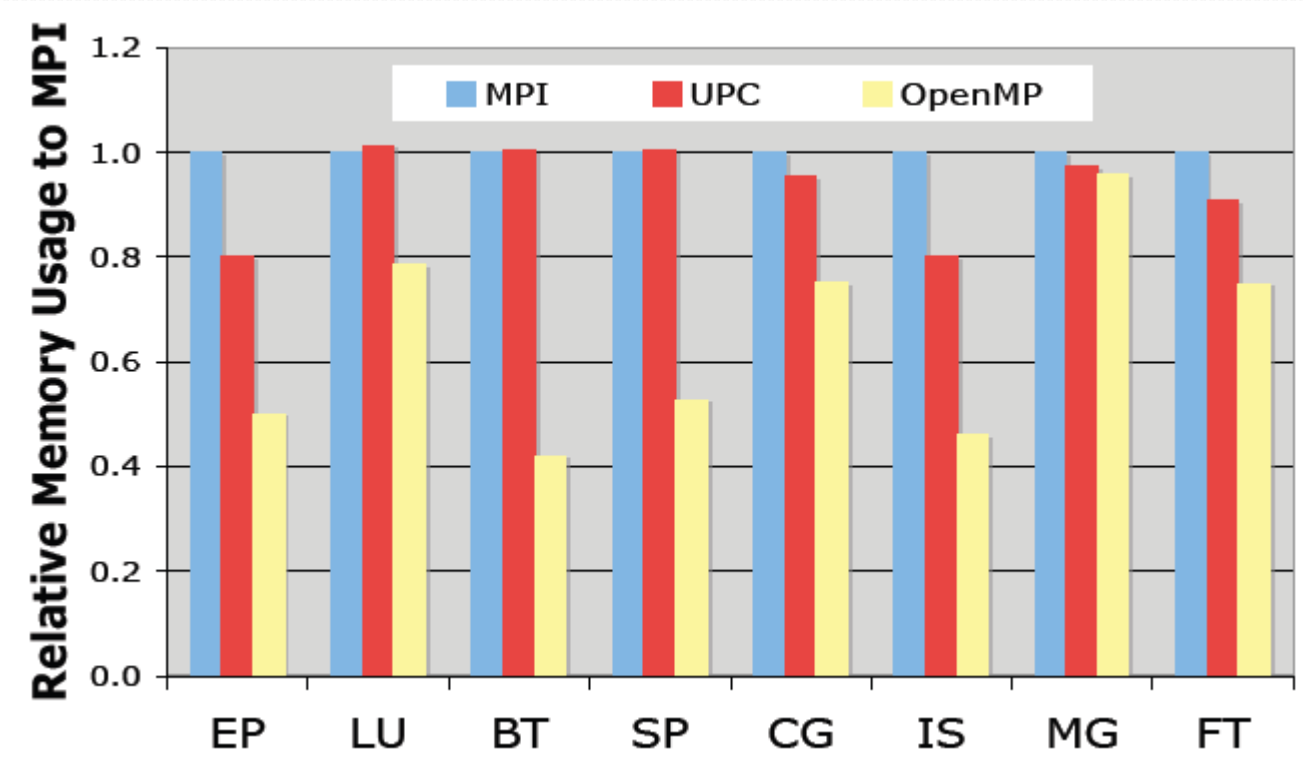
Достоинства использования OpenMP вместо MPI для многоядерных процессоров

- Возможность инкрементального распараллеливания
- Упрощение программирования и эффективность на нерегулярных вычислениях, проводимых над общими данными
- Ликвидация дублирования данных в памяти, свойственного MPI-программам
- Объем памяти пропорционален быстродействию процессора. В последние годы увеличение производительности процессора достигается удвоением числа ядер, при этом частота каждого ядра снижается. Наблюдается тенденция к сокращению объема оперативной памяти, приходящейся на одно ядро. Присущая OpenMP экономия памяти становится очень важна.
- Наличие локальных и/или разделяемых ядрами КЭШей будут учитываться при оптимизации OpenMP-программ компиляторами, что не могут делать компиляторы с последовательных языков для MPI-процессов.

Тесты NAS

BT	3D Навье-Стокс, метод переменных направлений
CG	Оценка наибольшего собственного значения симметричной разреженной матрицы
EP	Генерация пар случайных чисел Гаусса
FT	Быстрое преобразование Фурье, 3D спектральный метод
IS	Параллельная сортировка
LU	3D Навье-Стокс, метод верхней релаксации
MG	3D уравнение Пуассона, метод Multigrid
SP	3D Навье-Стокс, Beam-Warning approximate factorization

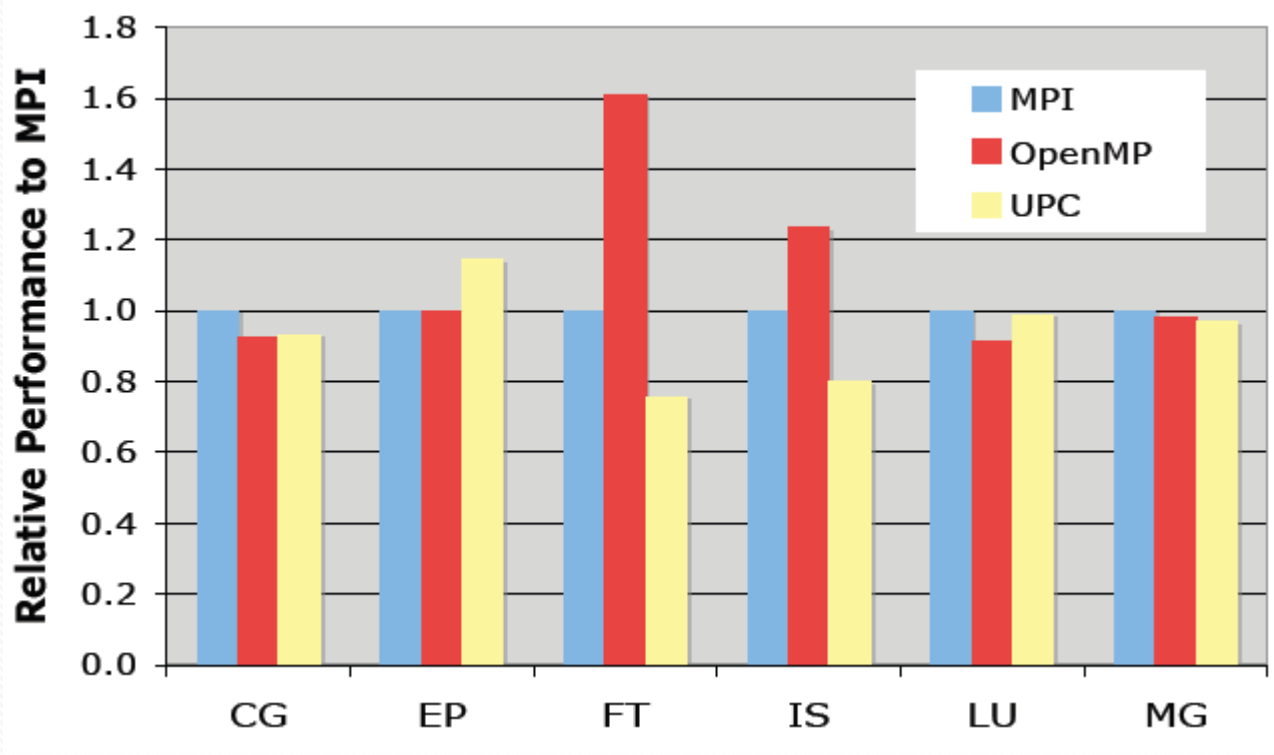
Тесты NAS



Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms

https://www.nersc.gov/assets/NERSC-Staff-Publications/2010/Cug2010Shan.pdf

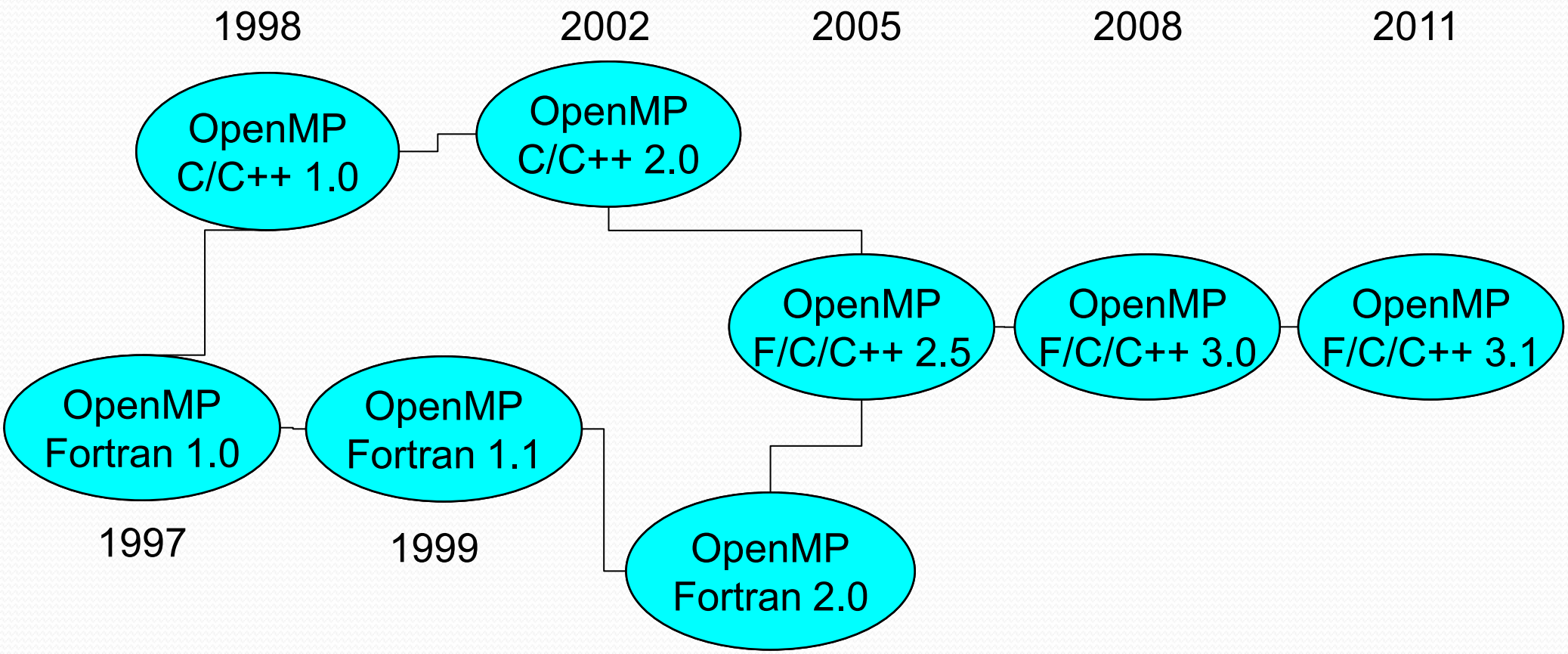
Тесты NAS



Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms

https://www.nersc.gov/assets/NERSC-Staff-Publications/2010/Cug2010Shan.pdf

История OpenMP



OpenMP Architecture Review Board

- AMD
- Cray
- Fujitsu
- HP
- IBM
- Intel
- NEC
- The Portland Group, Inc.
- Oracle Corporation
- Microsoft
- Texas Instrument
- CAPS-Enterprise
- NVIDIA
- Convey Computer
- ANL
- ASC/LLNL
- cOMPunity
- EPCC
- LANL
- NASA
- RWTH Aachen University
- Texas Advanced Computing Center

Компиляторы, поддеживающие OpenMP

OpenMP 3.1:

- Intel 12.0: Linux, Windows and MacOS
- Oracle Solaris Studio12.3: Linux and Solaris
- GNU gcc (4.7.0)

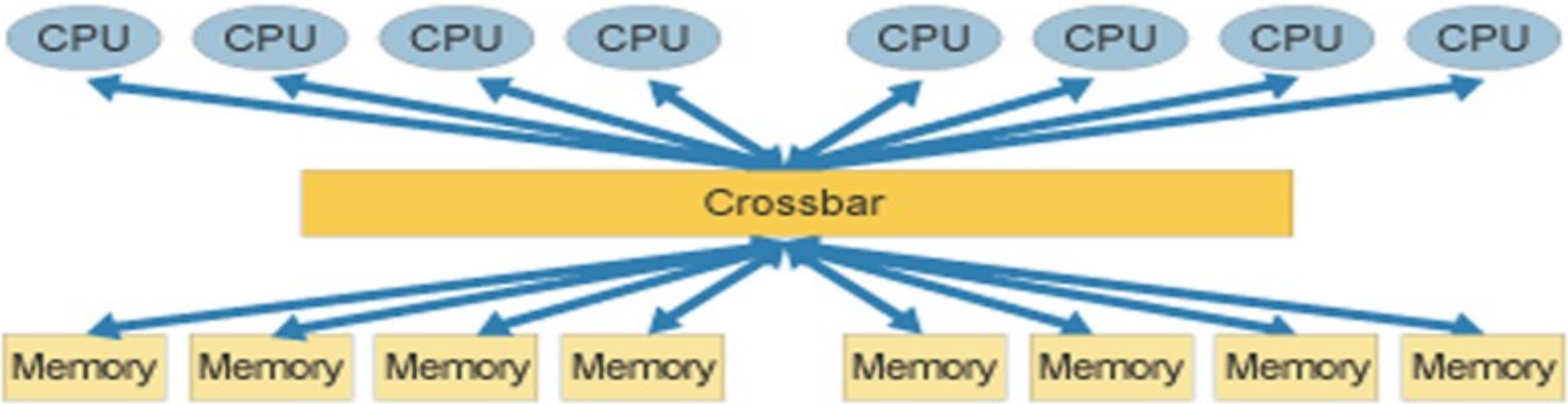
OpenMP 3.0:

- PGI 8.0: Linux and Windows
- IBM 10.1: Linux and AIX
- Cray: Cray XT series Linux environment
- Absoft Pro FortranMP: 11.1
- NAG Fortran Complier 5.3

Предыдущие версии OpenMP:

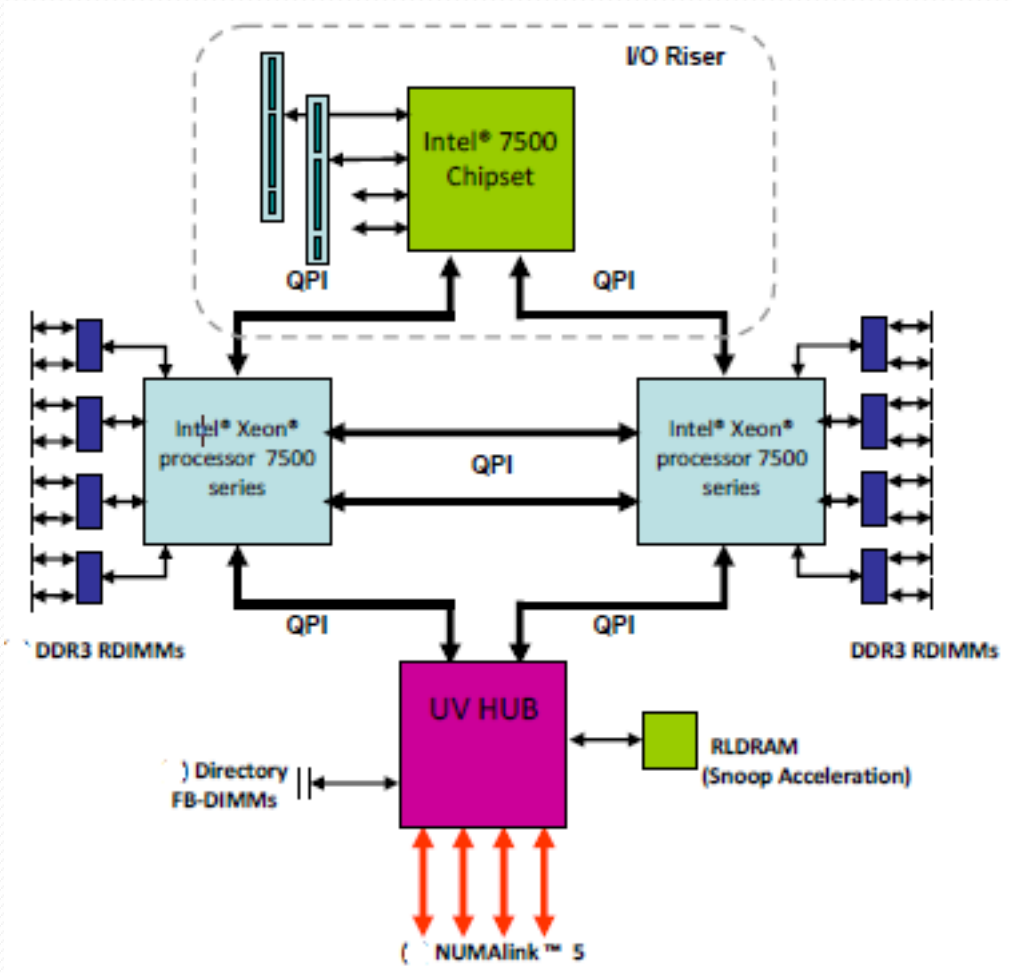
- Lahey/Fujitsu Fortran 95
- PathScale
- HP
- Microsoft Visual Studio 2008 C++

Симметричные мультипроцессорные системы (SMP)



- Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью.
- Процессоры подключены к памяти либо с помощью общей шины, либо с помощью crossbar-коммутатора.
- Аппаратно поддерживается когерентность кэшей.

Системы с неоднородным доступом к памяти (NUMA)



- Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти.
- Модули объединены с помощью высокоскоростного коммутатора.
- Поддерживается единое адресное пространство.
- Доступ к локальной памяти в несколько раз быстрее, чем к удаленной.

Системы с неоднородным доступом к памяти (NUMA)

SGI Altix UV (UltraVioloet) 1000

- 256 Intel® Xeon® quad-, six- or eight-core 7500 series (2048 cores)
- 16 TB памяти
- Interconnect Speed 15 ГБ/с, 1мкс

http://www.sgi.com/products/servers/altix/uv/

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы

OpenMP- модель параллелизма по управлению

- Основные понятия
- Выполнение OpenMP-программы
- Классы переменных
- Параллельная область

Обзор основных возможностей OpenMP

```
C$OMP FLUSH
C$OMP THREADPRIVATE(/ABC/)
C$OMP PARALLEL DO SHARED(A,B,C
CALL OMP_INIT_LOCK (LCK)
C$OMP SINGLE PRIVATE(X)
SET
C$OMP PARALLEL DO ORDERED PRI
C$OMP PARALLEL  REDUCTION (+:
#pragma omp parallel for private(a, b)
C$OMP BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO LASTPRIVATE(XX)
nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

OpenMP: API для написания многонитевых приложений

- Множество директив компилятора, набор функции библиотеки системы поддержки, переменные окружения
- Облегчает создание многонитиевых программ на Фортране, C и C++
- Обобщение опыта создания параллельных программ для SMP и DSM систем за последние 20 лет

Директивы и клаузы

Спецификации параллелизма в OpenMP представляют собой директивы вида:

#pragma omp название-директивы*[клауза[[,]клауза]...]*

Например:

#pragma omp parallel default (none) shared (i,j)

Исполняемые директивы:

- ***barrier***
- ***taskwait***
- ***flush***

Описательная директива:

- ***threadprivate***

Структурный блок

Действие остальных директив распространяется на структурный блок:

#pragma omp название-директивы*[клауза[[,]клауза]...]*

{

структурный блок

}

Структурный блок: блок кода с одной точкой входа и одной точкой выхода.

```
#pragma omp parallel
{
  …
  mainloop: res[id] = f (id);
  if (res[id] != 0) goto mainloop;
  …
  exit (0);
}
```

Структурный блок

```
#pragma omp parallel
{
  …
  mainloop: res[id] = f (id);
  …
}
if (res[id] != 0) goto mainloop;
```

Не структурный блок

Компиляция OpenMP-программы

Производитель	Компилятор	Опция компиляции
GNU	gcc	-fopenmp
IBM	XL C/C++ / Fortran	-qsmp=omp
Sun Microsystems	C/C++ / Fortran	-xopenmp
Intel	C/C++ / Fortran	-openmp /Qopenmp
Portland Group	C/C++ / Fortran	-mp
Microsoft	Visual Studio 2008 C++	/openmp

Условная компиляция OpenMP-программы

```
#include <stdio.h>
int main()
{
#ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
#endif
    return 0;
}
```

В значении переменной _OPENMP зашифрован год и месяц (yyyymm) версии стандарта OpenMP, которую поддерживает компилятор.

Использование функций поддержи выполнения OpenMP-программ (OpenMP API runtime library)

```
#include <stdio.h>
#include <omp.h> // Описаны прототипы всех функций и типов
int main()
{
#pragma omp parallel
  {
    int id = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    printf("Thread (%d) of (%d) threads alive\n", id, numt);
  }
  return 0;
}
```

Использование функций поддержки выполнения OpenMP-программ (OpenMP API runtime library)

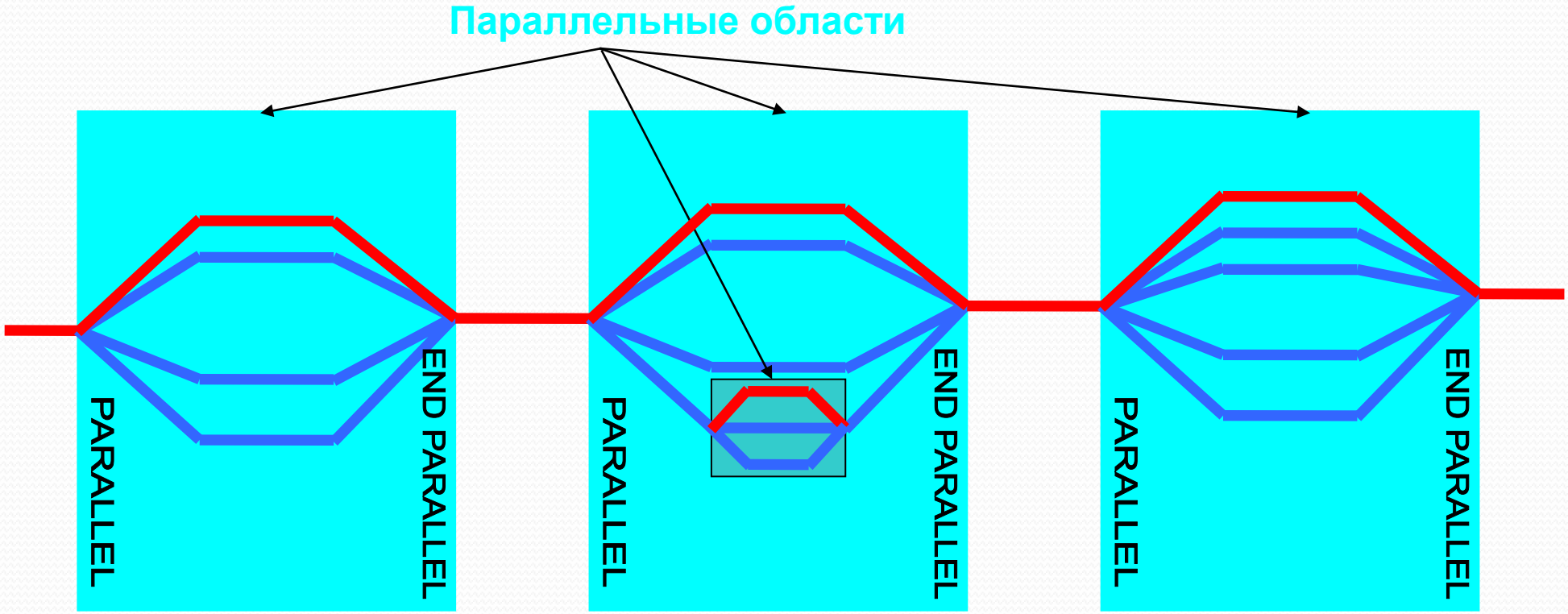
```
int omp_get_num_threads(void)
{
   return 1;
}
int omp_get_thread_num(void)
{
    return 0;
}
int main()
{
#pragma omp parallel
   {
      int id = omp_get_thread_num ();
      int numt = omp_get_num_threads ();
      printf("Thread (%d) of (%d) threads alive\n", id, numt);
    }
   return 0;
}
```

В стандарте OpenMP описаны «заглушки» для всех функций библиотеки поддержки – требуются при компиляции данной программы компилятором без поддержки OpenMP.

Выполнение OpenMP-программы

Fork-Join параллелизм:

- Главная (master) нить порождает группу (team) нитей по мере небходимости.
- Параллелизм добавляется инкрементально.



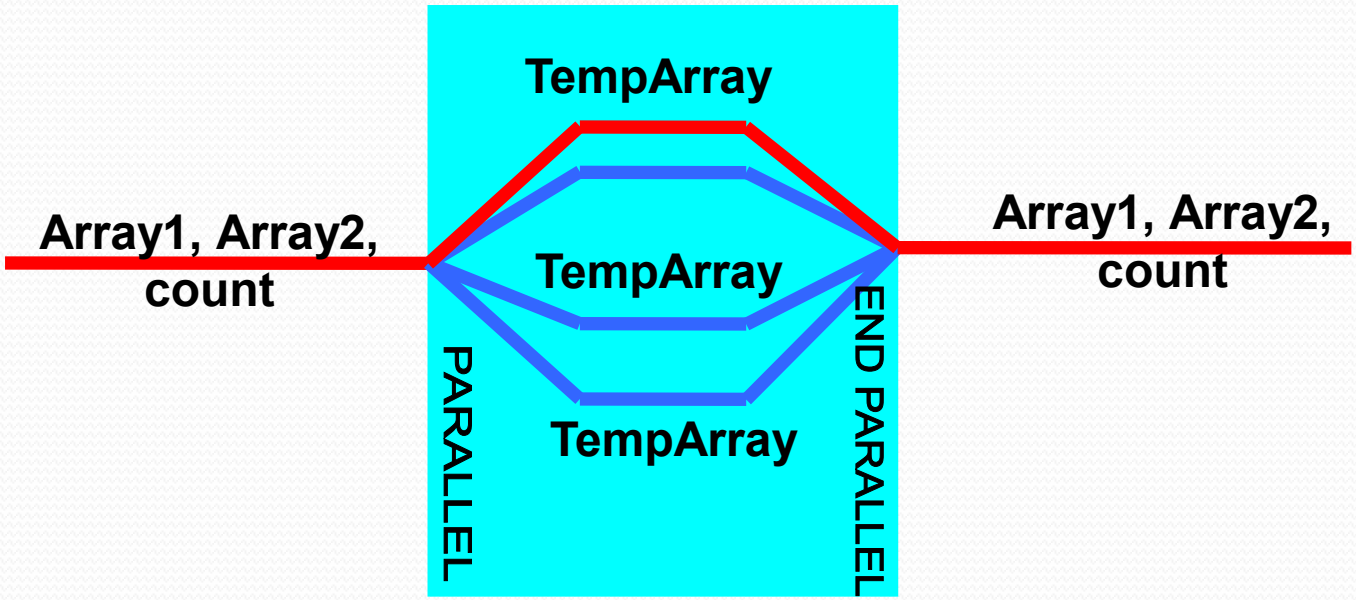
Классы переменных

- В модели программирования с разделяемой памятью:
  - Большинство переменных по умолчанию считаются **shared**
- Глобальные переменные совместно используются всеми нитями (shared) :
  - Фортран: COMMON блоки, SAVE переменные, MODULE переменные
  - Си: file scope, static
  - Динамически выделяемая память (ALLOCATE, malloc, new)
- Но не все переменные являются разделяемыми ...
  - Стековые переменные в подпрограммах (функциях), вызываемых из параллельного региона, являются **private**.
  - Переменные объявленные внутри блока операторов параллельного региона являются приватными.
  - Счетчики циклов витки которых распределяются между нитями при помощи конструкций **for** и **parallel for**.

Классы переменных

```
double Array1[100];
int main() {
    int Array2[100];
#pragma omp parallel
   work(Array2);
   printf("%d\n", Array2[0]);
}
```

```
extern double Array1[10];
void work(int *Array) {
    double TempArray[10];
    static int count;
    ...
}
```



Классы переменных

Можно изменить класс переменной при помощи конструкций:

- **shared** (список переменных)
- **private** (список переменных)
- **firstprivate** (список переменных)
- **lastprivate** (список переменных)
- **threadprivate** (список переменных)
- **default** (**private** | **shared** | **none**)

Конструкция private

- Конструкция «private(var)» создает локальную копию переменной «var» в каждой из нитей.
  - Значение переменной не инициализировано
  - Приватная копия не связана с оригинальной переменной
  - В OpenMP 2.5 значение переменной «var» не определено после завершения параллельной конструкции

```
#pragma omp parallel for private (i,j,sum)
for (i=0; i< m; i++)
{
   sum = 0.0;
   for (j=0; j< n; j++)
     sum +=b[i][j]*c[j];
   a[i] = sum;
}
```

Конструкция firstprivate

- «firstprivate» является специальным случаем «private»

  Инициализирует каждую приватную копию соответствующим значением из главной (master) нити.

```
BOOL FirstTime=TRUE;
#pragma omp parallel for firstprivate(FirstTime)
for (row=0; row<height; row++)
{
  if (FirstTime == TRUE)  { FirstTime = FALSE;   FirstWork (row); }
 AnotherWork (row);
}
```

Конструкция lastprivate

- lastprivate передает значение приватной переменной, посчитанной на последней итерации в глобальную переменную.

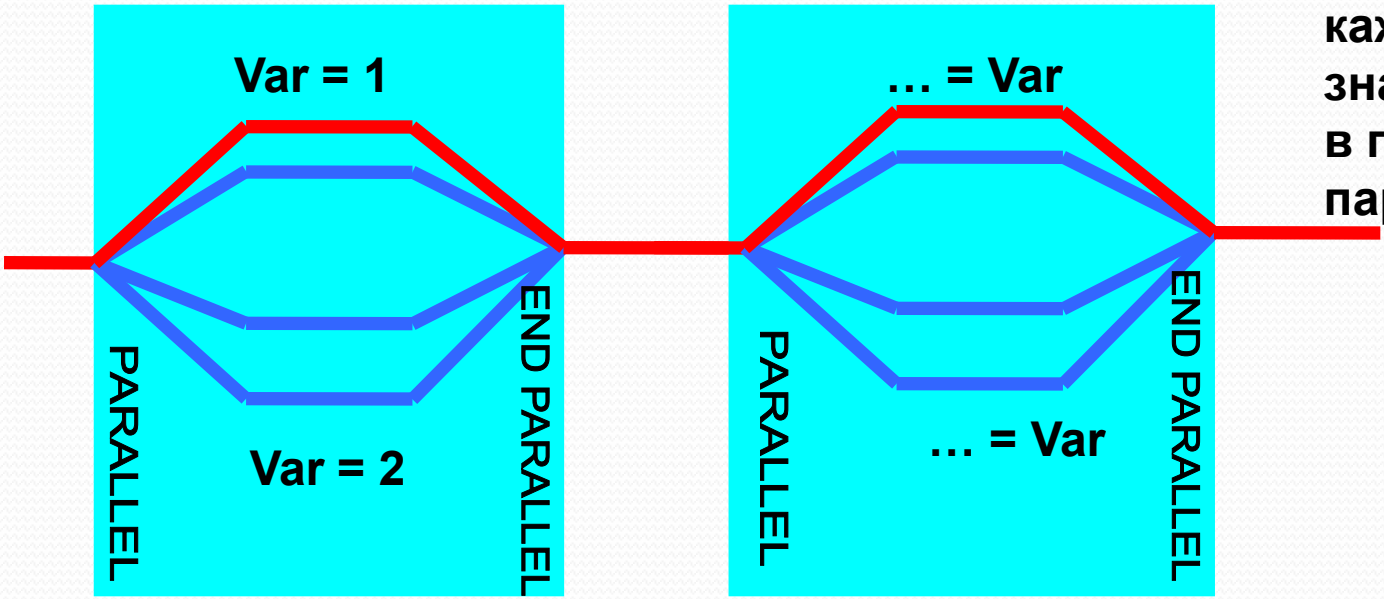
```
int i;
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];

}
a[i]=b[i];   /*i == n-1*/
```

Директива threadprivate

Отличается от применения конструкции **private**:

- **private** скрывает глобальные переменные
- **threadprivate** – переменные сохраняют глобальную область видимости внутри каждой нити

#pragma omp threadprivate (Var)



Если количество нитей не изменилось, то каждая нить получит значение, посчитанное в предыдущей параллельной области.

Конструкция default

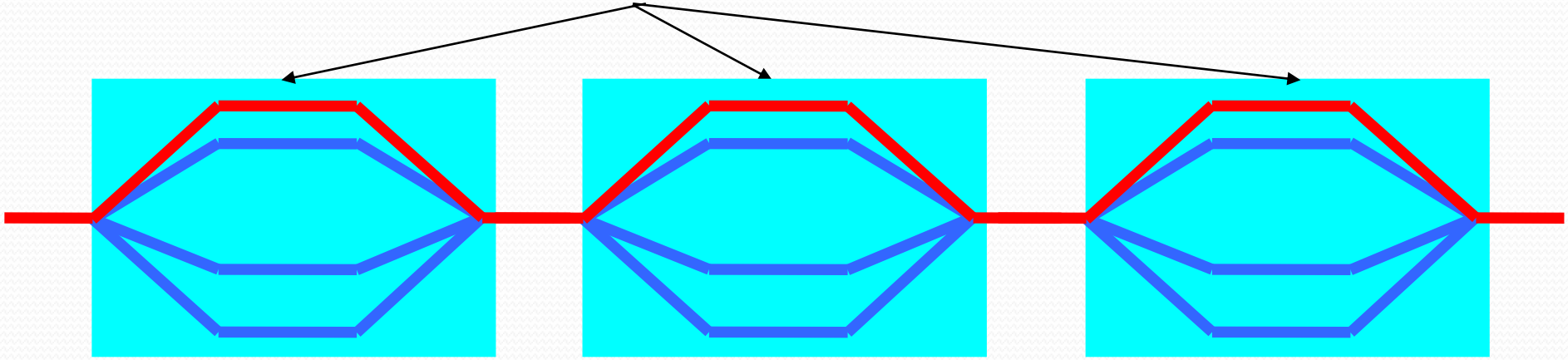
Меняет класс переменной по умолчанию:

- **default (shared)** – действует по умолчанию
- **default (private)** – есть только в Fortran
- **default (firstprivate)** – есть только в Fortran OpenMP 3.1
- **default (none)** – требует определить класс для каждой переменной

```
itotal = 100
#pragma omp parallel
private(np,each)
{
np = omp_get_num_threads()
each = itotal/np
……
}
```

```
itotal = 100
#pragma omp parallel default(none)
private(np,each) shared (itotal)
{
np = omp_get_num_threads()
each = itotal/np
……
}
```

Параллельная область (директива parallel)

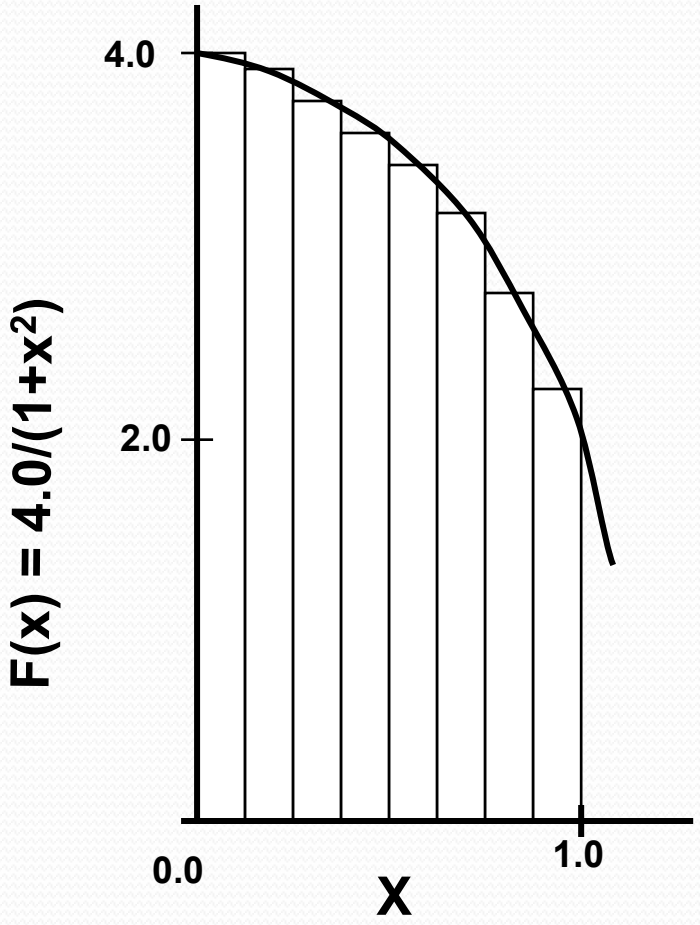


#pragma omp parallel *[клауза[[,] клауза] ...]*
структурный блок

где клауза одна из :

- **default(shared | none)**
- **private(***list***)**
- **firstprivate(***list***)**
- **shared(***list***)**
- **reduction(***operator***: *list***)**
- **if(***scalar-expression***)**
- **num_threads(***integer-expression***)**
- **copyin(***list***)**

Вычисление числа π



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

Мы можем аппроксимировать интеграл как сумму прямоугольников:

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

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

Вычисление числа π. Последовательная программа

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

Вычисление числа π на OpenMP

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

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 32
int main ()
{
   int n =100000, i;
   double pi, h, sum[NUM_THREADS], x;
   h   = 1.0 / (double) n;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
   {
      int id = omp_get_thread_num();
      int numt = omp_get_num_threads();
      for (i = id + 1, sum[id] = 0.0; i <= n;  i=i+numt)
      {
            x = h * ((double)i - 0.5);
            sum[id] += (4.0 / (1.0 + x*x));
      }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * h;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP. Клауза reduction

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

Клауза reduction

reduction(operator:list)

- Внутри паралельной области для каждой переменной из списка list создается копия этой переменной. Эта переменная инициализируется в соответствии с оператором operator (например, 0 для «+»).
- Для каждой нити компилятор заменяет в параллельной области обращения к редукционной переменной на обращения к созданной копии.
- По завершении выполнения параллельной области осуществляется объединение полученных результатов.

Оператор	Начальное значение
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0
max	Least number in reduction list item type
min	Largest number in reduction list item type

Клауза if

if(*scalar-expression*)

В зависимости от значения *scalar-expression* для выполнения структурного блока будет создана группа нитей или он будет выполняться одной нитью.

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int n = 0;
  printf("Enter the number of intervals: (0 quits) ");
  scanf("%d",&n);
#pragma omp parallel if (n>10)
   {
      int id = omp_get_thread_num ();
      func (n, id);
   }
   return 0;
}
```

Клауза num_threads

num_threads(*integer-expression)*

integer-expression задает максимально возможное число нитей, которые будут созданы для выполнения структурного блока

```
#include <omp.h>
int main()
{
  int n = 0;
  printf("Enter the number of intervals: (0 quits) ");
  scanf("%d",&n);
 omp_set_dynamic(1);
#pragma omp parallel num_threads(10)
   {
      int id = omp_get_thread_num ();
      func (n, id);
   }
   return 0;
}
```

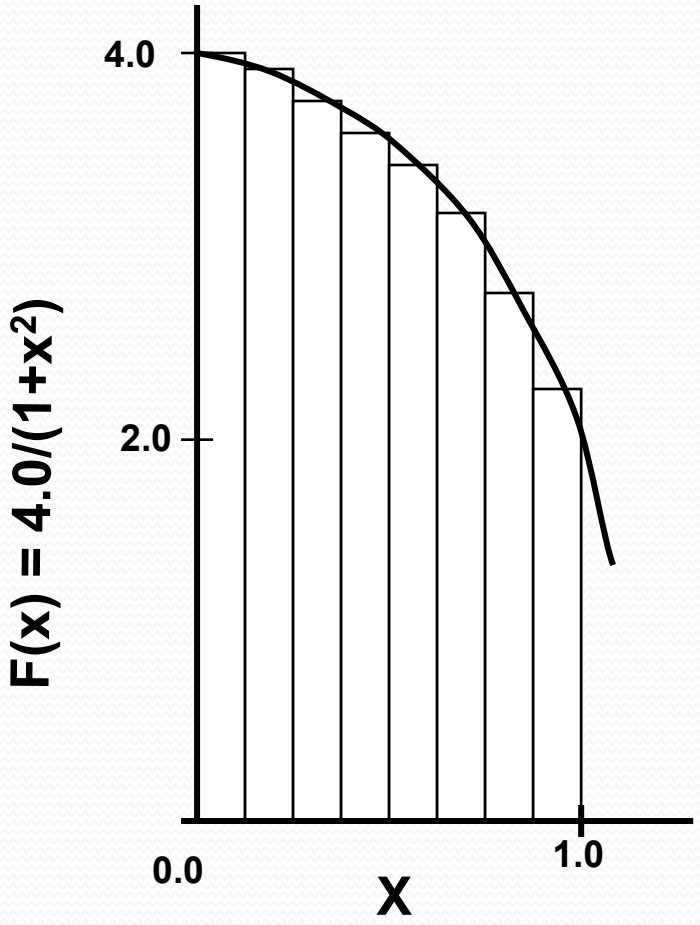
Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы

Конструкции распределения работы

- Распределение витков циклов (директива for)
- Выполнение структурного блока одной нитью (директива single)

Вычисление числа π



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

Мы можем аппроксимировать интеграл как сумму прямоугольников:

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

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

Вычисление числа π. Последовательная программа

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

Вычисление числа π на OpenMP

```
int main ()
{
   int n =100000, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
   {
      int iam = omp_get_thread_num();
      int numt = omp_get_num_threads();
      int start = iam * n / numt + 1;
      int end = (iam + 1) * n / numt;
      if (iam == numt-1) end = n;
      for (i = start; i <= end;  i++)
      {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
       }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
   int n =100, i;
   double pi, h, sum, x;
   h   = 1.0 / (double) n;
   sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
   {
#pragma omp for schedule (static)
for (i = 1; i <= n;  i++)
      {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
       }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Распределение витков цикла

#pragma omp for *[клауза[[,]клауза] ...]*
for (*init-expr***;** *test-expr***;** *incr-expr***)** *структурный блок*

где клауза одна из :

- **private(***list***)**
- **firstprivate(***list***)**
- **lastprivate(***list***)**
- **reduction(***operator***:** *list***)**
- **schedule(***kind[, chunk_size]***)**
- **collapse(***n***)**
- **ordered**
- **nowait**

Распределение витков цикла

init-expr : var = loop-invariant-expr1
| integer-type var = loop-invariant-expr1
| random-access-iterator-type var = loop-invariant-expr1
| pointer-type var = loop-invariant-expr1

test-expr:
var relational-op loop-invariant-expr2
| loop-invariant-expr2 relational-op var

incr-expr: ++var
| var++
| --var
| var --
| var += loop-invariant-integer- expr
| var -= loop-invariant-integer- expr
| var = var + loop-invariant-integer- expr
| var = loop-invariant-integer- expr + var
| var = var - loop-invariant-integer- expr

relational-op: <
| <=
| >
| >=

var: signed or unsigned integer type
| random access iterator type
| pointer type

Parallel Random Access Iterator Loop (OpenMP 3.0)

```
#include <vector>
void iterator_example()
{
  std::vector<int> vec(23);
  std::vector<int>::iterator it;
#pragma omp parallel for default(none) shared(vec)
  for (it = vec.begin(); it < vec.end(); it++)
  {
    // do work with *it //
  }
}
```

Использование указателей в цикле (OpenMP 3.0)

```
void pointer_example ()
{
   char a[N];
#pragma omp for default (none) shared (a,N)
   for (char *p = a; p < (a+N); p++ )
   {
       use_char (p);
   }
}
```

for (char *p = a; p != (a+N); p++) - ошибка

Вложенность конструкций распределения работы

```
void work(int i, int j) {}
void wrong1(int n)
{
#pragma omp parallel default(shared)
  {
    int i, j;
    #pragma omp for
    for (i=0; i < n; i++) {
      /* incorrect nesting of loop regions */
      #pragma omp for
        for (j=0; j < n; j++)
          work(i, j);
    }
  }
}
```

Вложенность конструкций распределения работы

```
void work(int i, int j) {}
void good_nesting(int n)
{
   int i, j;
   #pragma omp parallel default(shared)
   {
      #pragma omp for
      for (i=0; i < n; i++) {
        #pragma omp parallel shared(i, n)
          {
            #pragma omp for
             for (j=0; j < n; j++)
                work(i, j);
          }
      }
   }
}
```

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void good_collapsing(int n)
{
   int i, j;
   #pragma omp parallel default(shared)
   {
      #pragma omp for collapse (2)
      for (i=0; i<n; i++) {
            for (j=0; j < n; j++)
               work(i, j);
      }
    }
}
```

Клауза collapse:
collapse (*положительная целая константа*)

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void error_collapsing(int n)
{
   int i, j;
   #pragma omp parallel default(shared)
   {
      #pragma omp for collapse (2)
      for (i=0; i<n; i++) {
            work_with_i (i);                    //  Ошибка
            for (j=0; j < n; j++)
                work(i, j);
      }
    }
}
```

Клауза collapse может быть использована только для распределения витков тесно-вложенных циклов.

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void error_collapsing(int n)
{
   int i, j;
   #pragma omp parallel default(shared)
   {
      #pragma omp for collapse (2)
      for (i=0; i<n; i++) {
            for (j=0; j < i; j++)          // Ошибка
               work(i, j);
      }
    }
}
```

Клауза collapse может быть использована только для распределения витков циклов с прямоугольным индексным пространством.

Распределение витков цикла. Клауза schedule

Клауза schedule:
schedule(*алгоритм планирования[, число_итераций]*)

Где алгоритм планирования один из:

- **schedule(static*[, число_итераций]*) - статическое планирование;**
- **schedule(dynamic*[, число_итераций]*) - динамическое планирование;**
- **schedule(guided*[, число_итераций]*) - управляемое планирование;**
- **schedule(runtime) - планирование в период выполнения;**
- **schedule(auto) - автоматическое планирование (OpenMP 3.0).**

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(static, 10)
   for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре будет следующим:

- **Поток 0 получает право на выполнение итераций 1-10, 41-50, 81-90.**
- **Поток 1 получает право на выполнение итераций 11-20, 51-60, 91-100.**
- **Поток 2 получает право на выполнение итераций 21-30, 61-70.**
- **Поток 3 получает право на выполнение итераций 31-40, 71-80**

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(dynamic, 15)
  for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- Поток 0 получает право на выполнение итераций 1-15.
- Поток 1 получает право на выполнение итераций 16-30.
- Поток 2 получает право на выполнение итераций 31-45.
- Поток 3 получает право на выполнение итераций 46-60.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 61-75.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 76-90.
- Поток 0 завершает выполнение итераций.
- Поток 0 получает право на выполнение итераций 91-100.

Распределение витков цикла. Клауза schedule

число_выполняемых_потоком_итераций =
max(число_нераспределенных_итераций/omp_get_num_threads(),
число_итераций)

```
#pragma omp parallel for schedule(guided, 10)
  for(int i = 1; i <= 100; i++)
```

Пусть программа запущена на 4-х ядерном процессоре.

- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-44.
- Поток 2 получает право на выполнение итераций 45-59.
- Поток 3 получает право на выполнение итераций 60-69.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 70-79.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 80-89.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 90-99.
- Поток 1 завершает выполнение итераций.
- Поток 1 получает право на выполнение 100 итерации.

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(runtime)
   for(int i = 1; i <= 100; i++)  /* способ распределения витков цикла между
     нитями будет задан во время выполнения программы*/
```

При помощи переменных среды:

csh:

setenv OMP_SCHEDULE "dynamic,4“

ksh:

export OMP_SCHEDULE="static,10“

Windows:

set OMP_SCHEDULE=auto

или при помощи функции системы поддержки:

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

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(auto)
  for(int i = 1; i <= 100; i++)
```

Способ распределения витков цикла между нитями определяется реализацией компилятора.

На этапе компиляции программы или во время ее выполнения определяется оптимальный способ распределения.

Распределение витков цикла. Клауза nowait

```
void example(int n, float *a, float *b, float *z)
{
   int i;
   #pragma omp parallel
   {
      #pragma omp for schedule(static) nowait
      for (i=0; i<n; i++)
         c[i] = (a[i] + b[i]) / 2.0;
      #pragma omp for schedule(static) nowait
       for (i=0; i<n; i++)
          z[i] = sqrt(c[i]);
   }
}
```

Неверно в OpenMP 2.5

Верно в OpenMP 3.0, если количество итераций у циклов совпадает и параметры клаузы schedule совпадают (STATIC + *число_итераций*).

Распределение витков цикла. Клауза nowait

```
void example(int n, float *a, float *b, float *z)
{
   int i;
   float sum = 0.0;
   #pragma omp parallel
   {
      #pragma omp for schedule(static) nowait reduction (+: sum)
      for (i=0; i<n; i++) {
         c[i] = (a[i] + b[i]) / 2.0;
         sum += c[i];
      }
      #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
           z[i] = sqrt(c[i]);
       #pragma omp barrier
         … = sum

   }
}
```

Выполнение структурного блока одной нитью (директива single)

#pragma omp single *[клауза[[,] клауза] ...]*
структурный блок

где клауза одна из :

- **private(***list***)**
- **firstprivate(***list***)**
- **copyprivate(***list***)**
- **nowait**

Структурный блок будет выполнен одной из нитей. Все остальные нити будут дожидаться результатов выполнения блока, если не указана клауза NOWAIT.

```
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)
void init(float *a, float *b ) {
 #pragma omp single copyprivate(a,b,x,y)
     scanf("%f %f %f %f", a, b, &x, &y);
}
int main () {
  #pragma omp parallel
  {
     float x1,y1;
     init (&x1,&y1);
     parallel_work ();
  }
}
```

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы

Конструкции для синхронизации нитей

- Директива master
- Директива critical
- Директива atomic
- Директива barrier

Директива master

#pragma omp master

структурный блок

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

```
#include <stdio.h>
void init(float *a, float *b ) {
  #pragma omp master
      scanf("%f %f", a, b);
  #pragma omp barrier
}
int main () {
   float x,y;
   #pragma omp parallel
   {
       init (&x,&y);
       parallel_work (x,y);
   }
}
```

Взаимное исключение критических интервалов

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

Thread0: pi = pi + val; && Thread1: pi = pi + val;

Время	Thread 0	Thread 1
1	LOAD R1,pi	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R1,pi
4		LOAD R2,val
5		ADD R1,R2
6		STORE R1,pi
7	STORE R1,pi	

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

Взаимное исключение критических интервалов

Решение проблемы взаимного исключения должно удовлетворять требованиям:

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

Вычисление числа π. Последовательная программа

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

Вычисление числа π на OpenMP с использованием критической секции

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

Директива critical

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

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

Директива atomic

#pragma omp atomic
expression-stmt
где expression-stmt:

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

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

где binop - не перегруженный оператор:

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

Вычисление числа π на OpenMP с использованием директивы atomic

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

Директива barrier

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

#pragma omp barrier

По умолчанию барьерная синхронизация нитей выполняется:

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

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

Директива barrier

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

Директива barrier

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

Директива barrier

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

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы

Система поддержки выполнения OpenMP-программ

- Внутренние переменные, управляющие выполнением OpenMP-программы (ICV-Internal Control Variables).
- Задание/опрос значений ICV-переменных.
- Функции работы со временем.

Internal Control Variables

Для параллельных областей:

- *nthreads-var*
- *thread-limit-var*
- *dyn-var*
- *nest-var*
- *max-active-levels-var*

Для циклов:

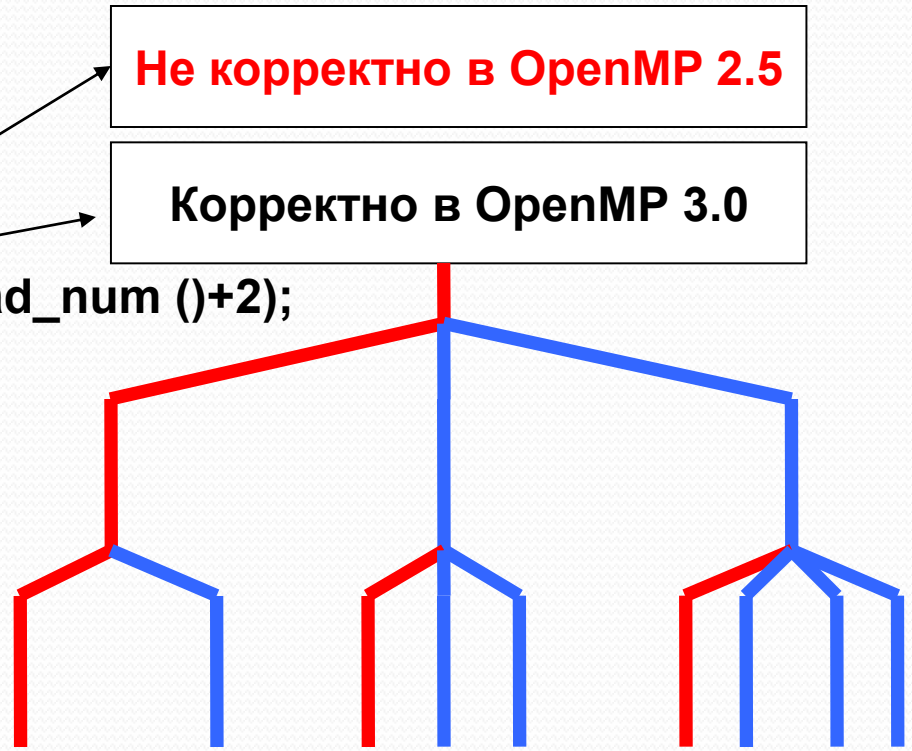
- *run-sched-var*
- *def-sched-var*

Для всей программы:

- *stacksize-var*
- *wait-policy-var*

Internal Control Variables. nthreads-var

```
void work();

int main () {
   omp_set_num_threads(3);
   #pragma omp parallel
   {
      omp_set_num_threads(omp_get_thread_num ()+2);
      #pragma omp parallel
         work();
   }
}
```



Internal Control Variables. nthreads-var

Определяет максимально возможное количество нитей в создаваемой параллельной области.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_NUM_THREADS 4,3,2

Korn shell:

export OMP_NUM_THREADS=16

Windows:

set OMP_NUM_THREADS=16

void omp_set_num_threads(int num_threads);

Узнать значение переменной можно:

int omp_get_max_threads(void);

Internal Control Variables. thread-limit-var

Определяет максимальное количество нитей, которые могут быть использованы для выполнения всей программы.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

C shell:

setenv OMP_THREAD_LIMIT 16

Korn shell:

export OMP_THREAD_LIMIT=16

Windows:

set OMP_THREAD_LIMIT=16

Узнать значение переменной можно:

int omp_get_thread_limit(void)

Internal Control Variables. dyn-var

Включает/отключает режим, в котором количество создаваемых нитей при входе в параллельную область может меняться динамически.

Начальное значение: Если компилятор не поддерживает данный режим, то false.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_DYNAMIC true

Korn shell:

export OMP_DYNAMIC=true

Windows:

set OMP_DYNAMIC=true

void omp_set_dynamic(int dynamic_threads);

Узнать значение переменной можно:

int omp_get_dynamic(void);

Internal Control Variables. nest-var

Включает/отключает режим поддержки вложенного параллелизма.

Начальное значение: **false**.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_NESTED true

Korn shell:

export OMP_NESTED=false

Windows:

set OMP_NESTED=true

void omp_set_nested(int nested);

Узнать значение переменной можно:

int omp_get_nested(void);

Internal Control Variables. max-active-levels-var

Задает максимально возможное количество активных вложенных параллельных областей.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

C shell:

setenv OMP_MAX_ACTIVE_LEVELS 2

Korn shell:

export OMP_MAX_ACTIVE_LEVELS=3

Windows:

set OMP_MAX_ACTIVE_LEVELS=4

void omp_set_max_active_levels (int max_levels);

Узнать значение переменной можно:

int omp_get_max_active_levels(void);

Internal Control Variables. run-sched-var

Задает способ распределения витков цикла между нитями, если указана клауза **schedule(runtime)**.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_SCHEDULE "guided,4"

Korn shell:

export OMP_SCHEDULE "dynamic,5"

Windows:

set OMP_SCHEDULE=static

```
typedef enum omp_sched_t {
   omp_sched_static = 1,
   omp_sched_dynamic = 2,
   omp_sched_guided = 3,
   omp_sched_auto = 4
} omp_sched_t;
```

void omp_set_schedule(omp_sched_t kind, int modifier);

Узнать значение переменной можно:

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

Internal Control Variables. run-sched-var

```
void work(int i);

int main () {
   omp_sched_t schedules [] = {omp_sched_static, omp_sched_dynamic,
omp_sched_guided, omp_sched_auto};
   omp_set_num_threads (4);
   #pragma omp parallel
   {
      omp_set_schedule (schedules[omp_get_thread_num()],0);
      #pragma omp parallel for schedule(runtime)
         for (int i=0;i<N;i++)  work (i);
   }
}
```

Internal Control Variables. def-sched-var

Задает способ распределения витков цикла между нитями по умолчанию.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

```
void work(int i);

int main () {
   #pragma omp parallel
   {
      #pragma omp for
         for (int i=0;i<N;i++)  work (i);
   }
}
```

Internal Control Variables. stack-size-var

Каждая нить представляет собой независимо выполняющийся поток управления со своим счетчиком команд, регистровым контекстом и стеком.

Переменная ***stack-size-var*** задает размер стека.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

setenv OMP_STACKSIZE 2000500B

setenv OMP_STACKSIZE "3000 k"

setenv OMP_STACKSIZE 10M

setenv OMP_STACKSIZE "10 M"

setenv OMP_STACKSIZE "20 m"

setenv OMP_STACKSIZE "1G"

setenv OMP_STACKSIZE 20000

Internal Control Variables. stack-size-var

```
int main () {
   int a[1024][1024];
   #pragma omp parallel private (a)
   {
      for (int i=0;i<1024;i++)
        for (int j=0;j<1024;j++)
          a[i][j]=i+j;
   }
}
```

icl /Qopenmp test.cpp

⇒Program Exception – stack overflow

Linux: ulimit -a

ulimit -s <stacksize in Kbytes>

Windows: /F<stacksize in bytes>

-Wl,--stack, <stacksize in bytes>

setenv KMP_STACKSIZE 10m

setenv GOMP_STACKSIZE 10000

setenv OMP_STACKSIZE 10M

Internal Control Variables. wait-policy-var

Подсказка OpenMP-компилятору о желаемом поведении нитей во время ожидания.
Начальное значение: зависит от реализации.
Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive

IBM AIX
SPINLOOPTIME=100000
YIELDLOOPTIME=40000

Internal Control Variables. Приоритеты

клауза	вызов функции	переменная окружения	ICV
	omp_set_dynamic()	OMP_DYNAMIC	*dyn-var*
	omp_set_nested()	OMP_NESTED	*nest-var*
num_threads	omp_set_num_threads()	OMP_NUM_THREADS	*nthreads-var*
schedule	omp_set_schedule()	OMP_SCHEDULE	*run-sched-var*
schedule			*def-sched-var*
		OMP_STACKSIZE	*stacksize-var*
		OMP_WAIT_POLICY	*wait-policy-var*
		OMP_THREAD_LIMIT	*thread-limit-var*
	omp_set_max_active_ levels()	OMP_MAX_ACTIVE_ LEVELS	*max-active-levels-var*

Система поддержки выполнения OpenMP-программ

int omp_get_num_threads(void);
-возвращает количество нитей в текущей параллельной области

```
#include <omp.h>
void work(int i);
void test()
{
    int np;
    np = omp_get_num_threads(); /* np == 1*/
    #pragma omp parallel private (np)
    {
       np = omp_get_num_threads();
       #pragma omp for schedule(static)
       for (int i=0; i < np; i++)
           work(i);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_thread_num(void);
-возвращает номер нити в группе [0: omp_get_num_threads()-1]

```
#include <omp.h>
void work(int i);
void test()
{
    int iam;
    iam = omp_get_thread_num(); /* iam == 0*/
    #pragma omp parallel private (iam)
    {
       iam = omp_get_thread_num();
       work(iam);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_num_procs(void);
-возвращает количество процессоров, на которых программа выполняется

```
#include <omp.h>
void work(int i);
void test()
{
    int nproc;
    nproc = omp_get_num_ procs();
   #pragma omp parallel num_threads(nproc)
    {
       int iam = omp_get_thread_num();
       work(iam);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_level(void)
- возвращает уровень вложенности для текущей параллельной области.

```
#include <omp.h>
void work(int i) {
 #pragma omp parallel
   {
      int ilevel = omp_get_level ();
   }
}
void test()
{
    int ilevel = omp_get_level (); /*ilevel==0*/
   #pragma omp parallel private (ilevel)
   {
       ilevel =  omp_get_level ();
       int iam = omp_get_thread_num();
       work(iam);
    }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_active_level(void)

- возвращает количество активных параллельных областей (выполняемых 2-мя или более нитями).

```
#include <omp.h>
void work(int iam, int size) {
 #pragma omp parallel
    {
       int ilevel = omp_get_active_level ();
    }
}
void test()
{
    int size = 0;
    int ilevel = omp_get_active_level (); /*ilevel==0*/
    scanf("%d",&size);
    #pragma omp parallel if (size>10)
    {
       int iam = omp_get_thread_num();
        work(iam, size);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_ancestor_thread_num (int level)

- для нити, вызвавшей данную функцию, возвращается номер нити-родителя, которая создала указанную параллельную область.

omp_get_ancestor_thread_num (0) = 0

```
If (level==omp_get_level()) {
    omp_get_ancestor_thread_num (level) == omp_get_thread_num ();
}

If ((level<0)||(level>omp_get_level())) {
    omp_get_ancestor_thread_num (level) == -1;
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_team_size(int level);
- количество нитей в указанной параллельной области.

omp_get_team_size (0) = 1

```
If (level==omp_get_level()) {
    omp_get_team_size (level) == omp_get_num _threads ();
}

If ((level<0)||(level>omp_get_level())) {
    omp_get_team_size (level) == -1;
}
```

Система поддержки выполнения OpenMP-программ. Функции работы со временем

double omp_get_wtime(void);

возвращает для нити астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Если некоторый участок окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время выполнения программы.

```
double start;
double end;
start = omp_get_wtime();
/*... work to be timed ...*/
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

double omp_get_wtick(void);

- возвращает разрешение таймера в секундах (количество секунд между последовательными импульсами таймера).

Литература

- **OpenMP Application Program Interface Version 3.1, July 2011.** http://ww.openmp.org/mp-documents/OpenMP3.1.pdf
- **Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009.** http://parallel.ru/info/parallel/openmp/OpenMP.pdf
- **Э. Таненбаум, М. ван Стеен.** Распределенные системы. Принципы и парадигмы. – СПб. Питер, 2003
- **Воеводин В.В., Воеводин Вл.В.** Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
- **Презентация** ftp://ftp.keldysh.ru/K_student/MSU2012/Academy2012_OpenMP.ppt

Автор

Бахтин Владимир Александрович, кандидат физико-математических наук, заведующий сектором Института прикладной математики им. М.В. Келдыша РАН, ассистент кафедры системного программирования факультета вычислительной математики и кибернетики Московского университета им. М.В. Ломоносова
bakhtin@keldysh.ru