# 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

# 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 5-6 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

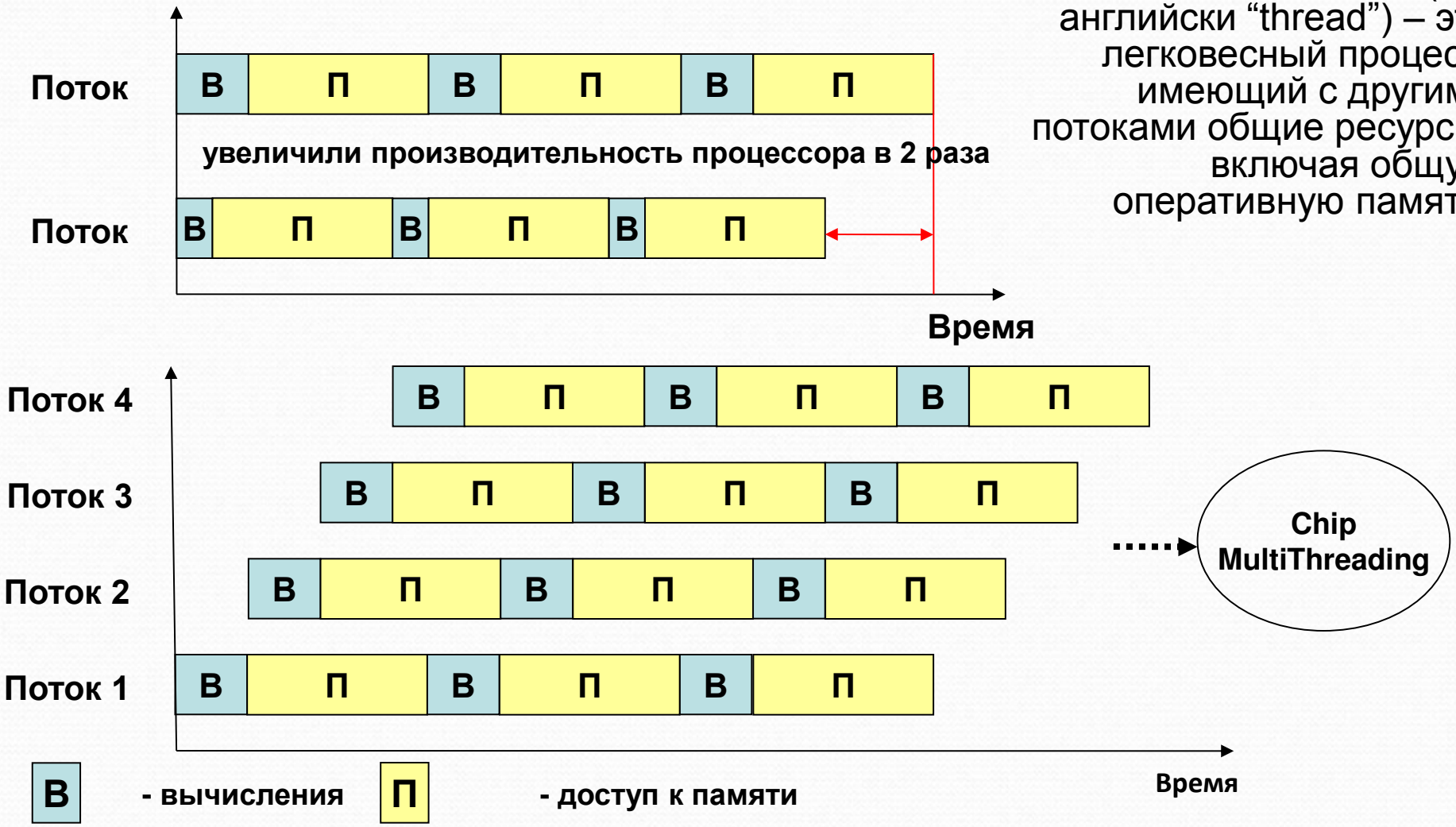
Latency to L3: 12 - 21 cycles

Latency to memory: 180 – 225 cycles

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

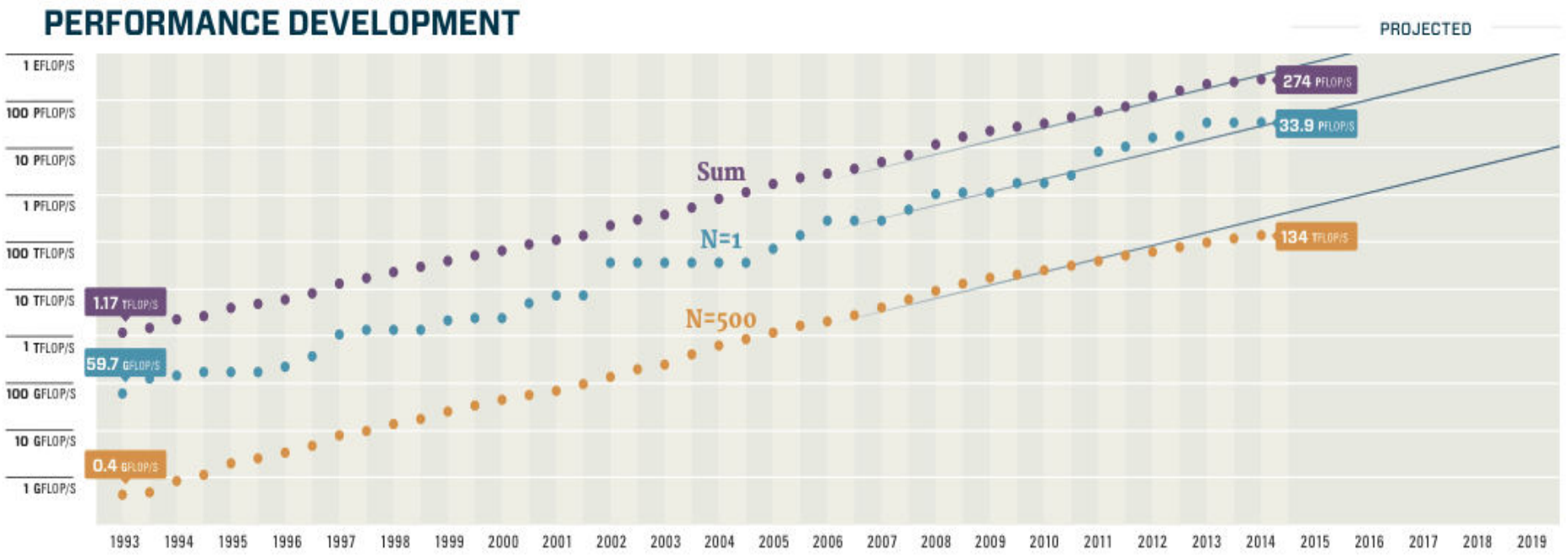
# Тенденции развития современных процессоров

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



# Суперкомпьютерные системы (Top500)

| | NAME | SPECS | SITE | COUNTRY | CORES | Rmax PFLOP/S | POWER MW |
|---|---|---|---|---|---|---|---|
| 1 | Tianhe-2 (Milkyway-2) | NUDT, Intel Ivy Bridge (12C, 2.2 GHz) & Xeon Phi (57C, 1.1 GHz), Custom interconnect | NSCC Guangzhou | China | 3,120,000 | 33.9 | 17.8 |
| 2 | Titan | Cray XK7, Operon 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect | DOE/SC/ORNL | USA | 560,640 | 17.6 | 8.2 |
| 3 | Sequoia | IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect | DOE/NNSA/LLNL | USA | 1,572,864 | 17.2 | 7.9 |
| 4 | K computer | Fujitsu SPARC64 VIIIfx (8C, 2.0GHz), Custom interconnect | RIKEN AICS | Japan | 705,024 | 10.5 | 12.7 |
| 5 | Mira | IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect | DOE/SC/ANL | USA | 786,432 | 8.59 | 3.95 |



# Суперкомпьютерные системы (Top500)

**№ 4 в Top 500**

**Суперкомпьютер K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect**

- Пиковая производительность - 11280 TFlop/s
- Число ядер в системе — 705 024
- Производительность на Linpack - 10510 TFlop/s (93.17 % от пиковой)
- Энергопотребление комплекса - **12659.89 кВт**

# Суперкомпьютерные системы (Top500)

**№ 3 в Top 500**

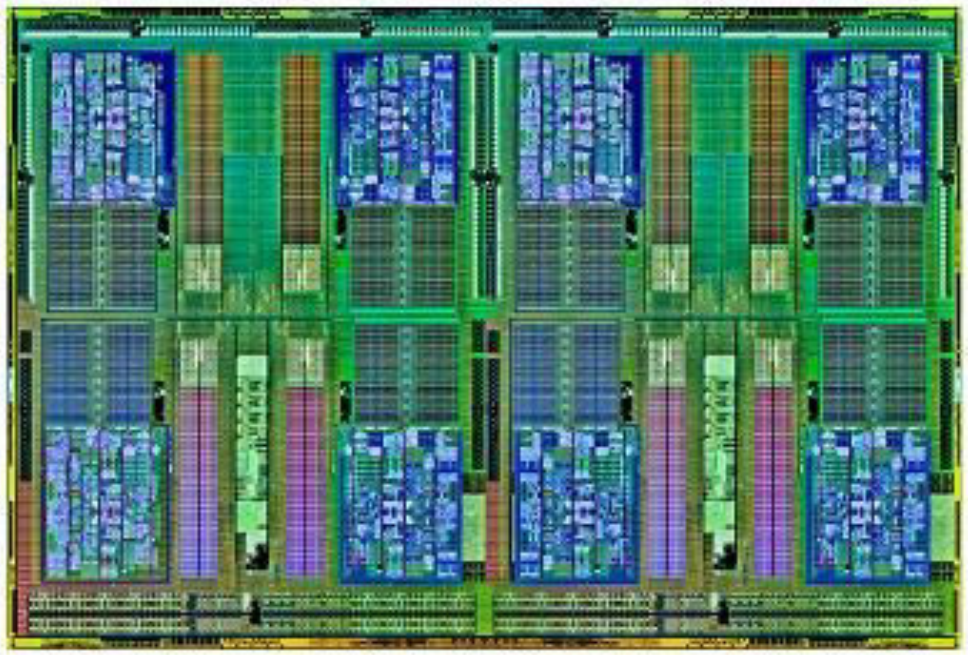
**Суперкомпьютер Sequoia, IBM BlueGene/Q, Power BQC 16C 1.6GHz, Custom interconnect**

- Пиковая производительность – 20142.66 TFlop/s
- Число ядер в системе — 1 572 864
- Производительность на Linpack – 16324.75 TFlop/s (81.08 % от пиковой)
- Энергопотребление комплекса - **7890.0 кВт**

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

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

# Тенденции развития современных процессоров



**AMD Opteron серии 6300**

6380 SE 16 ядер @ 2,5 ГГц, 16 МБ L3 Cache

6348 12 ядер @ 2,8 ГГц, 16 МБ L3 Cache

6328 8 ядер @ 3,2 ГГц, 16 МБ L3 Cache

6308 4 ядра @ 3,5 ГГц, 16 МБ L3 Cache

технология AMD Turbo CORE

встроенный контроллер памяти (4 канала памяти DDR3)

4 канала «точка-точка» с использованием HyperTransort 3.0

# Тенденции развития современных процессоров

**Intel Xeon Processor серии E5**

E5-2699 v3 (45M Cache, 2.30 GHz) 18 ядер, 36 нитей

E5-2698 v3 (40M Cache, 2.30 GHz) 16 ядер, 32 нити

E5-2697 v3 (35M Cache, 2.60 GHz) 14 ядер, 28 нитей

E5-2643 v3 (30M Cache, 3.40 GHz) 6 ядер, 12 нитей

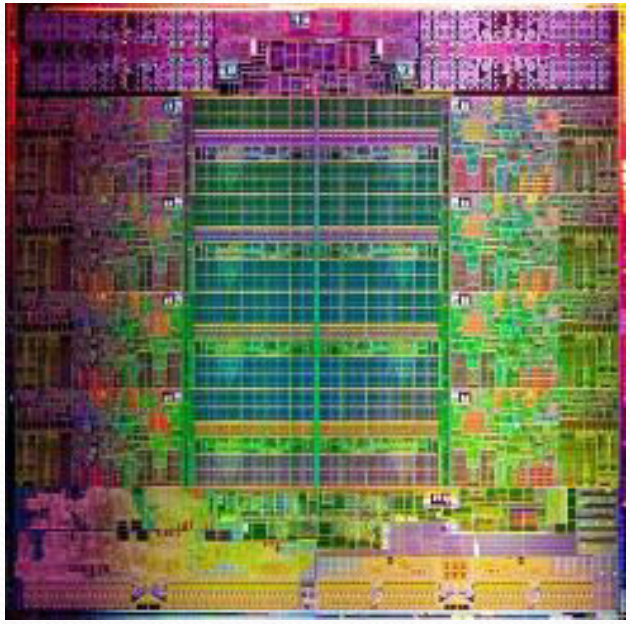
E5-2637 v3 (15M Cache, 3.50 GHz) 4 ядра, 8 нитей

Intel® Turbo Boost

Intel® Hyper-Threading

Intel® Intelligent Power

Intel® QuickPath



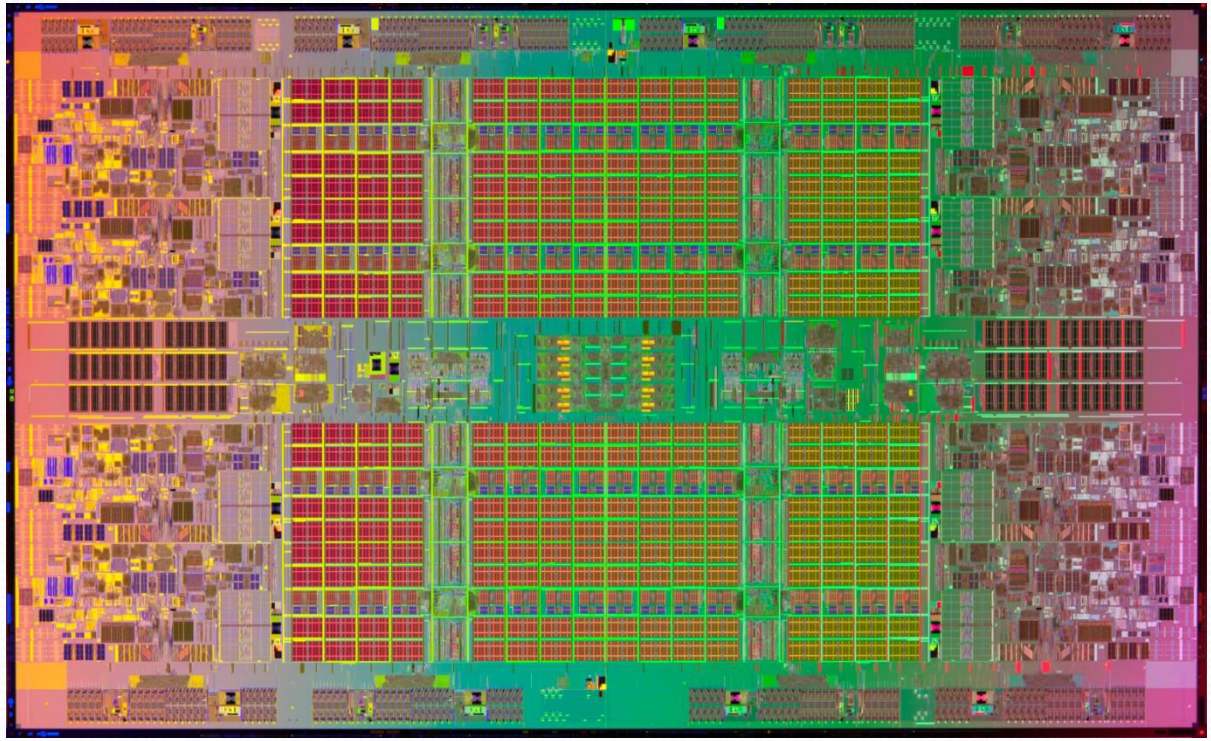
# Тенденции развития современных процессоров


**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

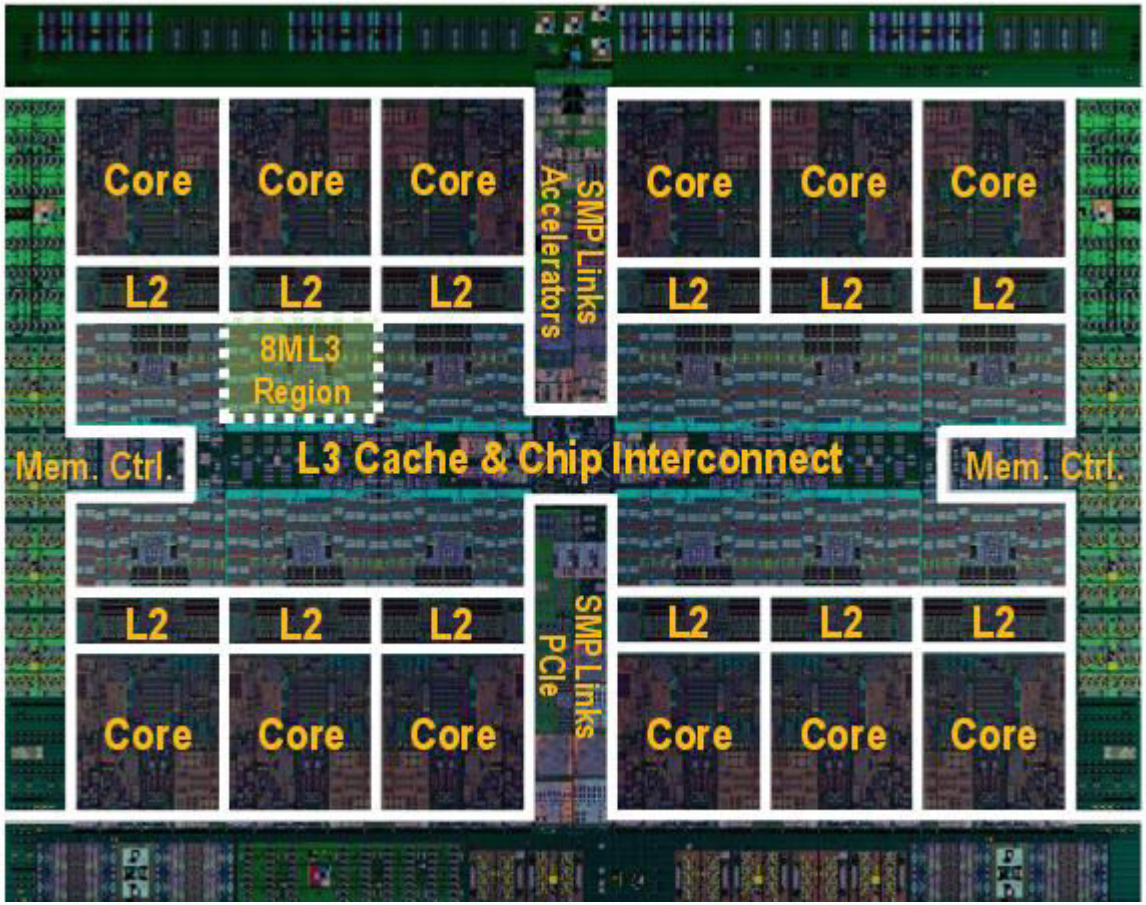


**Intel Itanium серии 9500**

9560 8 ядер @ 2,53 ГГц, 16 нитей, 32 МБ L3 Cache

9550 4 ядра @ 2,40 ГГц, 8 нитей, 32 МБ L3 Cache

# Тенденции развития современных процессоров



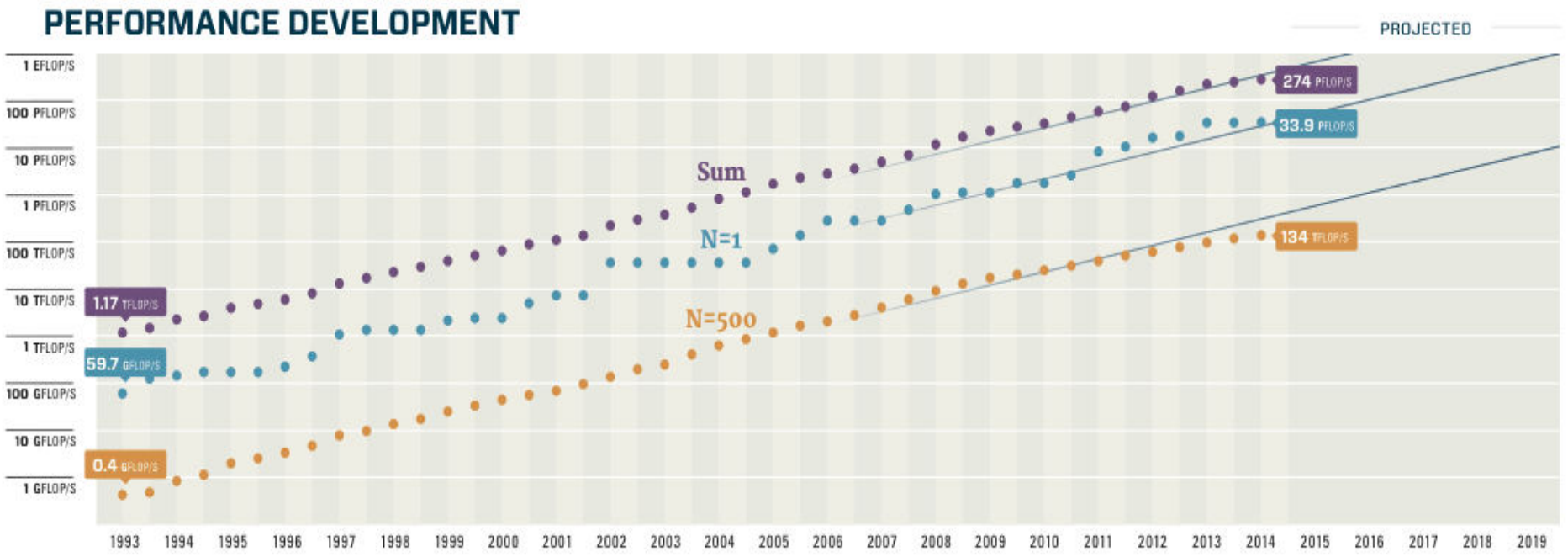

**IBM Power8**

- 2,75 – 4,2 ГГц
- 12 ядер x 8 нитей Simultaneuos MultiThreading
- 64 КБ Data Cache + 32КБ instruction Cache
- L2 512 КБ
- L3 96 МБ

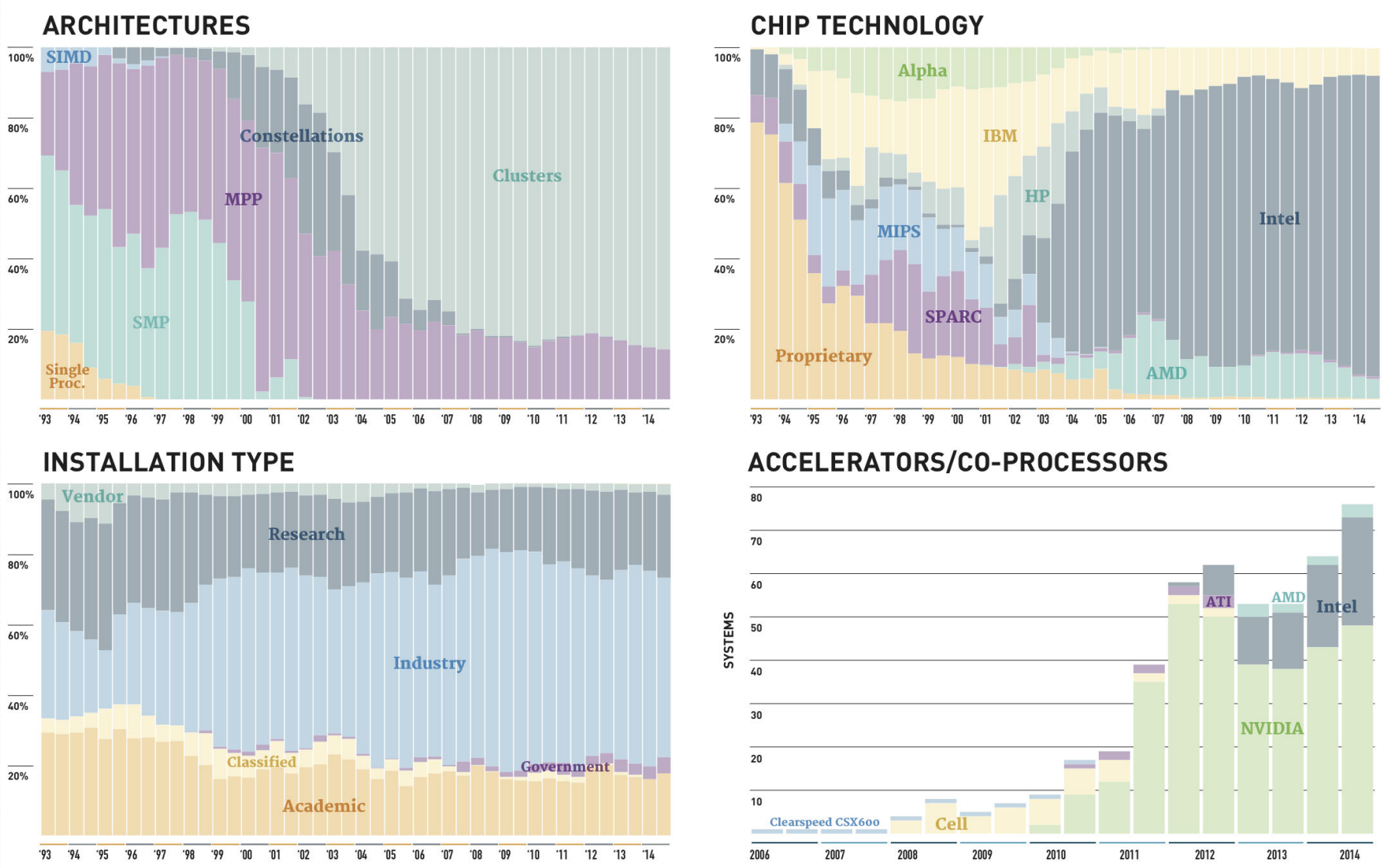
www.idh.ch/IBM_TU_2013/Power8.pdf

# Суперкомпьютерные системы (Top500)

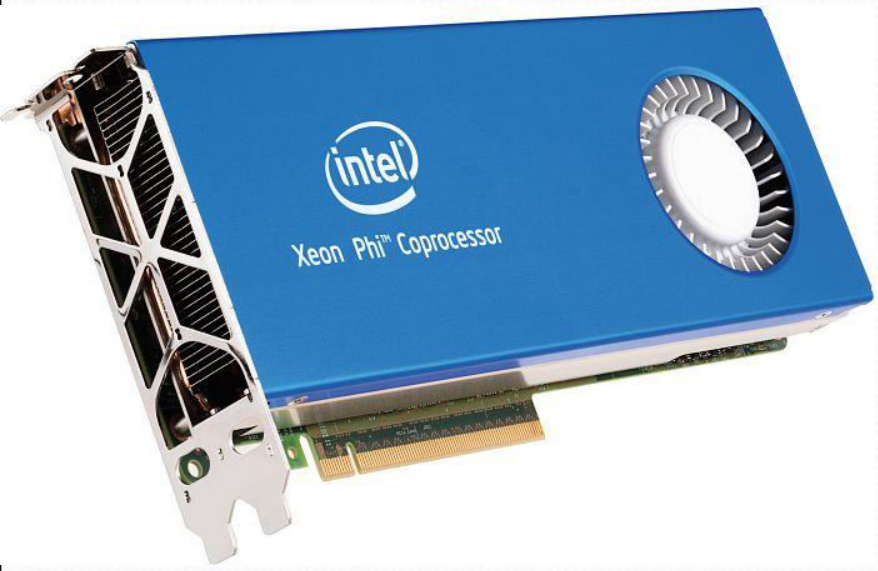
| | NAME | SPECS | SITE | COUNTRY | CORES | Rmax PFLOP/S | POWER MW |
|---|---|---|---|---|---|---|---|
| 1 | Tianhe-2 (Milkyway-2) | NUDT, Intel Ivy Bridge (12C, 2.2 GHz) & Xeon Phi (57C, 1.1 GHz), Custom interconnect | NSCC Guangzhou | China | 3,120,000 | 33.9 | 17.8 |
| 2 | Titan | Cray XK7, Operon 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect | DOE/SC/ORNL | USA | 560,640 | 17.6 | 8.2 |
| 3 | Sequoia | IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect | DOE/NNSA/LLNL | USA | 1,572,864 | 17.2 | 7.9 |
| 4 | K computer | Fujitsu SPARC64 VIIIfx (8C, 2.0GHz), Custom interconnect | RIKEN AICS | Japan | 705,024 | 10.5 | 12.7 |
| 5 | Mira | IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect | DOE/SC/ANL | USA | 786,432 | 8.59 | 3.95 |

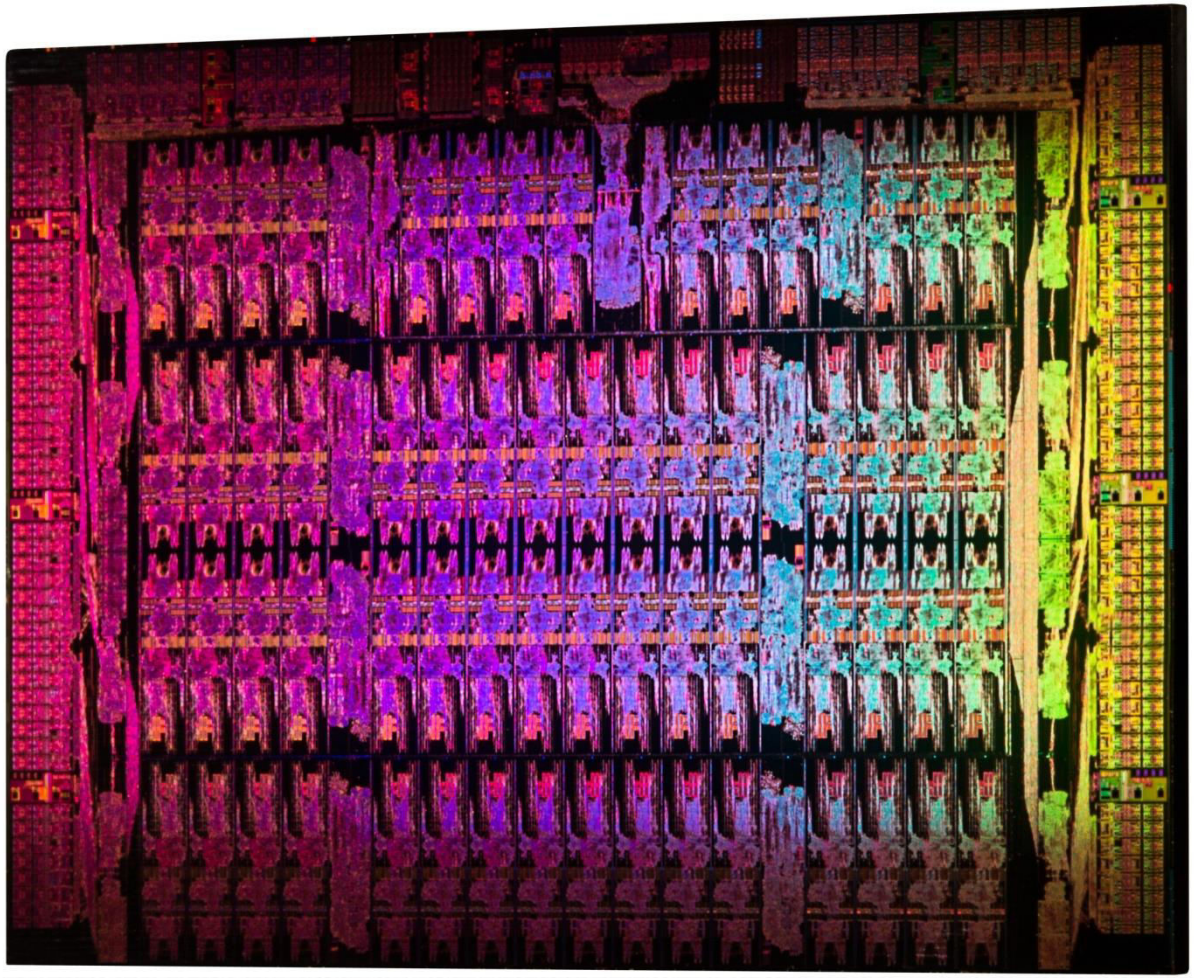


# Суперкомпьютерные системы (Top500)



# Intel Xeon Phi Coprocessor



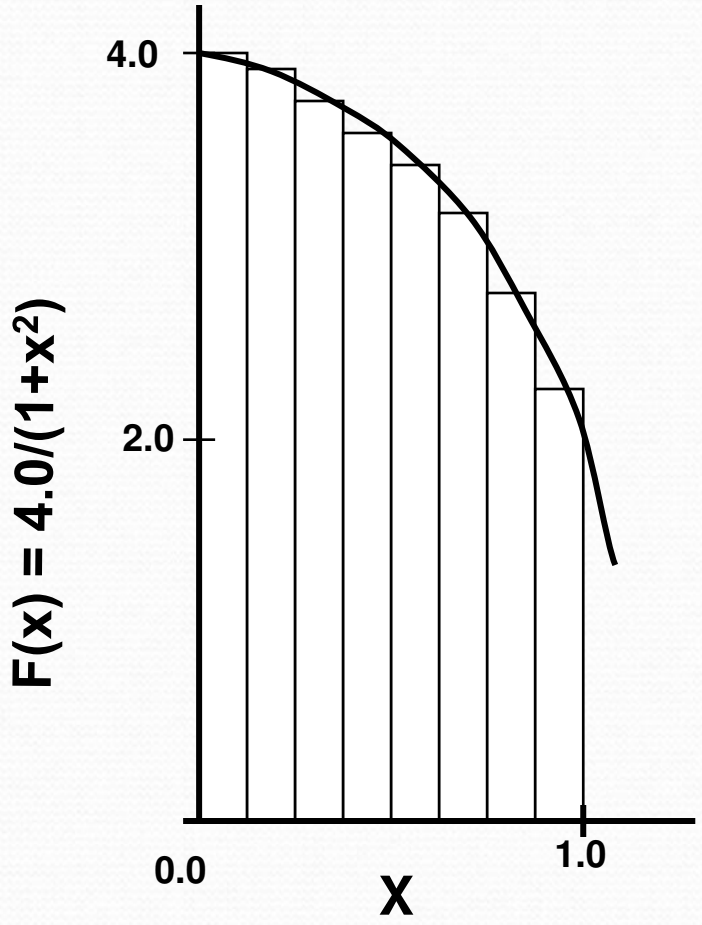
# Тенденции развития современных вычислительных систем

- 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

# Существующие подходы для создания параллельных программ

- 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 $\Delta x$ и высоту $F(x_i)$ в середине интервала

# Вычисление числа $\pi$. 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 $\pi$ 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 $\pi$ 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 $\pi$ 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);
    h   = 1.0 / (double) n;
    sum = 0.0;
```

# Вычисление числа $\pi$ 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;
}
```

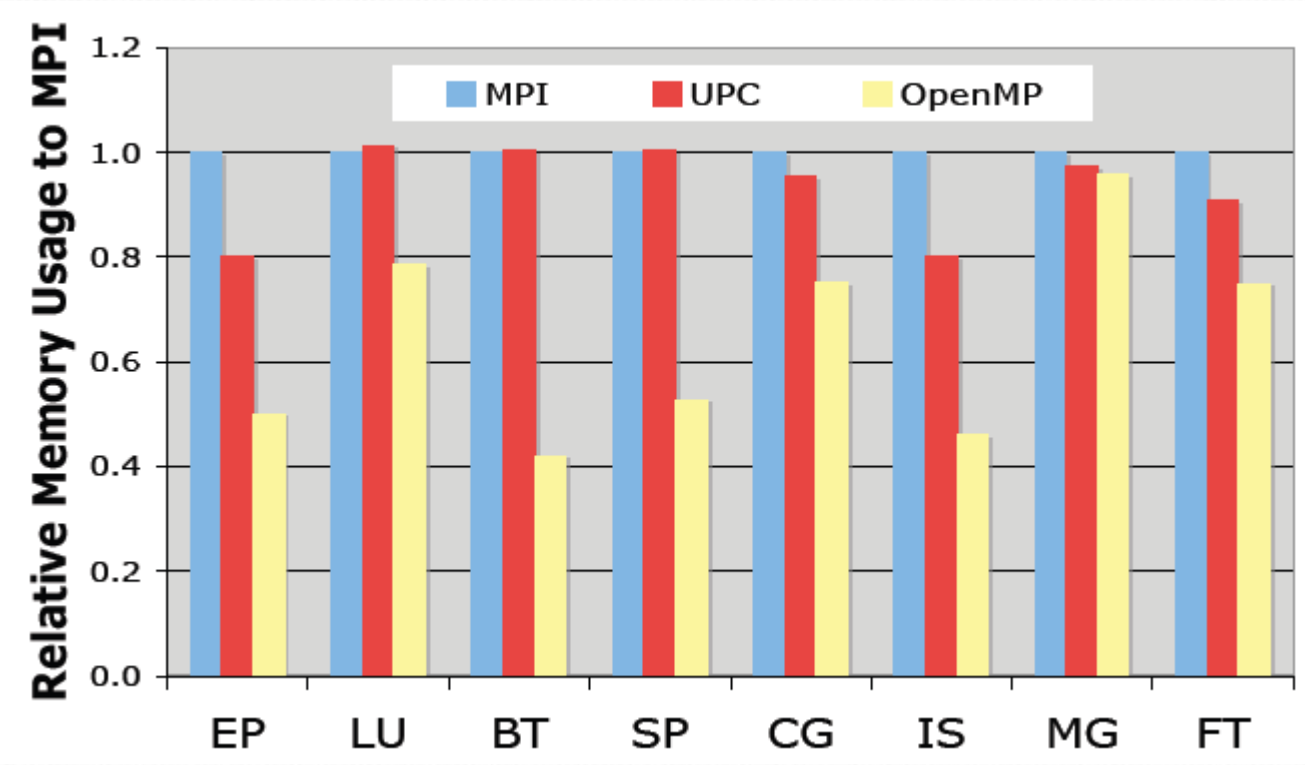
# Достоинства использования OpenMP вместо MPI для многоядерных процессоров

- 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 |

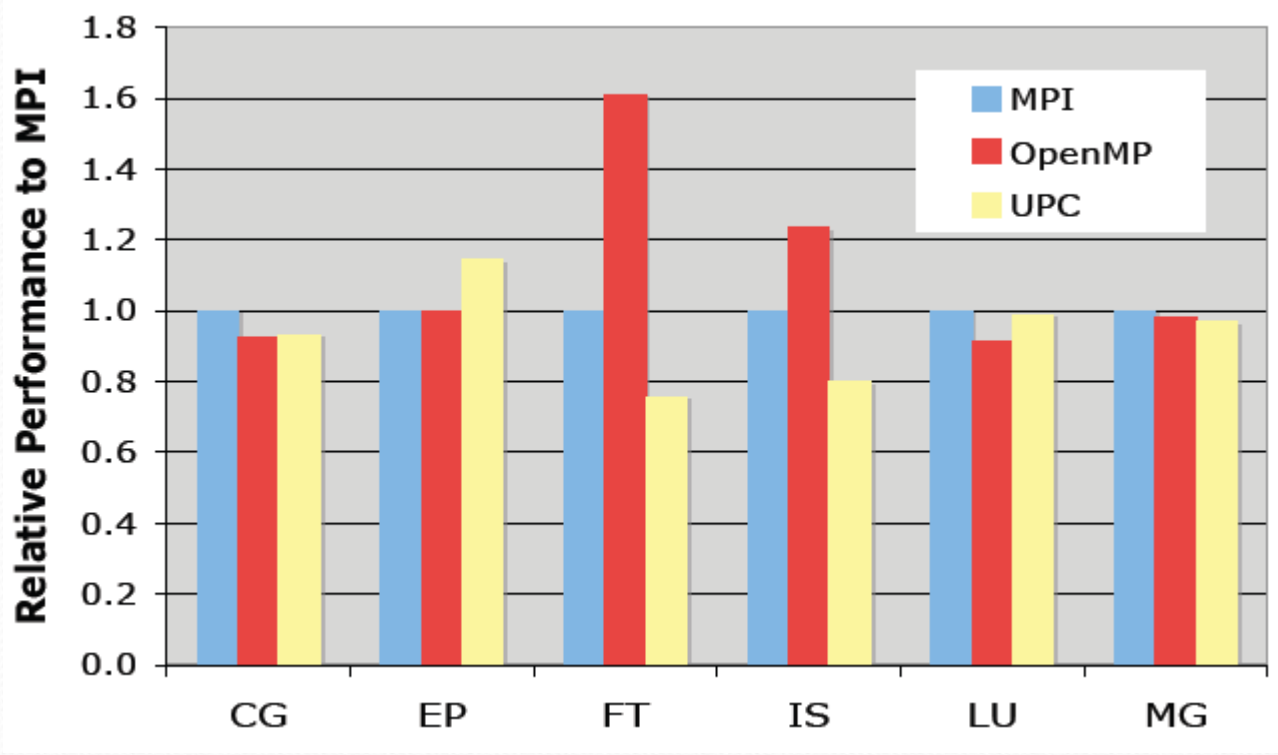
# Тесты NAS




**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



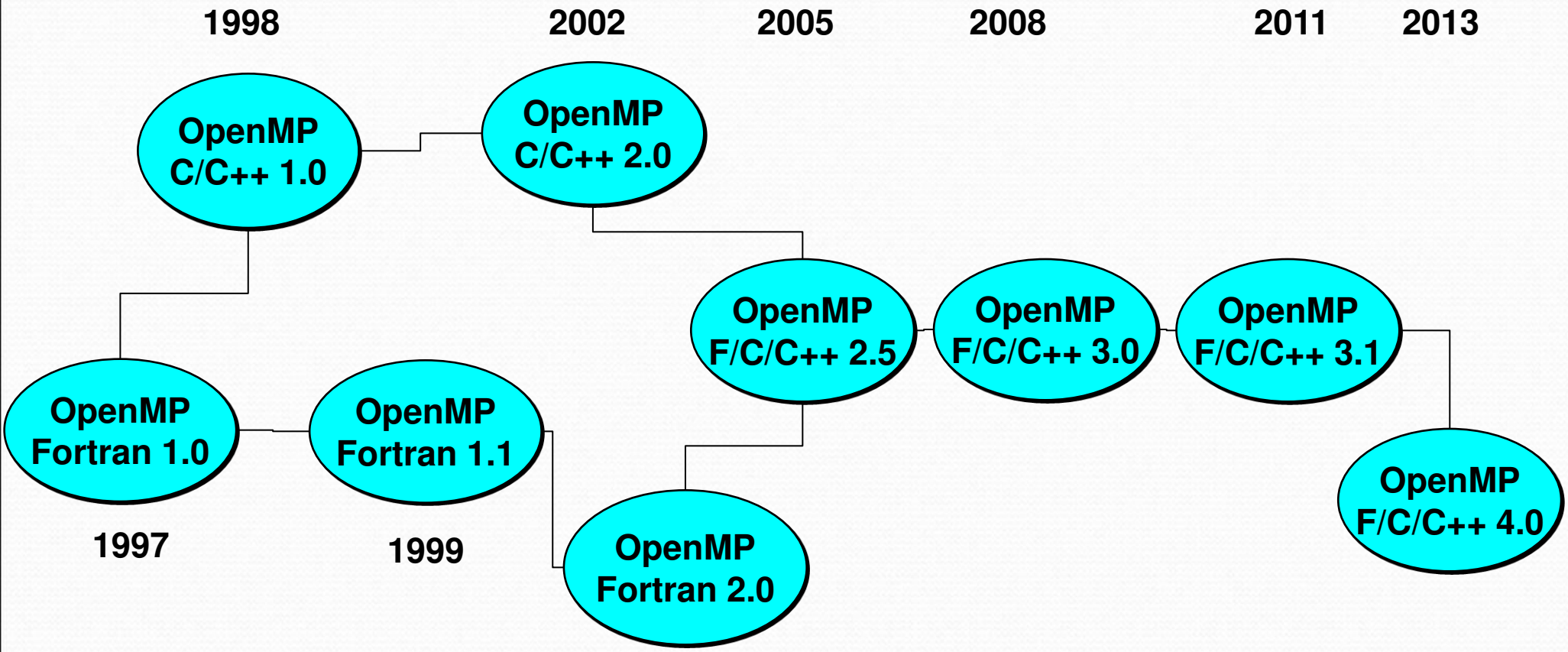

**Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms**

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

# Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- OpenMP 4.0

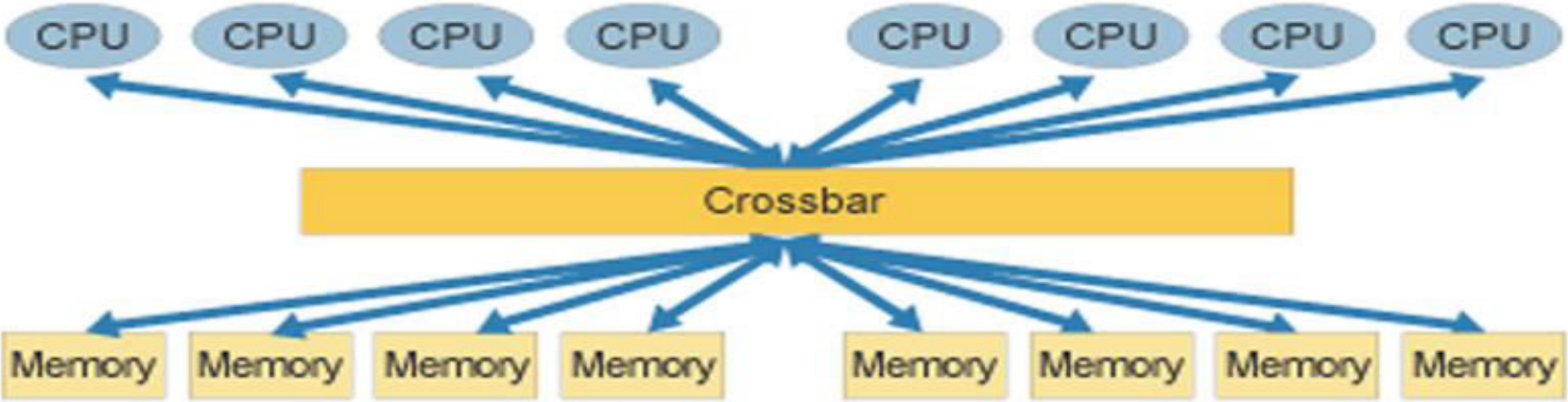
# История OpenMP



# 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
- Red Hat
- RWTH Aachen University
- Texas Advanced Computing Center
- SNL- Sandia National Lab
- BSC - Barcelona Supercomputing Center

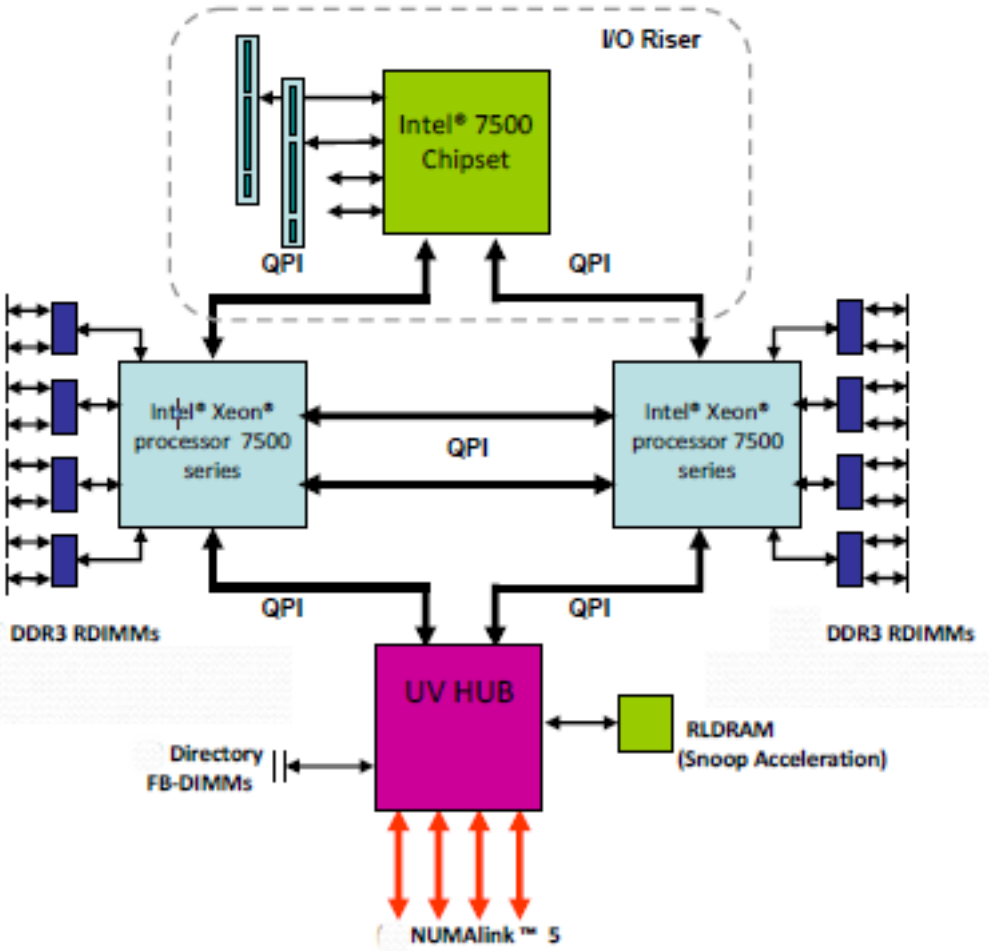
# Симметричные мультипроцессорные системы (SMP)




- 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.

**Например, серверы HP 9000 V-Class, Convex SPP-1200,…**

# 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) 2000**

- 256 Intel® Xeon® processor E5-4600 product family 2.4GHz-3.3GHz - 2048 Cores (4096 Threads)
- 64 TB памяти
- NUMAlink6 (NL6; 6.7GB/s bidirectional)

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

# 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 (+: A, B)
#pragma omp parallel for private(a, b)
C$OMP PARALLEL COPYIN(/blk/)
nthrds = OMP_GET_NUM_PROCS()
C$OMP BARRIER
C$OMP DO LASTPRIVATE(XX)
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***
- ***taskyield***
- ***tlush***
- ***taskgroup***

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**

# Составные директивы

```
#pragma omp parallel private(i)
{
#pragma omp for firstprivate(n)
for (i = 1; i <= n; i ++)
   {
      A[i]=A[i]+ B[i];
   }
}
```

```
#pragma omp parallel for private(i) \
  firstprivate(n)
for (i = 1; i <= n; i ++)
   {
      A[i]=A[i]+ B[i];
   }
```

# Компиляторы, поддеживающие OpenMP

**OpenMP 4.0:**

- GNU gcc (4.9.0)
- Intel 15.0: Linux, Windows and MacOS

**OpenMP 3.1:**

- Oracle Solaris Studio12.3: Linux and Solaris
- Cray: Cray XT series Linux environment
- LLVM: clang Linux and MacOS

**OpenMP 3.0:**

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

**Предыдущие версии OpenMP:**

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

# Компиляция OpenMP-программы

| Производитель | Компилятор | Опция компиляции |
|---|---|---|
| GNU | gcc | -fopenmp |
| LLVM | clang | -fopenmp |
| IBM | XL C/C++ / Fortran | -qsmp=omp |
| Oracle | C/C++ / Fortran | -xopenmp |
| Intel | C/C++ / Fortran | -openmp, -qopenmp /Qopenmp |
| Portland Group | C/C++ / Fortran | -mp |
| Microsoft | Visual Studio 2008 C++ | /openmp |

# Условная компиляция OpenMP-программы

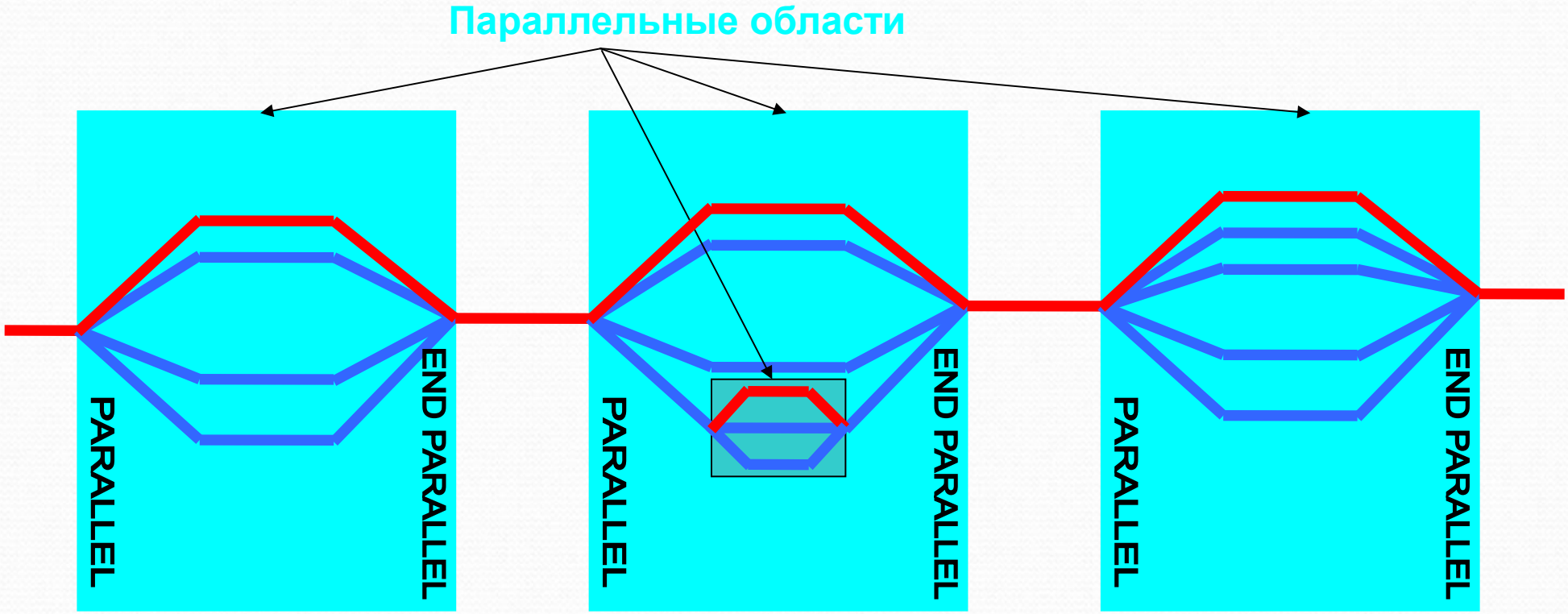
```
#include <stdio.h>
#include <omp.h> // Описаны прототипы всех функций и типов
int main()
{
#ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
    int id = omp_get_max_threads ();
#endif
    return 0;
}
```

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

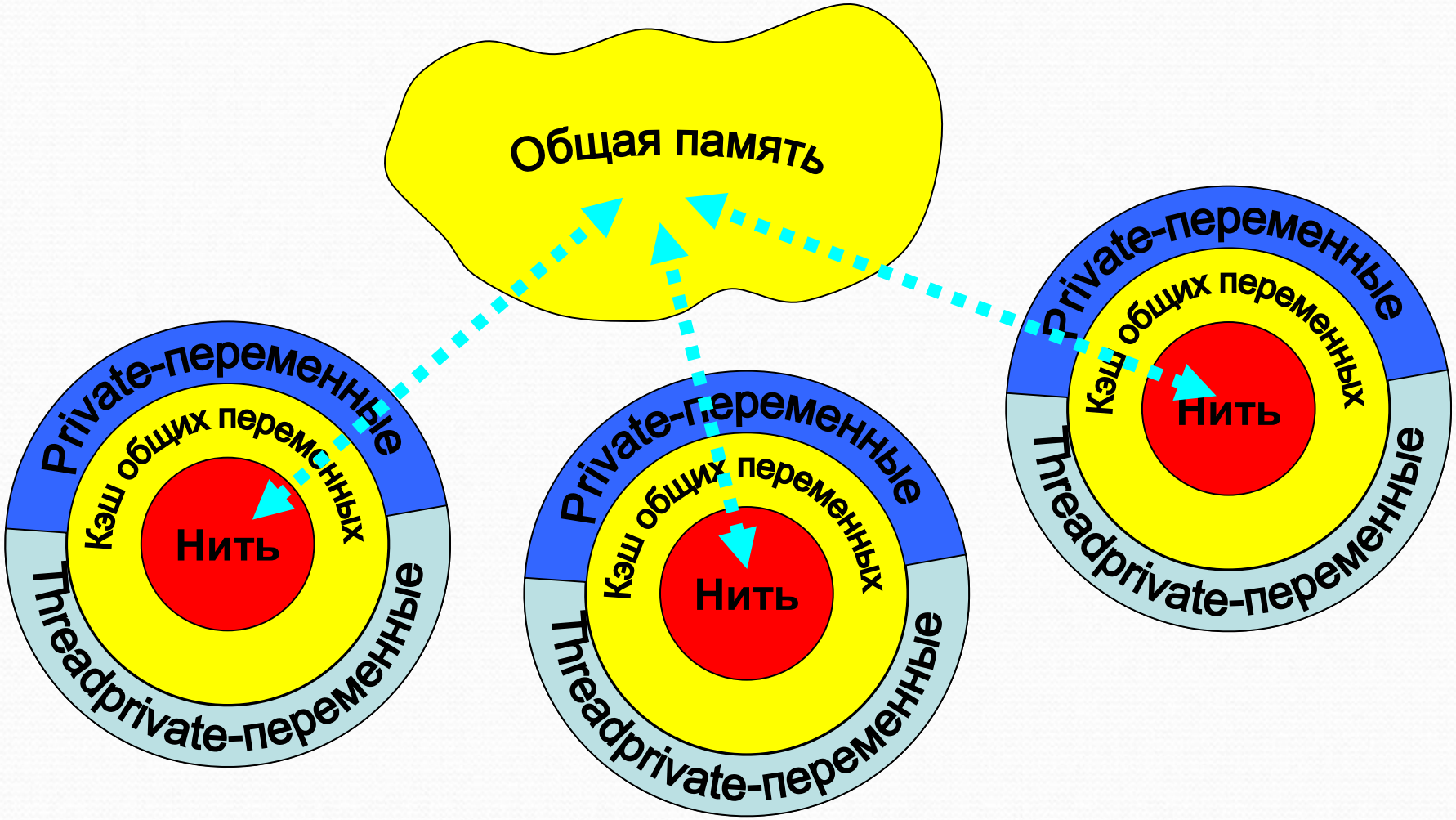
# Выполнение OpenMP-программы

Fork-Join параллелизм:

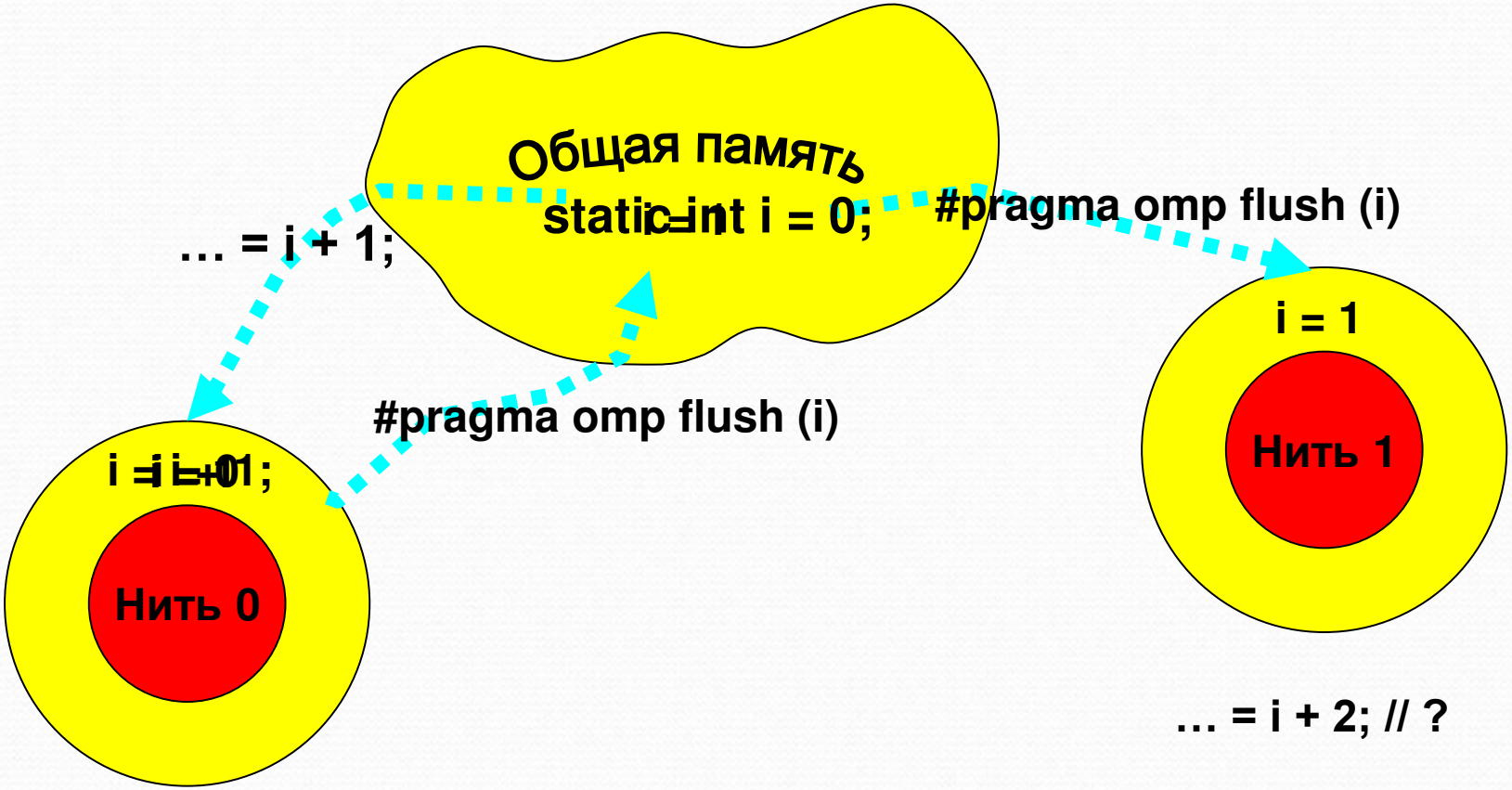
- Главная (master) нить порождает группу (team) нитей по мере небходимости.
- Параллелизм добавляется инкрементально.



# Модель памяти в OpenMP



# Модель памяти в OpenMP



# Консистентность памяти в OpenMP

Корректная последовательность работы нитей с переменной:

- Нить0 записывает значение переменной – write (var)
- Нить0 выполняет операцию синхронизации – flush (var)
- Нить1 выполняет операцию синхронизации – flush (var)
- Нить1 читает значение переменной – read (var)

1: A = 1

. . .

2: flush(A)

# Консистентность памяти в OpenMP

**#pragma omp flush** *[(список переменных)]*

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

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

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

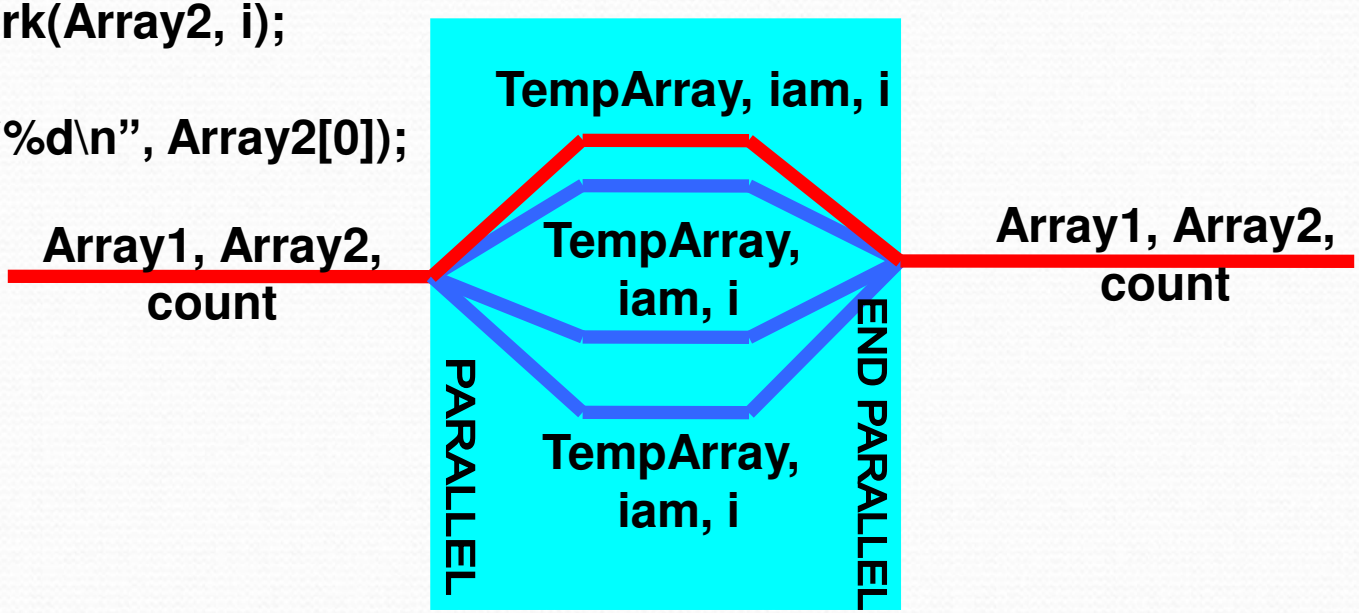
# Классы переменных

- 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

```
#define N 100
double Array1[N];
int main() {
    int Array2[N],i;
#pragma omp parallel
  {
      int  iam = omp_get_thread_num();
      #pragma omp for
      for (i=0;i < N; i++)
         work(Array2, i);
  }
  printf("%d\n", Array2[0]);
}
```

```
extern double Array1[N];
void work(int *Array, int i) {
    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

```
sum = -1.0;
#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;
}
//  sum == -1.0
```

# 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.

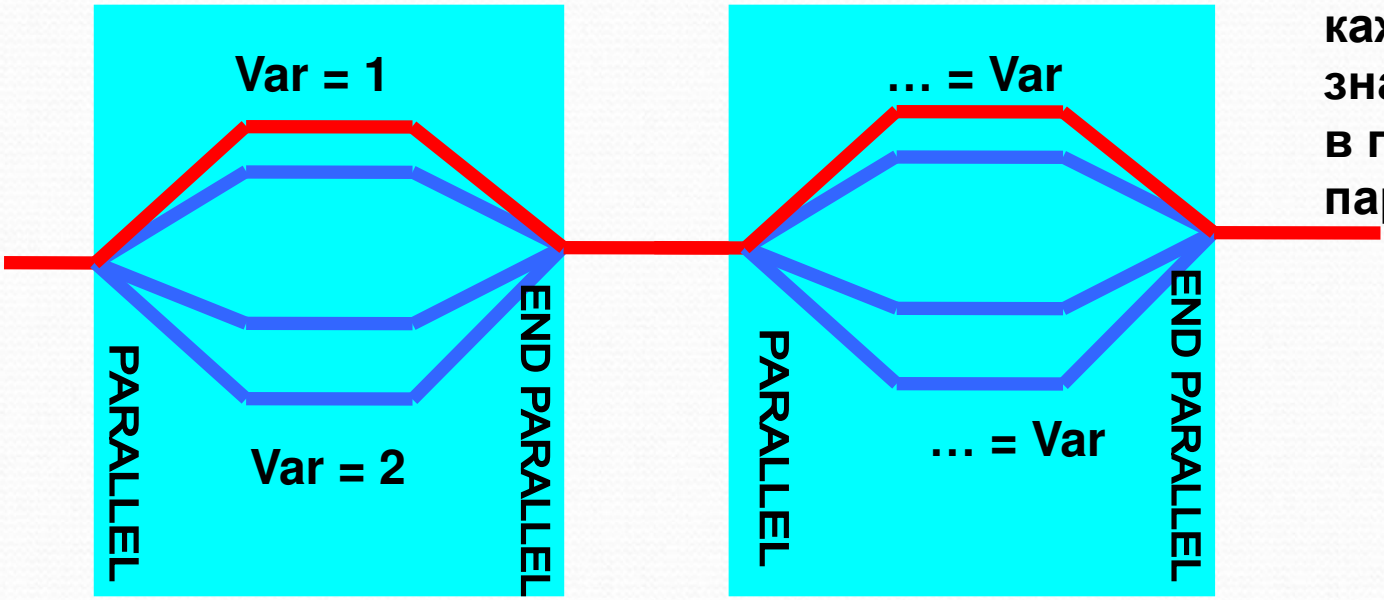
```
int i;
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];

}
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

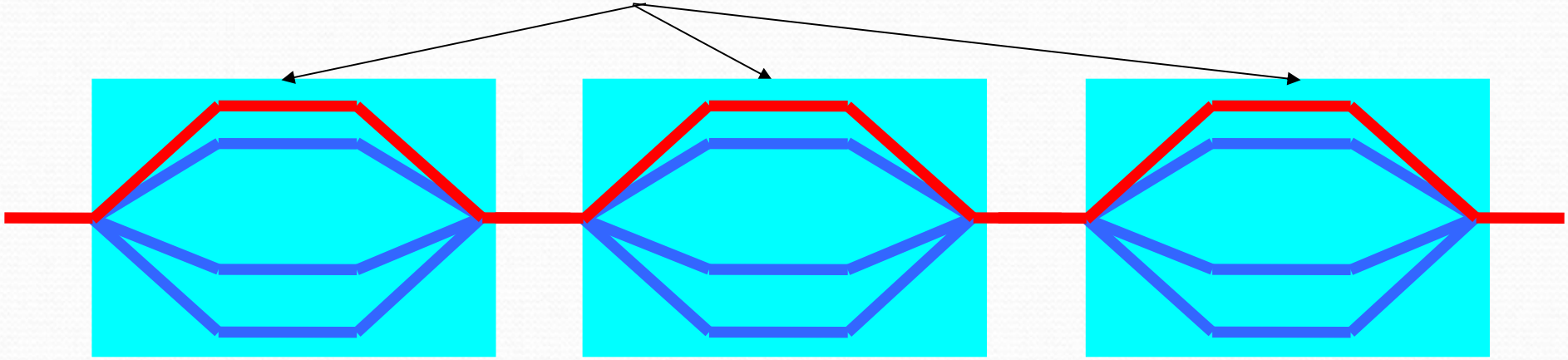
Меняет класс переменной по умолчанию:

- **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)

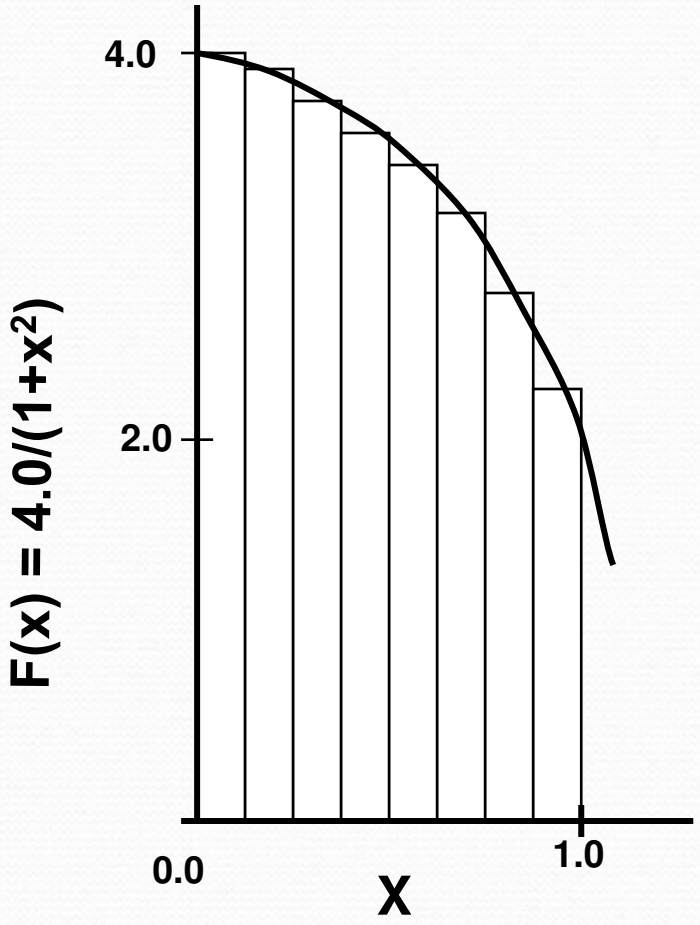


**#pragma omp parallel** *[ клауза[ [, ] клауза] ...]*
*структурный блок*

*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***)**
- **proc_bind ( master | close | spread )** **//OpenMP 4.0**

# 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 $\Delta x$ и высоту $F(x_i)$ в середине интервала

# Вычисление числа $\pi$. 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 $\pi$. Параллельная программа

```
#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;
       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);
             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;
}
```

# Вычисление числа $\pi$ 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;
}
```

# Редукционные операции

**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 |

# Использование редукционных операций

```
void reduction (float *x, int *y, int n)
{
  int i, b, c;
  float a, d;
  a = 0.0;
  b = 0;
  c = y[0];
  d = x[0];
  #pragma omp parallel for private(i) shared(x, y, n) \
                       reduction(+:a) reduction(^:b) \
                       reduction(min:c) reduction(max:d)
    for (i=0; i<n; i++) {
      a += x[i];
      b ^= y[i];
      if (c > y[i]) c = y[i];
      d = fmaxf(d,x[i]);
    }
}
```

# Реализация редукционных операций

```
#include <limits.h>
void reduction_by_hand (float *x, int *y, int n)
{
  int i, b, b_p, c, c_p;
  float a, a_p, d, d_p;
  a = 0.0f;
  b = 0;
  c = y[0];
  d = x[0];
  #pragma omp parallel shared(a, b, c, d, x, y, n)  private(a_p, b_p, c_p, d_p)
  {
    a_p = 0.0f;   b_p = 0;   c_p = INT_MAX;    d_p = -HUGE_VALF;
    #pragma omp for private(i)
    for (i=0; i<n; i++) {
      a_p += x[i];   b_p ^= y[i];   if (c_p > y[i]) c_p = y[i];   d_p = fmaxf(d_p,x[i]);
    }
    #pragma omp critical
    {
      a += a_p;   b ^= b_p;   if( c > c_p ) c = c_p;   d = fmaxf(d,d_p);
    }
  }
}
```

# Редукционные операции, определяемые пользователем (OpenMP 4.0)

**#pragma omp declare reduction (reduction-identifier : typename-list : combiner) [identity(identity-expr)]**

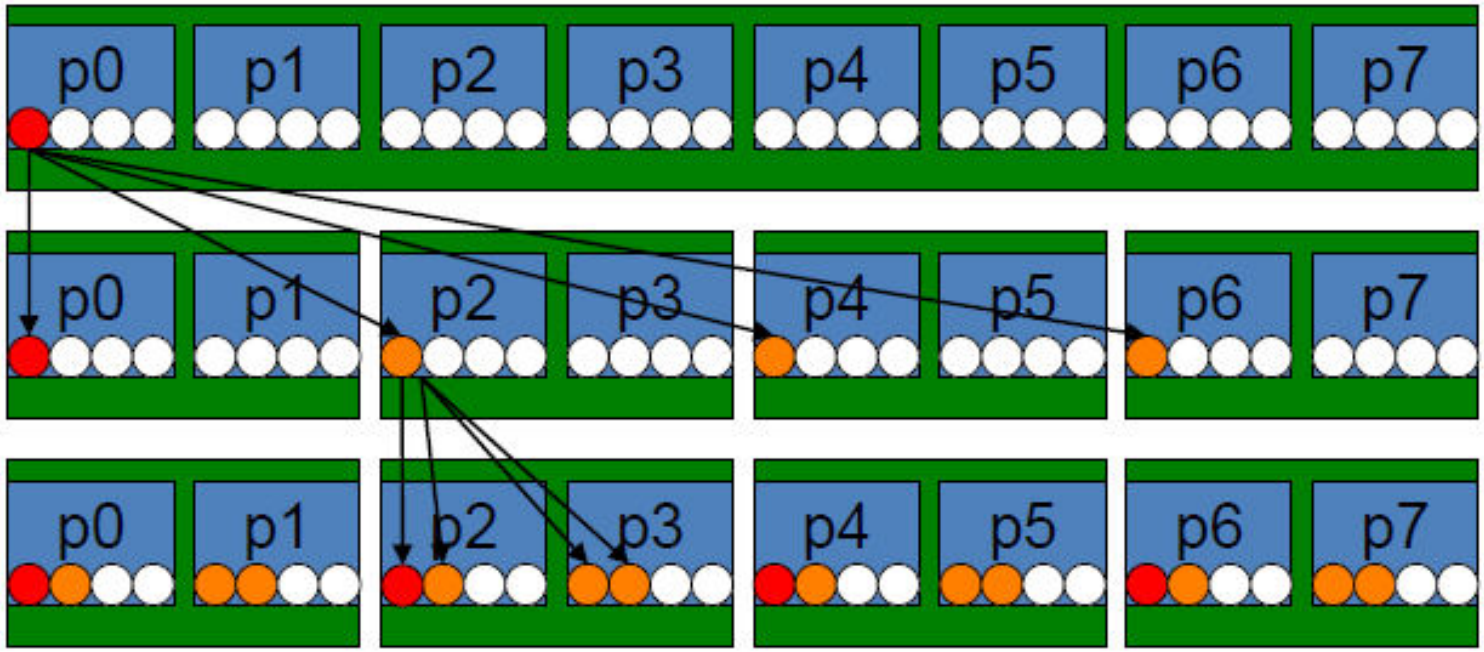
- **reduction-identifier - название редукционной операции**
- **typename-list – тип (типы)**
- **combiner – выражение для объединения частичных результатов**
- **identity – начальное значение создаваемых приватных переменных**

# Использование редукционных операций, определяемых пользователем (OpenMP 4.0)

```
struct point
{
   int x;
   int y;
} points[N], minp = { MAX_INT, MAX_INT };

#pragma omp declare reduction (min : struct point : \
    omp_out.x = omp_in.x > omp_out.x ? omp_out.x : omp_in.x, \
    omp_out.y = omp_in.y > omp_out.y ? omp_out.y : omp_in.y ) \
    initializer ( omp_priv = { MAX_INT, MAX_INT })

#pragma omp parallel for reduction (min: minp)
for (int i = 0; i < N; i++)
{
  if (points[i].x < minp.x) minp.x = points[i].x;
  if (points[i].y < minp.y) minp.y = points[i].y;
}
```

# Клауза proc_bind (OpenMP 4.0)

```
#pragma omp parallel proc_bind(spread) num_threads(4)
 {
   #pragma omp parallel proc_bind(close) num_threads(2)
        work();
 }
```

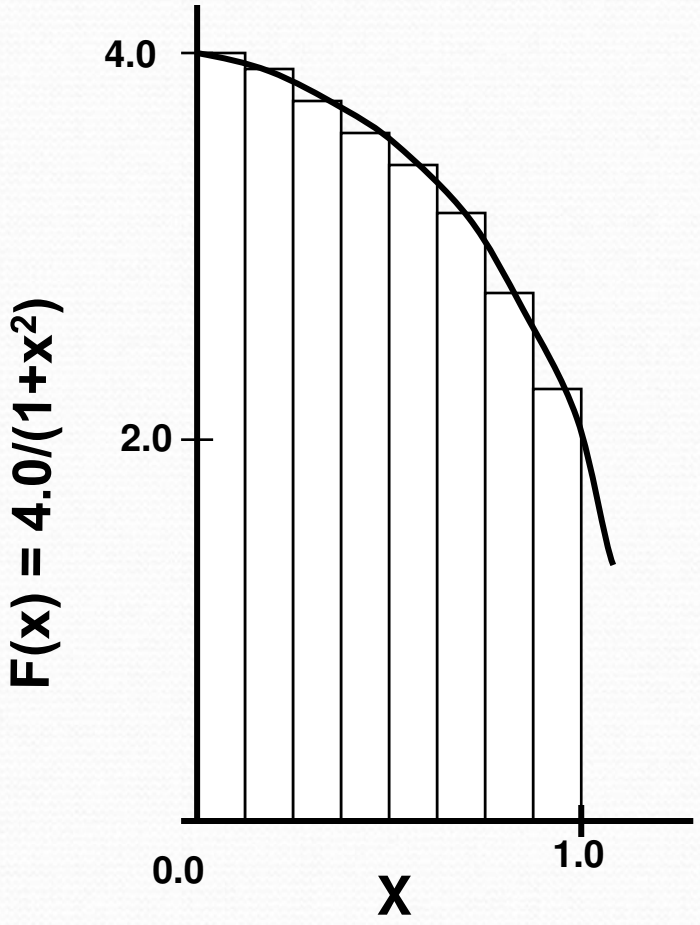


# Содержание

# Конструкции распределения работы

- Распределение витков циклов (директива for)
- Циклы с зависимостью по данным. Организация конвейерного выполнения для циклов с зависимостью по данным.
- Распределение нескольких структурных блоков между нитями (директива SECTION).
- Выполнение структурного блока одной нитью (директива single)
- Распределение операторов одного структурного блока между нитями (директива WORKSHARE)
- Понятие задачи

# 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 $\Delta x$ и высоту $F(x_i)$ в середине интервала

# Вычисление числа $\pi$ 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) 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;
}
```

# Вычисление числа $\pi$ 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) reduction(+:sum)
  {
    #pragma omp for schedule (static,1)
    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 $\pi$ 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 $\pi$ 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*

# 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

# 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.

# 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 //
  }
}
```

# Редукционные операции, определяемые пользователем (OpenMP 4.0)

```
#pragma omp declare reduction (merge : std::vector<int> :
omp_out.insert (omp_out.end(), omp_in.begin(), omp_in.end()))

void schedule (std::vector<int> &v, std::vector<int> &filtered)
{
    #pragma omp parallel for reduction (merge: filtered)
    for (std:vector<int>::iterator it = v.begin(); it < v.end(); it++)
        if ( filter(*it) ) filtered.push_back(*it);
}
```

**omp_out** refers to private copy that holds combined value
**omp_in** refers to the other private copy

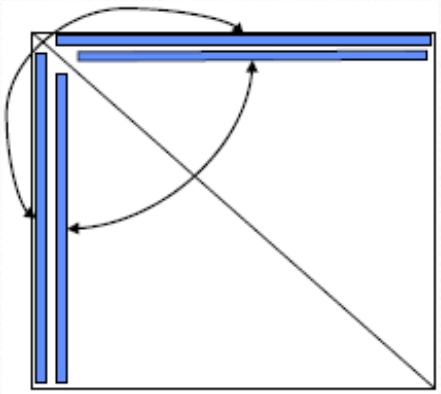
# Распределение витков цикла. Клауза schedule

**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).**

```
#pragma omp parallel for private(tmp) shared (a) schedule (runtime)
    for (int i=0; i<N-2; i++)
     for (int j = i+1; j< N-1; j++) {
       tmp = a[i][j];
       a[i][j]=a[j][i];
       a[j][i]=tmp;
     }
```



# Распределение витков цикла. Клауза schedule

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

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

- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-50.
- Поток 2 получает право на выполнение итераций 51-75.
- Поток 3 получает право на выполнение итераций 76-100.

# 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]);
   }
}
```

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

# Распределение циклов с зависимостью по данным

```
for(int i = 1; i < 100; i++)
    a[i]= a[i-1] + a[i+1];
```

Между витками цикла с индексами i1 и i2 ( i1<i2 ) существует зависимость по данным (информационная связь) массива *a*, если оба эти витка осуществляют обращение к одному элементу массива по схеме запись-чтение или чтение-запись.

Если виток i1 записывает значение, а виток i2 читает это значение, то между этими витками существует потоковая зависимость или просто зависимость i1 -> i2.

Если виток i1 читает "старое" значение, а виток i2 записывает "новое" значение, то между этими витками существует обратная зависимость i1<- i2.

В обоих случаях виток i2 может выполняться только после витка i1.

# Распределение циклов с зависимостью по данным

```
for (int i = 0; i < n; i++) {
   x = (b[i] + c[i]) / 2;
   a[i] = a[i + 1] + x;       // ANTI dependency
}
```

```
#pragma omp parallel shared(a, a_copy) private (x)
{
   #pragma omp for
   for (int i = 0; i < n; i++) {
      a_copy[i] = a[i + 1];
   }
   #pragma omp for
   for (int i = 0; i < n; i++) {
      x = (b[i] + c[i]) / 2;
      a[i] = a_copy[i] + x;
   }
}
```

# Распределение циклов с зависимостью по данным

```
for (int i = 1; i < n; i++) {
   b[i] = b[i] + a[i - 1];
   a[i] = a[i] + c[i]; // FLOW dependency
}
```

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

# Клауза и директива ordered

```
void print_iteration(int iter) {
    #pragma omp ordered
        printf("iteration %d\n", iter);
}

int main( ) {
    int i;
    #pragma omp parallel
    {
        #pragma omp for ordered
          for (i = 0 ; i < 5 ; i++) {
              print_iteration(i);
              another_work (i);
          }
    }
}
```

**Результат выполнения программы:**

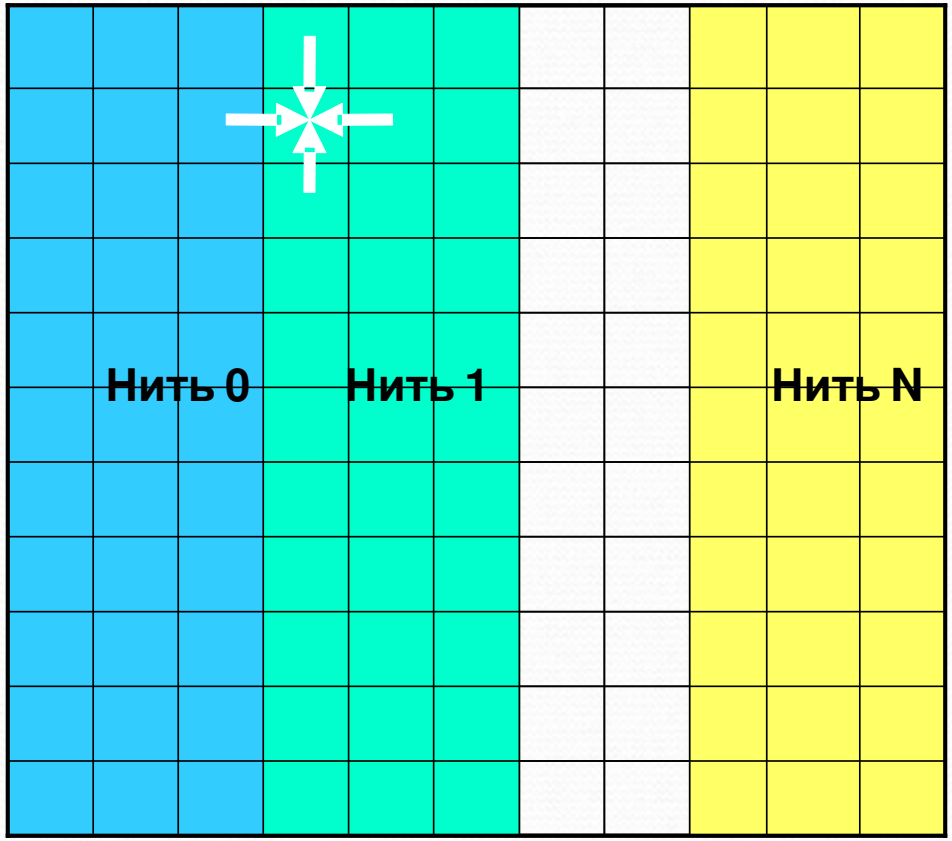
```
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
```

# Распределение циклов с зависимостью по данным. Клауза и директива ordered

```
#pragma omp parallel for ordered
for(int i = 1; i < 100; i++) {
    #pragma omp ordered
      {
        a[i]= a[i-1] + a[i+1];
      }
}
```

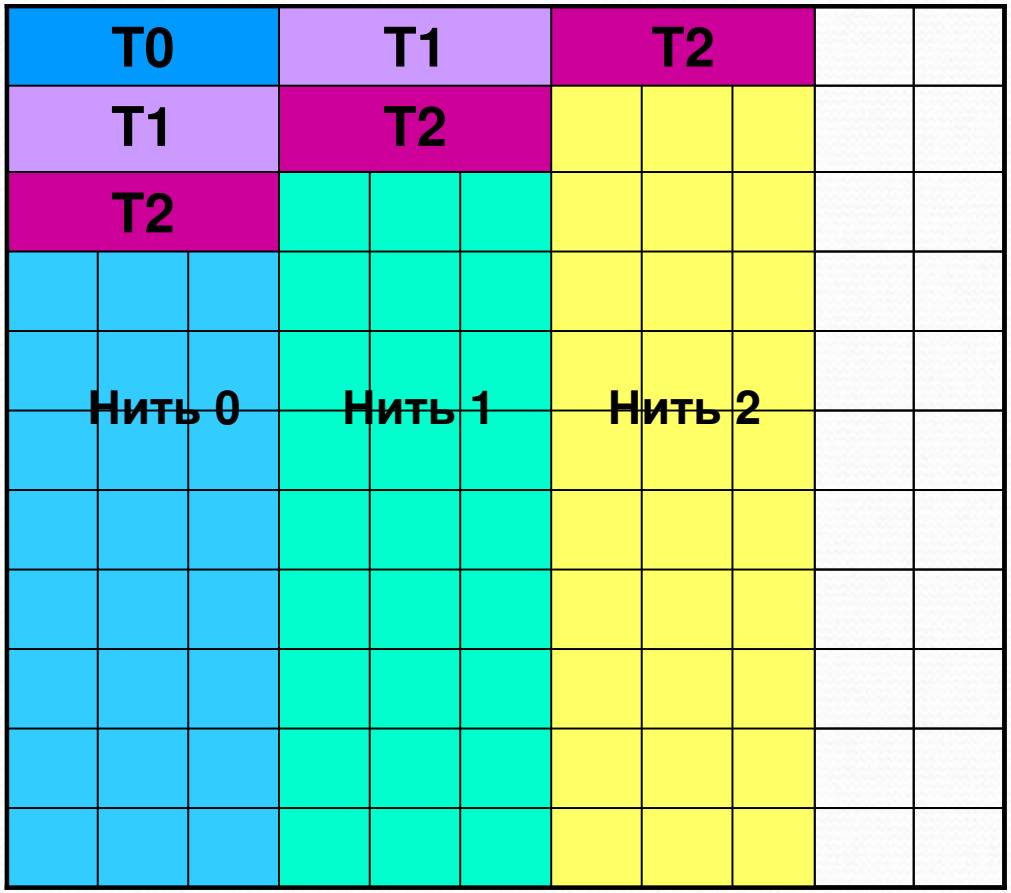
# Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```



# Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```



# Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
int iam, numt, limit;
int *isync = (int *)
malloc(omp_get_max_threads()*sizeof(int));
#pragma omp parallel private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,N-2);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<M; i++) {
    if ((iam>0) && (iam<=limit)) {
      for (;isync[iam-1]==0;) {
        #pragma omp flush (isync)
      }
      isync[iam-1]=0;
      #pragma omp flush (isync)
    }
#pragma omp for schedule(static) nowait
    for (int j=1; j<N; j++) {
      a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] +
               a[i][j+1])/4;
    }
    if (iam<limit) {
      for (;isync[iam]==1;) {
        #pragma omp flush (isync)
      }
      isync[iam]=1;
      #pragma omp flush (isync)
    }
  }
}
```

# Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
#pragma omp parallel
  {
    int iam = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    for (int newi=1; newi<M + numt - 1; newi++) {
      int i = newi - iam;
      #pragma omp for
        for (int j=1; j<N; j++) {
          if (i >= 1) && (i< N)) {
            a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1])/4;
          }
        }
    }
  }
```

# Распределение нескольких структурных блоков между нитями (директива sections)

```
#pragma omp sections [клауза[[,] клауза] ...]
{
  [#pragma omp section]
  структурный блок
  [#pragma omp section
  структурный блок ]
  ...
}
```

где клауза одна из :

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator: list)

nowait

```
void XAXIS();
void YAXIS();
void ZAXIS();
void example()
{
   #pragma omp parallel
   {
      #pragma omp sections
      {
         #pragma omp section
         XAXIS();
         #pragma omp section
         YAXIS();
         #pragma omp section
         ZAXIS();
      }
   }
}
```

# 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>
static 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 ();
   }
}
```

# Распределение операторов одного структурного блока между нитями (директива WORKSHARE)

```
      SUBROUTINE EXAMPLE (AA, BB, CC, DD, EE, FF, GG, HH, N)
       INTEGER N
       REAL AA(N,N), BB(N,N), CC(N,N)
       REAL DD(N,N), EE(N,N), FF(N,N)
       REAL GG(N,N), HH(N,N)
       REAL SHR
!$OMP PARALLEL SHARED(SHR)
!$OMP WORKSHARE
       AA = BB
       CC = DD
       WHERE (EE .ne. 0) FF = 1 / EE
       SHR = 1.0
       GG (1:50,1) = HH(11:60,1)
       HH(1:10,1) = SHR
!$OMP END WORKSHARE
!$OMP END PARALLEL
       END SUBROUTINE EXAMPLE
```

# Понятие задачи

**Задачи появились в OpenMP 3.0**
**Каждая задача:**

- **Представляет собой последовательность операторов, которые необходимо выполнить.**
- **Включает в себя данные, которые используются при выполнении этих операторов.**
- **Выполняется некоторой нитью.**

**В OpenMP 3.0 каждый оператор программы является частью одной из задач.**

- **При входе в параллельную область создаются неявные задачи (implicit task), по одной задаче для каждой нити.**
- **Создается группа нитей.**
- **Каждая нить из группы выполняет одну из задач.**
- **По завершении выполнения параллельной области, master-нить ожидает, пока не будут завершены все неявные задачи.**

# Понятие задачи. Директива task

**Явные задачи (explicit tasks) задаются при помощи директивы:**
**#pragma omp task [клауза[[,] клауза] ...]**
**структурный блок**
**где клауза одна из :**

- **if (scalar-expression)**
- **final(scalar-expression)** **//OpenMP 3.1**
- **untied**
- **mergeable** **//OpenMP 3.1**
- **shared (list)**
- **private (list)**
- **firstprivate (list)**
- **default (shared | none )**
- **depend (dependence-type: list)** **//OpenMP 4.0**

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

# Понятие задачи. Директива task

```
#pragma omp for schedule(dynamic)
    for (i=0; i<n; i++) {
        func(i);
    }
```

```
#pragma omp single
{
    for (i=0; i<n; i++) {
        #pragma omp task firstprivate(i)
            func(i);
    }
}
```

# Использование директивы task

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

# Использование директивы task. Клауза if

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

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

# Использование директивы task

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
  #pragma omp parallel shared (item)
    {
      #pragma omp single
        {
          for (int i=0; i<LARGE_NUMBER; i++)
            #pragma omp task
              process(item[i]);
        }
    }
}
```

Как правило, в компиляторах существуют ограничения на количество создаваемых задач. Выполнение цикла, в котором создаются задачи, будет приостановлено. Нить, выполнявшая этот цикл, будет использована для выполнения одной из задач

# Использование директивы task. Клауза untied

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
    #pragma omp parallel
      {
        #pragma omp single
          {
              #pragma omp task untied
             {
                for (int i=0; i<LARGE_NUMBER; i++)
                  #pragma omp task
                    process(item[i]);
             }
          }
      }
}
```

Клауза untied - выполнение задачи после приостановки может быть продолжено любой нитью группы

# Использование задач. Директива taskwait

```
#pragma omp taskwait

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

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

# Использование директивы task. Клауза final

```
void fib (int n, int d) {
    int x, y;
    if (n < 2) return 1;
#pragma omp task final (d > LIMIT) mergeable
        x =  fib (n - 1, d + 1);
#pragma omp task final (d > LIMIT) mergeable
        y =  fib (n - 2, d + 1);
#pragma omp taskwait
        return x + y;
}

int omp_in_final (void);
```

# Зависимости между задачами (OpenMP 4.0)

**Клауза depend(dependence-type : list)**
*где dependence-type:*
- **in**
- **out**
- **inout**

```
int i, y, a[100];

#pragma omp task depend(out : a)
{
    for (i=0;i<100; i++) a[i] = i + 1;
}

#pragma omp task depend(in : a[0:49]) depend(out : y)
{
    y = 0;
    for (i=0;i<50; i++) y += a[i];
}

#pragma omp task depend(in : y) {
    printf("%d\n", y);
}
```

# Зависимости между задачами (OpenMP 4.0)

```
void matmul_depend (int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
#pragma omp task private(ii, jj, kk) firstprivate(i, j, k) \
        depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
        depend ( inout: C[i:BS][j:BS] )
        for (ii = i; ii < i+BS; ii++ )
          for (jj = j; jj < j+BS; jj++ )
            for (kk = k; kk < k+BS; kk++ )
              C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

# Содержание

# Конструкции для синхронизации нитей

- Директива master
- Директива critical
- Директива atomic
- Семафоры
- Директива barrier
- Директива taskyield
- Директива taskwait
- Директива taskgroup  // OpenMP 4.0

# 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);
   }
}
```

# Вычисление числа $\pi$ 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 nowait
       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;
}
```

# Использование критической секции

```
int *next_from_queue(int type);
void work(int *val);

void critical_example()
{
  #pragma omp parallel
  {
    int *ix_next, *iy_next;
    #pragma omp critical (xaxis)
         ix_next = next_from_queue(0);
    work(ix_next);

    #pragma omp critical (yaxis)
        iy_next = next_from_queue(1);
    work(iy_next);
   }
}
```

# Директива atomic

```
#pragma omp atomic [ read | write | update | capture ] [seq_cst]
      expression-stmt

#pragma omp atomic capture
      structured-block
```

Если указана клауза read:

```
      v = x;
```

Если указана клауза write:

```
      x = expr;
```

Если указана клауза update или клаузы нет, то expression-stmt:

```
x binop= expr;
x = x binop expr;
x++;
++x;
x--;
--x;
```

x – скалярная переменная, expr – выражение, в котором не присутствует переменная x.

binop - не перегруженный оператор:

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

binop=:

++ , --

# Директива atomic

**Если указана клауза capture, то expression-stmt:**

```
v = x++;
v = x--;
v = ++x;
v = -- x;
v = x binop= expr;
```

**Если указана клауза capture, то structured-block:**

```
{ v = x; x binop= expr;}
{ v = x; x =  x binop expr;}
{ v = x; x++;}
{ v = x; ++x;}
{ v = x; x--;}
{ v = x; --x;}
{ x binop= expr; v = x;}
{ x =  x binop expr;  v = x;}
{ v = x; x binop= expr;}
{ x++; v = x;}
{ ++ x ; v = x;}
{ x--; v = x;}
{ --x; v = x;}
```

# Встроенные функции для атомарного доступа к памяти в GCC

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)
type __sync_add_and_fetch (type *ptr, type value, ...)
type __sync_sub_and_fetch (type *ptr, type value, ...)
type __sync_or_and_fetch (type *ptr, type value, ...)
type __sync_and_and_fetch (type *ptr, type value, ...)
type __sync_xor_and_fetch (type *ptr, type value, ...)
type __sync_nand_and_fetch (type *ptr, type value, ...)
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
```

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

# Вычисление числа $\pi$ 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;
}
```

# Использование директивы atomic

```
int atomic_read(const int *p)
{
  int value;
  /* Guarantee that the entire value of *p is read atomically. No part of
  * *p can change during the read operation.
  */
  #pragma omp atomic read
  value = *p;
  return value;
}

void atomic_write(int *p, int value)
{
  /* Guarantee that value is stored atomically into *p. No part of *p can change
  * until after the entire write operation is completed.
  */
  #pragma omp atomic write
  *p = value;
}
```

# Использование директивы atomic

```
int fetch_and_add(int *p)
{
   /* Atomically read the value of *p and then increment it. The previous value is
   * returned. */
   int old;
   #pragma omp atomic capture
   { old = *p; (*p)++; }
   return old;
}
```

seq_cst - sequentially consistent atomic construct, the operation to have the same meaning as a memory_order_seq_cst atomic operation in C++11/C11

```
#pragma omp atomic capture seq_cst  // OpenMP 4.0
{--x; v = x;} // capture final value of x in v and flush all variables
```

# Семафоры

**Концепцию семафоров описал Дейкстра (Dijkstra) в 1965**

**Семафор - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:**

**P - функция запроса семафора**
**P(s): [if (s == 0) <заблокировать текущий процесс>; else s = s-1;]**

**V - функция освобождения семафора**
**V(s): [if (s == 0) <разблокировать один из заблокированных процессов>; s = s+1;]**

# Семафоры в OpenMP

**Состояния семафора:**

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

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

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

# Вычисление числа $\pi$ с использованием семафоров

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

# Использование семафоров

```
#include <stdio.h>
#include <omp.h>
int main()
{
   omp_lock_t lck;
   int id;
   omp_init_lock(&lck);
   #pragma omp parallel shared(lck) private(id)
   {
      id = omp_get_thread_num();
      omp_set_lock(&lck);
      printf("My thread id is %d.\n", id); /* only one thread at a time can execute this printf */
      omp_unset_lock(&lck);
      while (! omp_test_lock(&lck)) {
            skip(id); /* we do not yet have the lock, so we must do something else*/
      }
      work(id); /* we now have the lock and can do the work */
      omp_unset_lock(&lck);
   }
   omp_destroy_lock(&lck);
   return 0;
}
```

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

# Использование семафоров

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

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

**Deadlock!**

# Использование семафоров

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

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

# 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 taskyield

```
#include <omp.h>
void something_useful ( void );
void something_critical ( void );
void foo ( omp_lock_t * lock, int n )
{
    int i;
    for ( i = 0; i < n; i++ )
        #pragma omp task
        {
            something_useful();
            while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

# Директива taskwait

```
#pragma omp taskwait

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

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

# Директива taskwait

```
#pragma omp task {}  // Task1
#pragma omp task // Task2
{
   #pragma omp task {} // Task3
}
#pragma omp task {} // Task4

#pragma omp taskwait
// Гарантируется что в данной точке завершатся Task1 && Task2  && Task4
```

# Директива taskgroup

```
#pragma omp task {}  // Task1
#pragma omp taskgroup
{
  #pragma omp task // Task2
  {
    #pragma omp task {} // Task3
  }
  #pragma omp task {} // Task4
}
// Гарантируется что в данной точке завершатся Task2 && Task3  && Task4
```

# Использование директивы taskgroup

```
struct tree_node
{
  struct tree_node *left, *right;
  float *data;
};
typedef struct tree_node* tree_type;
void compute_tree(tree_type tree)
{
  if (tree->left)
  {
    #pragma omp task
      compute_tree(tree->left);
  }
  if (tree->right)
  {
    #pragma omp task
      compute_tree(tree->right);
  }
  #pragma omp task
    compute_something(tree->data);
}
```

```
int main()
{
  tree_type tree;
  init_tree(tree);
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task
      start_background_work();
    #pragma omp taskgroup
    {
      #pragma omp task
        compute_tree(tree);
    }
    print_something ();
  } // only now background work is required
}   // to be complete
```

# Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- OpenMP 4.0

# Внутренние переменные, управляющие выполнением OpenMP-программы (ICV-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:**

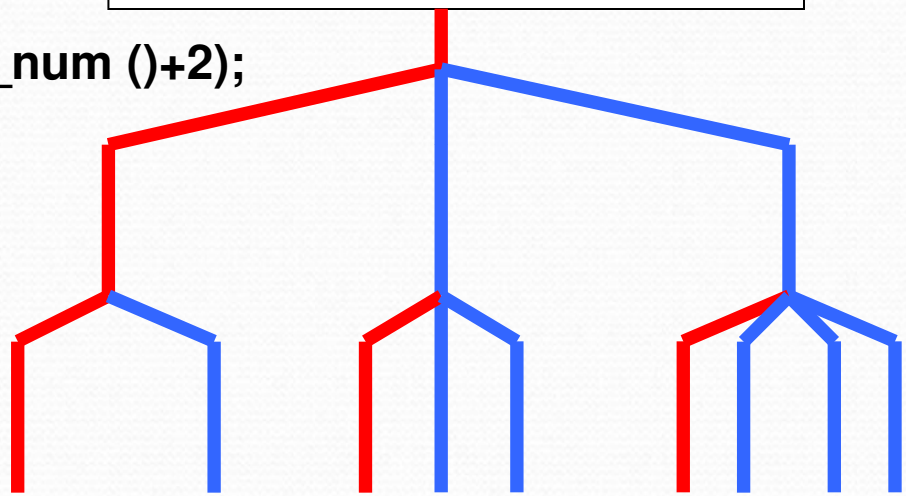
- stacksize-var
- wait-policy-var
- bind-var
- cancel-var
- place-partition-var

# 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();
   }
}
```

Не корректно в OpenMP 2.5

Корректно в OpenMP 3.0



Существует одна копия этой переменной для каждой задачи

# 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. 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  #  Size in Kilobytes**

# 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).

# Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- OpenMP 4.0

# Новые возможности OpenMP 4.0

- Векторизация кода
- Обработка исключительных ситуаций / cancellation constructs
- Привязка нитей к ядрам
- Поддержка ускорителей/сопроцессоров

# Использование векторизации

```
void add_float (float *restrict a, float *restrict b, float *restrict c,
float *restrict d, float *restrict e, int n) // C99
{
    for (int i=0; i<n; i++)
      a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

# Использование векторизации. Спецификация simd

**#pragma omp simd *[clause[[,] clause]..]***
***for-loops***

**#pragma omp for simd *[clause[[,] clause]..]***
***for-loops***

**где клауза одна из:**

- **safelen (length)**
- **linear (list[:linear-step])**
- **aligned (list[:alignment])**
- **private (list)**
- **lastprivate (list)**
- **reduction (reduction-identifier: list)**
- **collapse (n)**

# Использование векторизации

```
void add_float (float *a, float *b, float *c, float *d, float *e, int n)
{
  #pragma omp simd
  for (int i=0; i<n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

```
void add_float (float *restrict a, float *restrict b, float *restrict c,
float *restrict d, float *restrict e, int n) // C99
{
   for (int i=0; i<n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

# Использование векторизации. Спецификация declare simd

**#pragma omp declare simd *[clause[[,] clause]..]***
***function definition or declaration***

**где клауза одна из:**

- **simdlen (length)**
  the largest size for a vector that the compiler is free to assume
- **linear (argument-list[:constant-linear-step])**
  in serial execution parameters are incremented by steps (induction variables with constant stride)
- **aligned (argument-list[:alignment])**
  all arguments in the argument-list are aligned on a known boundary not less than the specified alignment
- **uniform (argument-list)**
  shared, scalar parameters are broadcasted to all iterations
- **inbranch**
- **notinbranch**

# Использование векторизации

```
#pragma omp declare simd notinbranch
float min(float a, float b) {
   return a < b ? a : b;
}

#pragma omp declare simd notinbrach
float distance (float x, float y) {
   return (x - y) * (x - y);
}

#pragma omp parallel for simd
for (i=0; i<N; i++)
   d[i] = min (distance (a[i], b[i]), c[i]);
```

# Cancellation Constructs

Директива

**#pragma omp cancel** ***clause[[,] clause ]***

*где* ***clause*** *одна из:*

- **parallel**
- **sections**
- **for**
- **taskgroup**
- **if** (*scalar-expression*)

Директива

**#pragma omp cancellation point** ***clause[[,] clause ]***

*где* ***clause*** *одна из:*

- **parallel**
- **sections**
- **for**
- **taskgroup**

**Новая функция системы поддержки:**

- **omp_get_cancellation**

**Новая переменная окружения:**

- **OMP_CANCELLATION**

# Обработка исключительных ситуаций

```
void example() {
   std::exception *ex = NULL;
   #pragma omp parallel shared(ex)
   {
      #pragma omp for
      for (int i = 0; i < N; i++) {
         try {
            causes_an_exception();
         } catch (std::exception *e) {
            #pragma omp atomic write
               ex = e; // still must remember exception for later handling
            #pragma omp cancel for // cancel worksharing construct
         }
      }
      if (ex) { // if an exception has been raised, cancel parallel region
        #pragma omp cancel parallel
      }
   }
   if (ex) {  // handle exception stored in ex
   }
}
```

# Поиск в дереве (часть 1)

```
typedef struct binary_tree_s {
   int value;
   struct binary_tree_s *left, *right;
} binary_tree_t;

binary_tree_t *search_tree_parallel (binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
    {
#pragma omp master
        {
#pragma omp taskgroup
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}
```

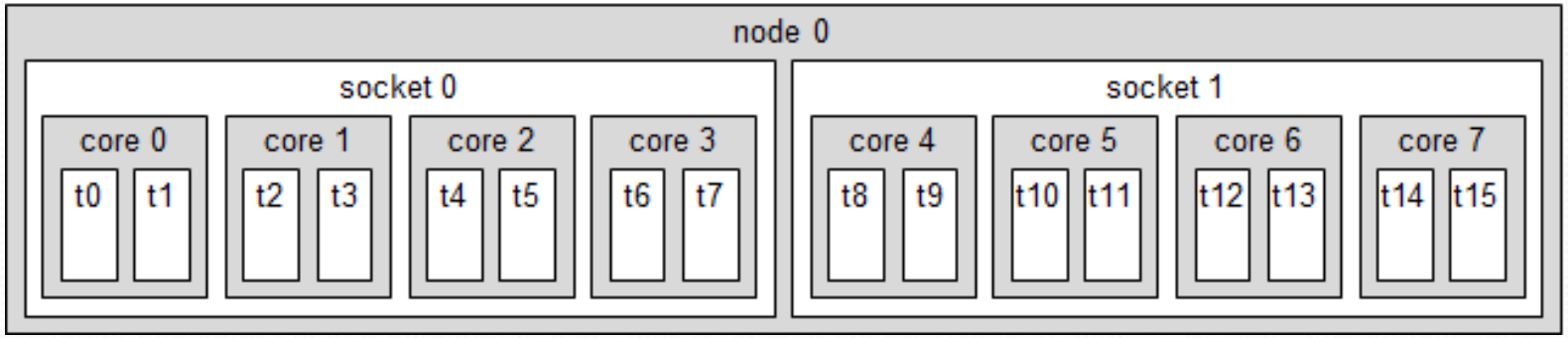
# Поиск в дереве (часть 2)

```
binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
    binary_tree_t *found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        } else {
            #pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_left = NULL;
                found_left = search_tree(tree->left, value, level + 1);
                if (found_left) {
                    #pragma omp atomic write
                      found = found_left;
                    #pragma omp cancel taskgroup
                }
            }
```

# Поиск в дереве (часть 3)

```
            #pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_right = NULL;
                found_right = search_tree(tree->right, value, level + 1);
                if (found_right) {
                    #pragma omp atomic write
                        found = found_right;
                    #pragma omp cancel taskgroup
                }
            }
            #pragma omp taskwait
        }
    }
    return found;
}
```

# Привязка нитей к ядрам




**setenv OMP_PLACES=cores**

**setenv OMP_PLACES="(0,1),(2,3),(4,5),(6,7),(8,9),(10,11),(12,13),(14,15)"**
**setenv OMP_PLACES="(0:2):2:8"**

**setenv OMP_PLACES threads**

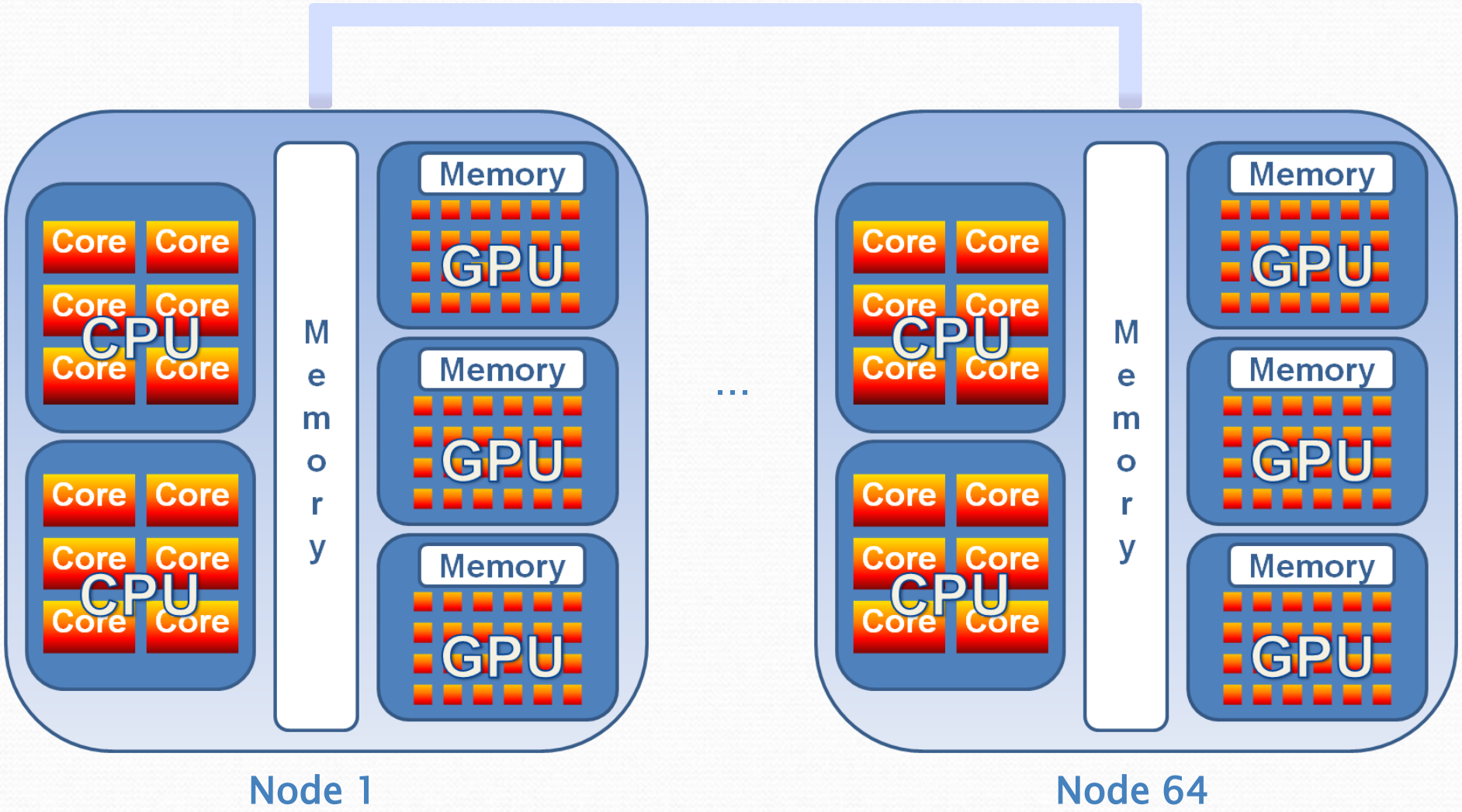
**setenv OMP_PLACES "threads(4)"**

**setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"**

**setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"**

**setenv OMP_PLACES "{0:4}:4:4"**

# Расширение OpenMP для использования ускорителей



# Алгоритм Якоби на языке Fortran

```
      PROGRAM    JACOB_SEQ
      PARAMETER    (L=4096,  ITMAX=100)
      REAL     A(L,L), B(L,L)
      PRINT *,  '**********  TEST_JACOBI   **********'
      DO IT  =  1,  ITMAX
          DO  J  =  2, L-1
              DO  I  =  2, L-1
                  A(I, J)  =  B(I, J)
              ENDDO
          ENDDO
          DO  J = 2,  L-1
              DO  I = 2,  L-1
                  B(I, J) =  (A(I-1, J) + A(I, J-1) +  A(I+1, J) +
     *                       A(I, J+1)) / 4
              ENDDO
           ENDDO
       ENDDO
       END
```

# Алгоритм Якоби на языке Fortran Cuda

```
PROGRAM JACOB_CUDA
      use cudafor
      use jac_cuda
      PARAMETER (L=4096,  ITMAX=100)
      parameter (block_dim = 16)
      real, device, dimension(l, l) :: a, b
      type(dim3) :: grid, block
      PRINT *, '*****  TEST_JACOBI   *******'
      grid = dim3(l / block_dim, l / block_dim, 1)
      block = dim3(block_dim, block_dim, 1)
      DO IT = 1, ITMAX
         call arr_copy<<<grid, block>>>(a, b, l)
         call arr_renew<<<grid, block>>>(a, b, l)
      ENDDO
END
```

# Алгоритм Якоби на языке Fortran Cuda

```
module jac_cuda
      contains
      attributes(global) subroutine arr_copy(a, b, k)
        real, device, dimension(k, k) :: a, b
        integer, value :: k
        integer i, j
        i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
        j = (blockIdx%y - 1) * blockDim%y + threadIdx%y
        if (i.ne.1 .and. i.ne.k .and. j.ne.1 .and. j.ne.k) A(I, J) = B(I, J)
      end subroutine arr_copy
      attributes(global) subroutine arr_renew(a, b, k)
        real, device, dimension(k, k) :: a, b
        integer, value :: k
        integer i, j
        i = (blockIdx%x - 1) * blockDim%x + threadIdx%x
        j = (blockIdx%y - 1) * blockDim%y + threadIdx%y
       if (i.ne.1 .and. i.ne.k .and. j.ne.1 .and. j.ne.k)  B(I,J) =(A( I-1,J)+A(I,J-1)+A(I+1,J)+
A(I,J+1))/4
      end subroutine arr_renew
end module jac_cuda
```

# Алгоритм Якоби в модели HMPP

```
!$HMPP jacoby codelet, target = CUDA
      SUBROUTINE JACOBY(A,B,L)
      IMPLICIT NONE
      INTEGER, INTENT(IN) :: L
      REAL, INTENT(IN) :: A(L,L)
      REAL, INTENT(INOUT) :: B(L,L)
      INTEGER I,J
      DO  J  =  2, L-1
          DO  I  =  2, L-1
              A(I,J)  =  B(I,J)
          ENDDO
      ENDDO
       DO  J = 2,  L-1
          DO  I = 2,  L-1
              B(I,J) =  (A(I-1,J ) + A(I,J-1 ) +
     *             A(I+1,J ) + A(I,J+1 )) / 4
          ENDDO
       ENDDO
      END SUBROUTINE JACOBY
```

```
      PROGRAM JACOBY_HMPP
      PARAMETER    (L=4096,  ITMAX=100)
      REAL     A(L,L), B(L,L)
       PRINT *,  '**********TEST_JACOBI**********'
      DO IT  =  1,  ITMAX
!$HMPP   jacoby callsite
        CALL JACOBY(A,B,L)
      ENDDO
      PRINT *, B
      END
```

# Алгоритм Якоби в модели HMPP

```
          PROGRAM    JACOBY_HMPP
           PARAMETER    (L=4096,  ITMAX=100)
           REAL     A(L,L), B(L,L)
!$hmpp jac allocate, args[A;B].size={L,L}
!$hmpp jac advancedload, args[B]
           PRINT *,  '**********  TEST_JACOBI   **********'
           DO IT  =  1,  ITMAX
!$hmpp jac region, args[A;B].noupdate=true
                DO  J  =  2, L-1
                   DO  I  =  2, L-1
                        A(I, J)  =  B(I, J)
                   ENDDO
                ENDDO
               DO  J = 2,  L-1
                  DO  I = 2,  L-1
                      B(I, J)=(A(I-1,J)+A(I,J-1)+A(I+1,J) +
          *                 A(I, J+1)) / 4
                  ENDDO
               ENDDO
!$hmpp jac endregion
           ENDDO
!$hmpp jac delegatedstore, args[B]
!$hmpp jac release
           PRINT *,B
           END
```

# Алгоритм Якоби в модели PGI APM

```
      PROGRAM JACOBY_PGI_APM
      PARAMETER    (L=4096,  ITMAX=100)
      REAL     A(L,L), B(L,L)
      PRINT *,  '**********  TEST_JACOBI   **********'
!$acc data region copyin(B), copyout(B), local(A)
      DO IT  =  1,  ITMAX
!$acc     region
            DO  J  =  2, L-1
                DO  I  =  2, L-1
                    A(I,J)  =  B(I,J)
                ENDDO
            ENDDO
            DO  J = 2,  L-1
                DO  I = 2,  L-1
                    B(I,J) =  (A(I-1,J ) + A(I,J-1 ) + A(I+1,J ) + A(I,J+1 )) / 4
                ENDDO
            ENDDO
!$acc     end  region
      ENDDO
!$acc end data region
      PRINT *, B
      END
```

# Cray Compiling Environment 7.4.0

```
!$omp acc_region
!$omp acc_loop
          DO j = 1,M
            DO i = 2,N
                c(i,j) = a(i,j) + b(i,j)
             ENDDO
           ENDDO
!$omp end acc_loop
!$omp end acc_region
```

**acc_region:**

acc_copy, acc_copyin, acc_copyout, acc_shared, private, firstprivate, default(<any of above>|none), present, if(scalar-logical-expression), device(integer-expression), num_pes(depth:num [, depth:num]), async(handle)

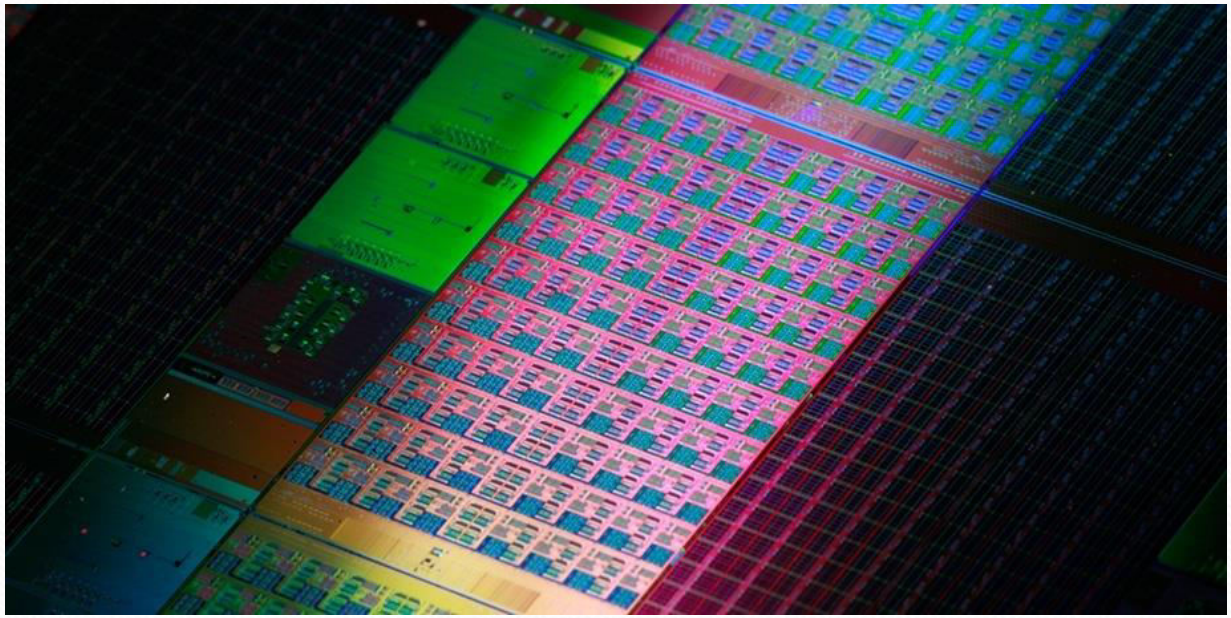
**acc_loop:**

reduction(operator:list), collapse(n), schedule, cache(obj[:depth], hetero…

```
#pragma acc data copy(A), create(Anew)
while (iter<iter_max) {
  #pragma acc kernels loop
  for (int j = 1; j < n-1; j++) {
     for (int I = 1; I < m-1; i++) {
        Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1] +A[j-1][i] + A[j+1][i]);
     }
   }
  #pragma acc kernels loop
  for (int j = 1; j < n-1; j++) {
     for (int I = 1; i< m-1;i++ ) {
       A[j][i] = Anew[j][i];
     }
  }
  iter++;
 }
```

# Intel Many Integrated Core (MIC)

```
!dir$ offload target(mic)
!$omp parallel do
        do i=1,10
            A(i) = B(i) * C(i)
        enddo
!$omp end parallel
```



# OpenMP accelerator model

**Новые директивы**

- target
- target data
- target update
- teams
- ditribute

**Новые функции системы поддержки**

- omp_get_num_devices
- omp_set_default_device
- omp_get_default_device
- omp_is_initial_device
- omp_get_num_teams
- omp_get_team_num

**Новая переменная окружения**

- OMP_DEFAULT_DEVICE

# OpenMP accelerator model. Директива target

**#pragma omp target** ***[clause[[,] clause ]]***
***structured-block***

где ***clause*** *одна из:*

- **device(***integer-expression***)**
- **map** *([map-type]:list)*

*map-type:*

- **alloc**
- **to**
- **from**
- **tofrom (по умолчанию)**

- **if (***scalar-expression***)**

```
sum=0;
#pragma omp target device(acc0) map(A,B)
#pragma omp parallel for reduction(+: sum)
   for (i=0;i<N;i++)
        sum += A[i]*B[i];
```

# OpenMP accelerator model

**#pragma omp target data** ***[clause[[,] clause ]]***
***structured-block***

*где* ***clause*** *одна из:*
- **device(***integer-expression***)**
- **map** *([map-type]:list)*

*map-type:*
  - **alloc**
  - **to**
  - **from**
  - **tofrom**

- **if (***scalar-expression***)**

**#pragma omp target update** ***[clause[[,] clause ]]***

*где* ***clause*** *одна из:*
- **to** *(list)*
- **from** *(list)*
- **device(***integer-expression***)**
- **if (***scalar-expression***)**

# OpenMP accelerator model. Директива target data

```
#pragma omp target data device(acc0) map(alloc: tmp[0:N]) \
        map(to: input[:N)) map(from: output)
{
  #pragma omp target device(acc00)
  #pragma omp parallel for
    for (int i=0; i<N; i++)
        tmp[i] = some_device_computation (input[i]);

  input[0] = some_host_computation ();
  #pragma omp target update to (input[0]) device(acc0)

  #pragma omp target device(acc0)
  #pragma omp parallel for reduction(+: output)
     for (int i=0; i<N; i++) output += final_device_computation (tmp[i], input[i])
}
```

# OpenMP accelerator model. Директива declare target

```
#pragma omp declare target
   function-defenition-or-declaration

#pragma omp declare target
   float Q[N][N];
   #pragma omp declare simd uniform(i) linear(j) notinbranch
   float func(const int i, const int j)
   {
      return Q[i][j] * Q[j][i];
   }
#pragma omp end declare target

…
#pragma omp target
#pragma omp parallel for reduction(+: sum)
  for (int i=0; i < N; i++) {
    for (int j=0; j < N; j++) {
        sum += func (i,j);
    }
  }

…
```

# OpenMP accelerator model. Директива teams

**#pragma omp teams** *[clause[ [, ]clause] ,...]*
*structured-block*

*где clause одна из:*

- **num_teams** *(integer-expression)*
- **thread_limit** *(integer-expression)*
- **private** *(list)*
- **firstprivate** *(list)*
- **shared** *(list)*
- **default (shared | none)**
- **reduction** *(reduction-identifier: list)*

# Использование директивы teams

```
float dotprod(float B[], float C[], int N)
{
  float sum0 = 0.0, sum1 = 0.0;
  #pragma omp target map(to: B[:N], C[:N])
  #pragma omp teams num_teams(2)
  {
    if (omp_get_team_num() == 0)
    {
      #pragma omp parallel for reduction(+:sum0)
      for (int i=0; i<N/2; i++)
        sum0 += B[i] * C[i];
    } else if (omp_get_team_num() == 1) {
      #pragma omp parallel for reduction(+:sum1)
      for (int i=N/2; i<N; i++)
        sum1 += B[i] * C[i];
    }
  }
  return sum0 + sum1;
}
```

# OpenMP accelerator model. Директива distribute

**#pragma omp distribute** *[clause[ [, ]clause] ,...]*
*for-loops*

*где clause одна из:*

- **private** *(list)*
- **firstprivate** *(list)*
- **collapse** *(n)*
- **dist_schedule** *(kind[,: chunk_size]) // kind=static*

Может использоваться внутри конструкции **teams**.

# OpenMP accelerator model. Директива distribute

```
float dotprod(float B[], float C[], int N)
{
  float sum = 0;
  int i;
  #pragma omp target teams map(to: B[0:N], C[0:N])
  #pragma omp distribute parallel for reduction(+:sum)
  for (i=0; i<N; i++)
    sum += B[i] * C[i];
  return sum;
}
```

# OpenMP accelerator model. Директивы teams&&distribute

```
#pragma omp declare target
extern void func(int, int, int);

#pragma omp target device(0)
#pragma omp teams num_teams(60) num_threads (4)
// 60 physical cores, 4 threads in each team
{
  #pragma omp distribute // this loop is distributed across teams
  for (int i = 0; i < 2048; i++) {
    #pragma omp parallel for // loop is executed in parallel by 4 threads of team
    for (int j = 0; j < 512; j++) {
      #pragma omp simd  // create SIMD vectors for the machine
      for (int k=0; k<32; k++) {
          func (i,j,k);
      }
    }
  }
}
```

# OpenMP accelerator model. Умножение векторов

```
extern void func_on_host();
extern void output_on_host(float *,int);
#pragma omp declare target
extern void init_on_device(float *,float *,int);

void vec_mult(float *p, int N, int dev)
{
  float *v1, *v2; int i;
  #pragma omp task shared(v1, v2) depend(out: v1, v2)
  #pragma omp target device(dev) map(v1, v2)
  {
     v1=malloc(N*sizeof(float)); v2=malloc(N*sizeof(float)); init_on_device(v1,v2,N);
  }
  func (); // execute other work asychronously
  #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
  #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
  {
     #pragma omp parallel for
        for (i=0; i<N; i++) p[i] = v1[i] * v2[i];
     free(v1); free(v2);
  }
  #pragma omp taskwait
  output(p, N);
}
```

# Литература


- **OpenMP Application Program Interface Version 4.0, July 2013.** http://www.openmp.org/mp-documents/OpenMP4.0.0.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/Academy2015/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