Суперкомпьютеры и параллельная обработка данных

Бахтин Владимир Александрович
к.ф.-м.н., ведущий научный сотрудник
Института прикладной математики им М.В.Келдыша РАН
кафедра системного программирования
факультет вычислительной математики и кибернетики
Московского университета им. М.В. Ломоносова

Вычисление числа π с использованием POSIX threads

```
#include <pthread.h>
#define NUM_RECT 100000
#define NUM_THREADS 2
double gPi = 0.0;
pthread_mutex_t gLock;
int main(int argc, char **argv) {
    pthread_t tHandles[NUM_THREADS];
    int tNum[NUM_THREADS], i;
    pthread_mutex_init(&gLock, NULL);
    for ( i = 0; i < NUM_THREADS; ++i ) {
            tNum[i] = i;
            pthread_create(&tHandles[i], // Returned thread handle
            NULL, // Thread Attributes
            Pi, // Thread function
           (void*)&tNum[i]); // Data
    }
    for ( i = 0; i < NUM_THREADS; ++i ) {
        pthread_join(tHandles[i], NULL);
    }
   printf("pi is approximately %.16f”, gPi);
   pthread_mutex_destroy(&gLock);
   return 0;
}
```

Вычисление числа π с использованием POSIX threads

```
void *Pi (void *pArg) {
        int myNum = *((int *)pArg);
        double h = 1.0 / NUM_RECT;
        double partialSum = 0.0, x;  // local to each thread
        for (int i = myNum; i < NUM_RECT; i += NUM_THREADS) {
              x = h * ((double)i - 0.5);
              partialSum += (4.0 / (1.0 + x*x));
        }
        pthread_mutex_lock(&gLock);
        gPi += partialSum;  // add partial to global final answer
        pthread_mutex_unlock(&gLock);
        return 0;
}
```

Вычисление числа π с использованием TBB

```
#include "tbb/parallel_reduce.h“
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
using namespace std;
using namespace tbb;
long n = 1000000;
…
double pi;
double width = 1./(double) n;
MyPi area(&width);   //construct MyPi with initializer of step(&width)
parallel_reduce(blocked_range<size_t>(0,n),
                        area,
                        auto_partitioner());
pi = area.sum ;
...
```

Вычисление числа π с использованием TBB

```
class MyPi {
  double *const my_h;
public:
  double sum;
  void operator()( const blocked_range<size_t>& r ) {
    double h = *my_h;
    double x;
    for (size_t i = r.begin(); i != r.end(); ++i) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
     }
   }
   void join( const MyPi& y ) {
      sum += y.sum;
   }
   MyPi(double *const width) : my_h(width), sum(0) {}
   MyPi( MyPi& x, split ) : my_h(x.my_h), sum(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;
}
```

Вычисление числа π с использованием MPI

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

Вычисление числа π с использованием MPI

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

Вычисление числа π с использованием SHMEM

```
#include <shmem.h>
#include <stdio.h>
long sync[SHMEM_REDUCE_SYNC_SIZE] = {SHMEM_SYNC_VALUE};
double work[SHMEM_REDUCE_MIN_WRKDATA_SIZE];
double pi;
int main (int argc, char *argv[])
{
    int n =100000, myid, numprocs, i;
    double h, sum, x;
    shmem_init();
    numprocs = shmem_n_pes();
    myid = shmem_my_pe();
    h   = 1.0 / (double) n;
    sum = 0.0;
```

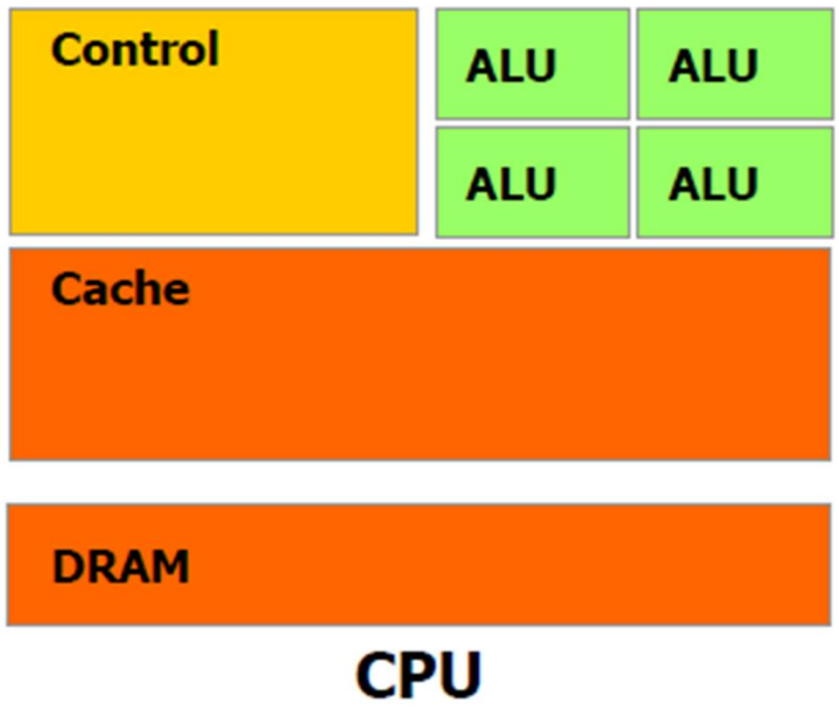
Вычисление числа π с использованием SHMEM

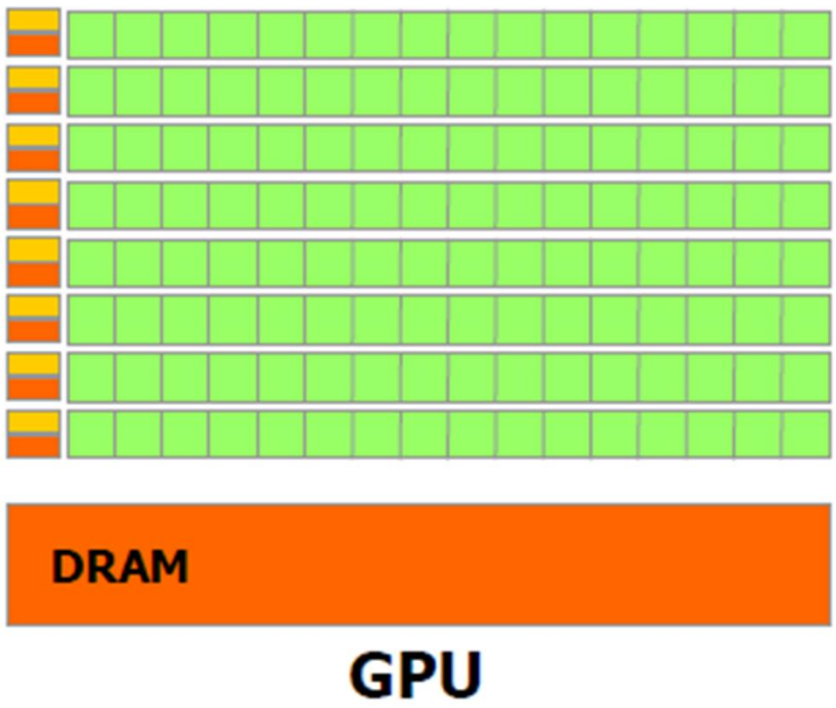
```
    for (i = myid + 1; i <= n; i += numprocs)
    {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
    }
    pi = h * sum;
    shmem_double_sum_to_all(&pi, &pi, 1, 0, 0, numprocs, work, sync);
    if (myid == 0) printf("pi is approximately %.16f”, pi);
    shmem_finalize();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

Вычисление числа π с использованием UPC

```
#include <stdio.h>
#include <upc.h>
#include <upc_collective.h>
shared double pi;
shared double sum[THREADS];
int main ()
{
    int n =100000, i;
    double h, x;
    h   = 1.0 / (double) n;
    sum[MYTHREAD] = 0.0;
    for (i=1+MYTHREAD; i<=n;i+= THREADS)
    {
        x = h * ((double)i - 0.5);
        sum[MYTHREAD] += (4.0 / (1.0 + x*x));
     }
upc_all_reducel (&pi, sum, UPC_ADD,
  THREADS, 1,NULL,
  UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC);
 if (MYTHREAD == 0)
 {
    pi *= h;
    printf("pi is approximately %.16f", pi);
 }
return 0;
}
```

CPU или GPU





Вычисление числа π с использованием CUDA

```
#include <stdio.h>
#include <cuda.h>
#define N 1000000
#define NUM_BLOCK  32  // Number of thread blocks
#define NUM_THREAD  32  // Number of threads per block
int tid;
float pi = 0;
// Kernel that executes on the CUDA device
__global__ void cal_pi(float *sum) {
      int i;
      float x, step=1.0/N; // Step size
      // Sequential thread index across the blocks
      int idx = blockIdx.x*blockDim.x+threadIdx.x;
      for (i=idx; i< N; i+=NUM_BLOCK*NUM_THREAD) {
          x = (i+0.5)*step;
          sum[idx] += 4.0/(1.0+x*x);
      }}
```

http://cacs.usc.edu/education/cs596/src/cuda/pi.cu

Вычисление числа π с использованием CUDA

```
int main(void) {  // Main routine that executes on the host
    dim3 dimGrid(NUM_BLOCK,1,1);  // Grid dimensions
    dim3 dimBlock(NUM_THREAD,1,1);  // Block dimensions
    float *sumHost, *sumDev;  // Pointer to host & device arrays
    size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float);  //Array size
    sumHost = (float *)malloc(size);  // Allocate array on host
    cudaMalloc((void **) &sumDev, size); // Allocate array on device
    cudaMemset(sumDev, 0, size); // Initialize array in device to 0
    // Do calculation on device
    cal_pi <<<dimGrid, dimBlock>>> (sumDev); // call CUDA kernel
    // Retrieve result from device and store it in host array
    cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
    for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++) pi += sumHost[tid];
    pi *= step;
    printf("PI = %f\n",pi); // Print results
    free(sumHost);  // Cleanup
    cudaFree(sumDev);
    return 0;
}
```

http://cacs.usc.edu/education/cs596/src/cuda/pi.cu

Вычисление числа π с использованием OpenACC

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma acc parallel loop
    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;
}
```

```
pgcc -acc test.c -Minfo=all
main:
     8, Accelerator kernel generated
        Generating Tesla code
         9, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
        12, Sum reduction generated for sum
```

Вычисление числа π с использованием DVM

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, s, x;
#pragma dvm template[n] distribute[block]
    void *tmp;
    h   = 1.0 / (double) n;
    s = 0.0;
#pragma dvm parallel (i on tmp[i]) reduction(sum(s))  private(x)
    for (i = 1; i <= n; i ++)
    {
            x = h * ((double)i - 0.5);
            s += (4.0 / (1.0 + x*x));
     }
     pi = h * s;
     printf("pi is approximately %.16f”, pi);
     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;
}
```

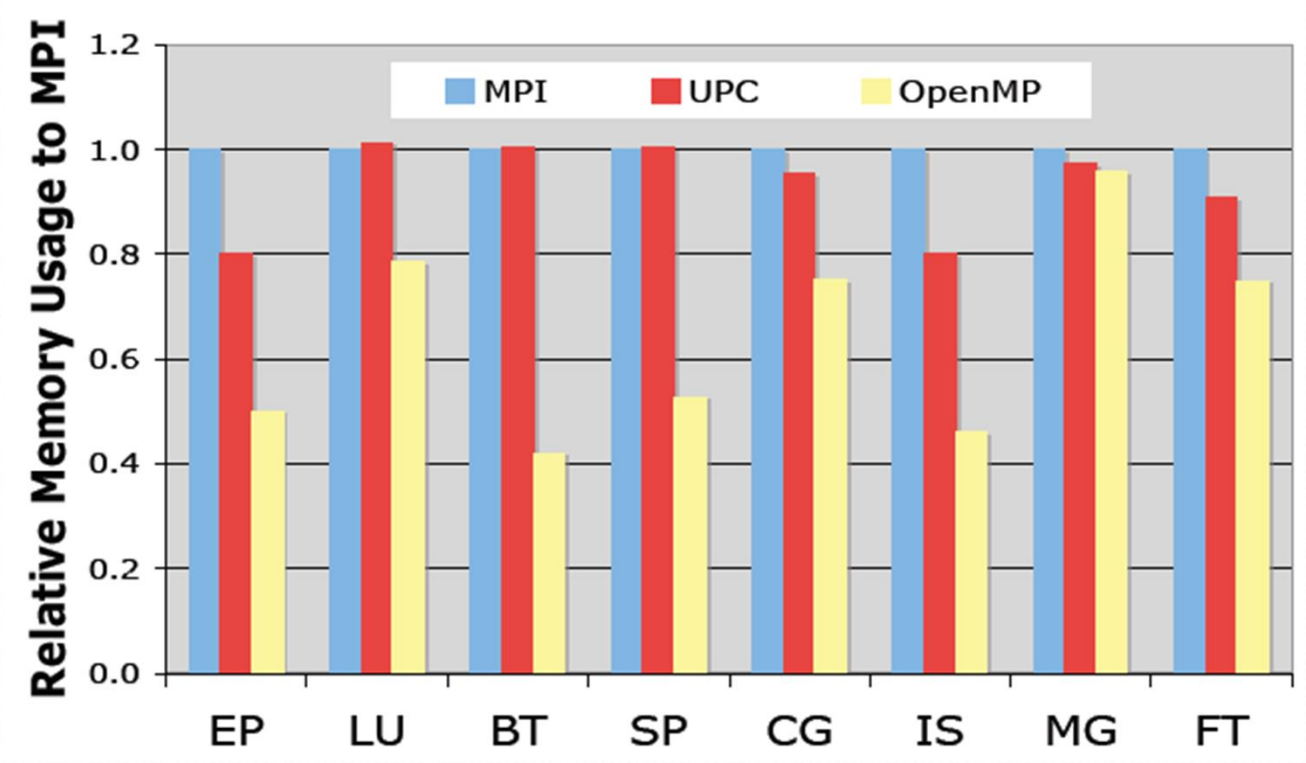
Достоинства использования OpenMP вместо MPI для многоядерных процессоров

- 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

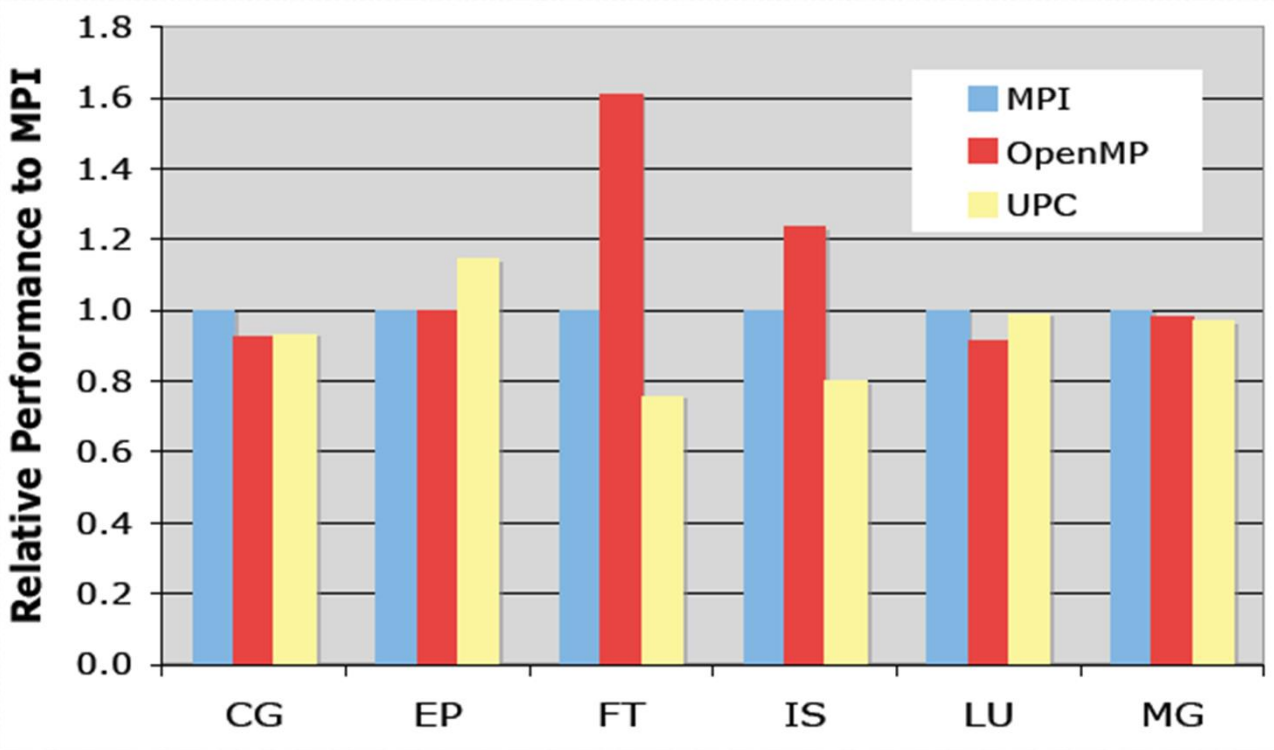
Тесты NAS



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

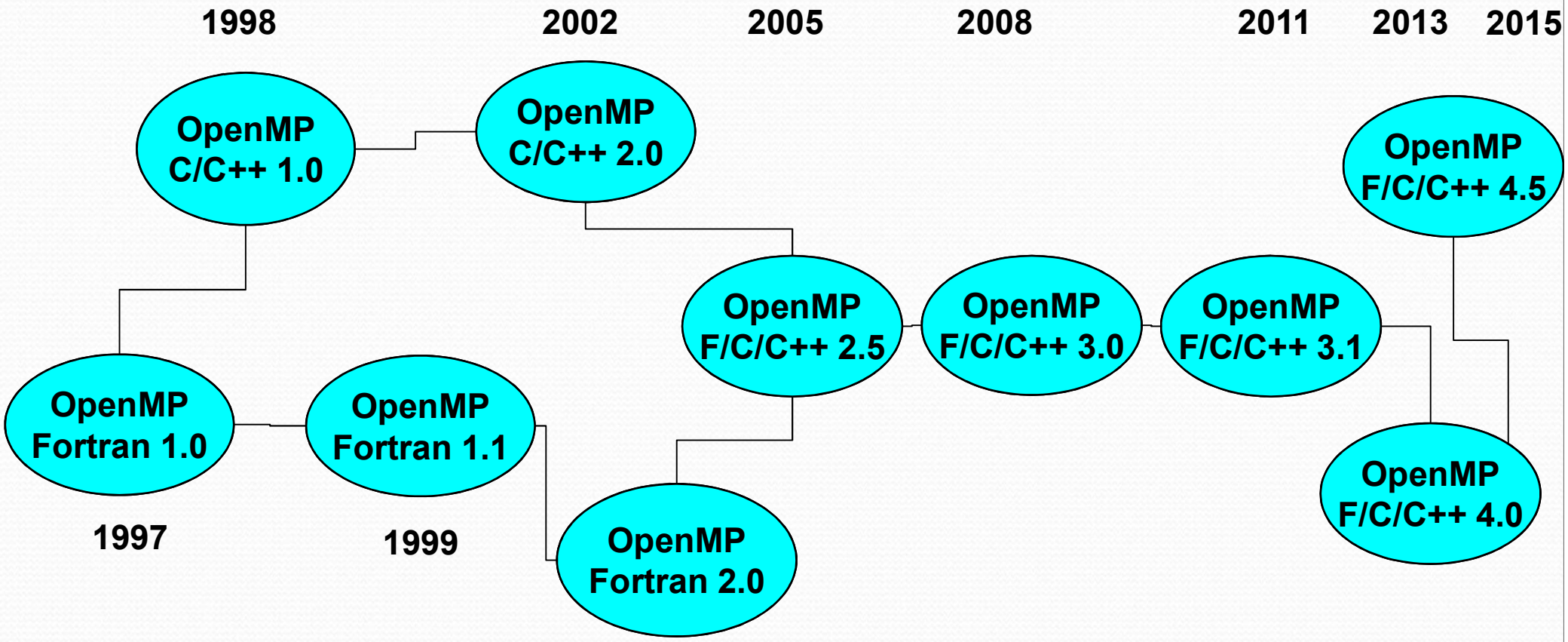


Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms

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

Содержание

История OpenMP



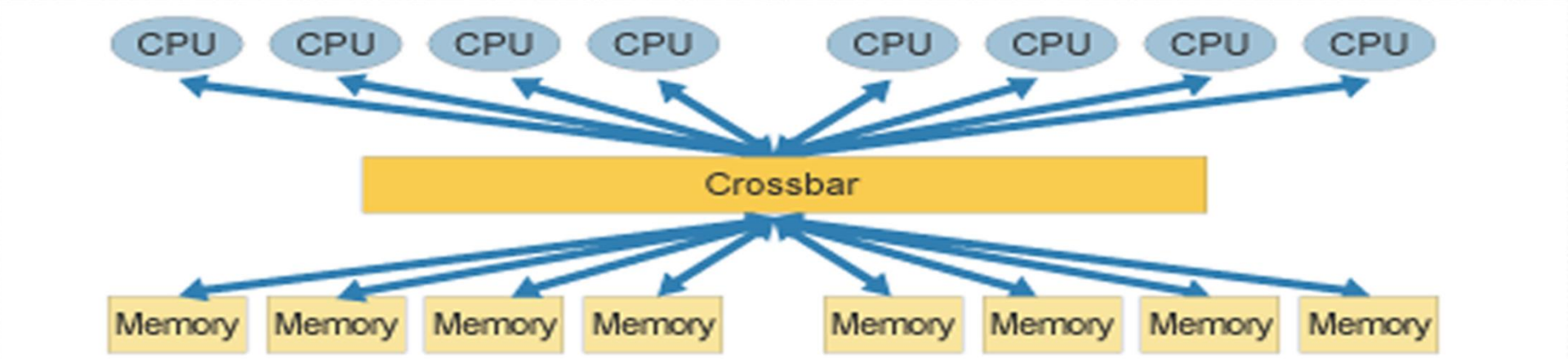
История OpenMP

Month/Year	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2018	OpenMP 5.0
Nov 2020	OpenMP 5.1
Nov 2021	OpenMP 5.2

OpenMP Architecture Review Board

- AMD
- ARM
- Cray
- Fujitsu
- HP
- IBM
- Intel
- Micron
- NEC
- NVIDIA
- Oracle Corporation
- Red Hat
- Texas Instrument
- ANL
- ASC/LLNL
- cOMPunity
- EPCC
- LANL
- LBNL
- NASA
- ORNL
- RWTH Aachen University
- Texas Advanced Computing Center
- SNL- Sandia National Lab
- BSC - Barcelona Supercomputing Center
- University of Houston

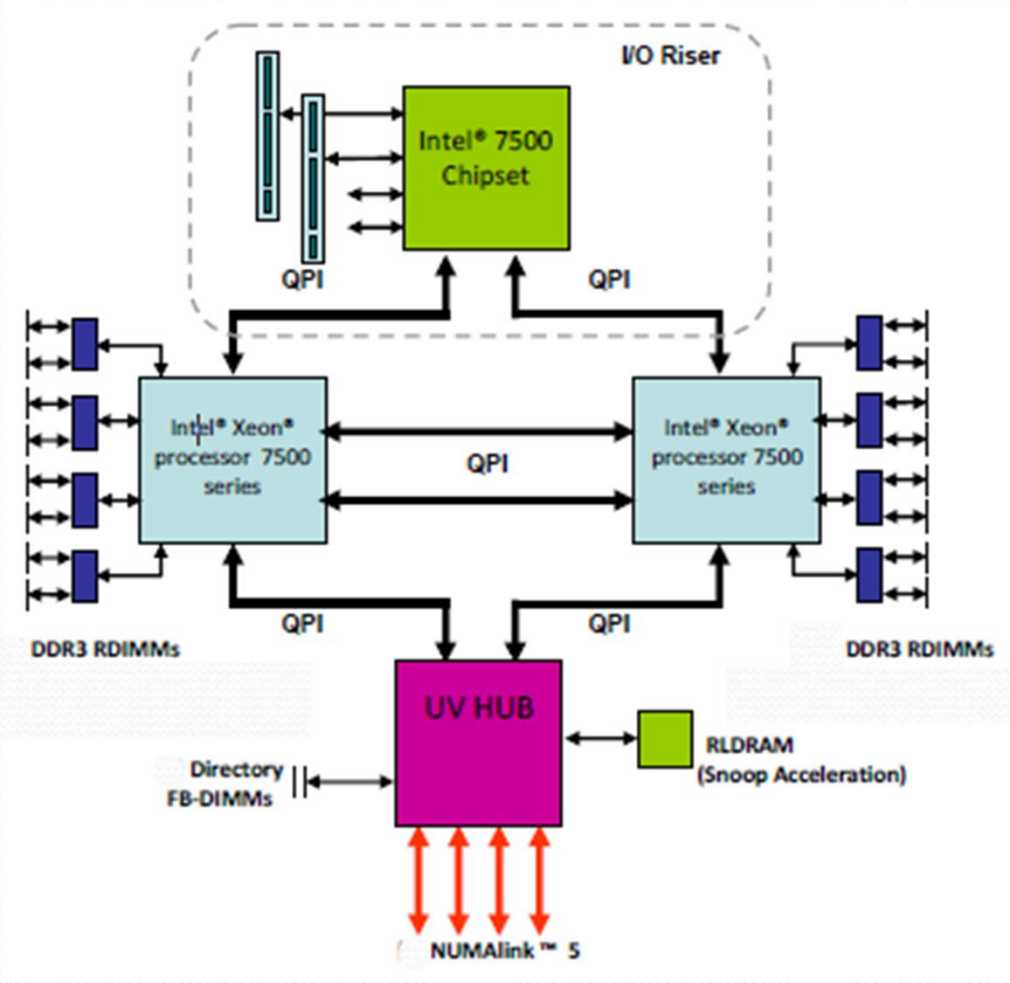
Симметричные мультипроцессорные системы (SMP)



- 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

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

```
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)
C$OMP SECTIONS
#pragma omp parallel for private(a, b)
C$OMP BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO LASTPRIVATE(XX)
nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

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***
- ***flush***
- ***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 4.5:

- GNU gcc (6.1): Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS
- Intel 17.0: Linux, Windows and MacOS
- LLVM: clang (3.9) Linux and MacOS
- HPE CCE Compiling Environment (CCE) 11.0

OpenMP 4.0:

- Oracle Developer Studio12.5: Linux and Solaris
- Cray Compiling Environment (CCE) 8.5

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
- Microsoft Visual Studio 2008 C++

http://www.openmp.org/resources/openmp-compilers/

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

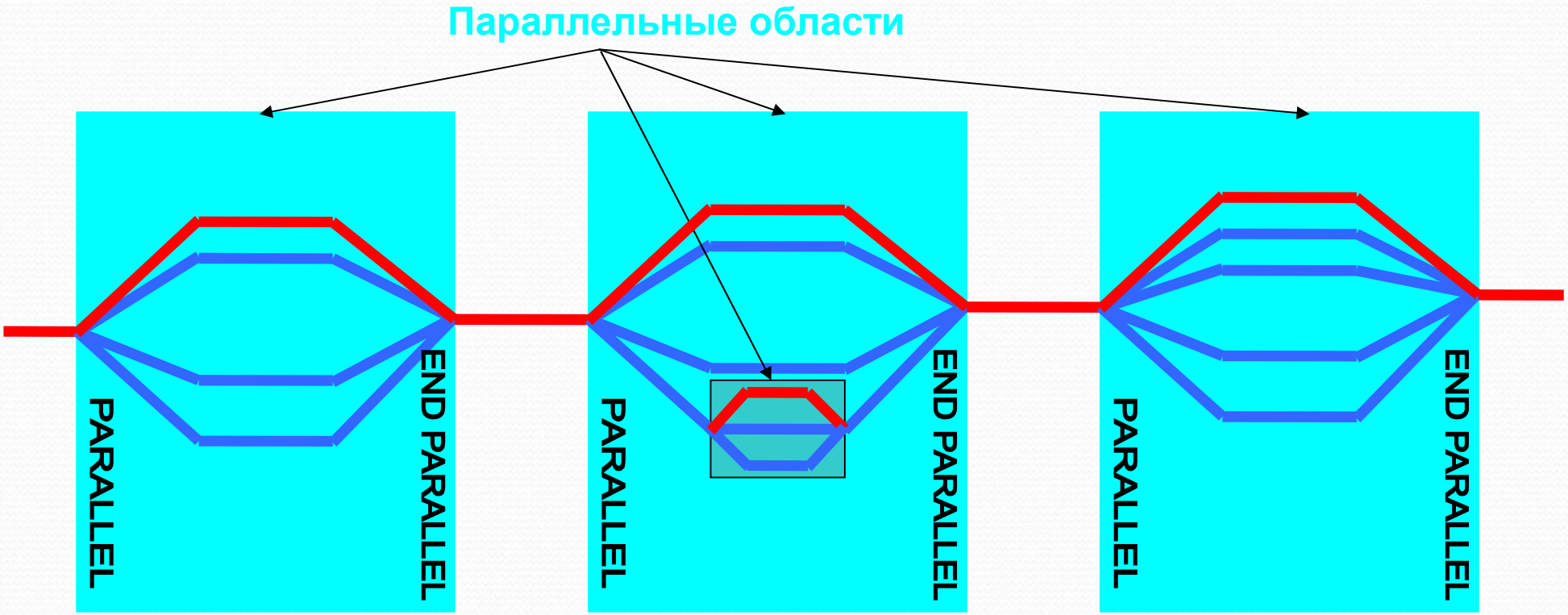
```
#include <stdio.h>
#include <omp.h> // Описаны прототипы всех функций и типов
int main()
{
#ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
    int id = omp_get_max_threads ();
#endif
    return 0;
}
```

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

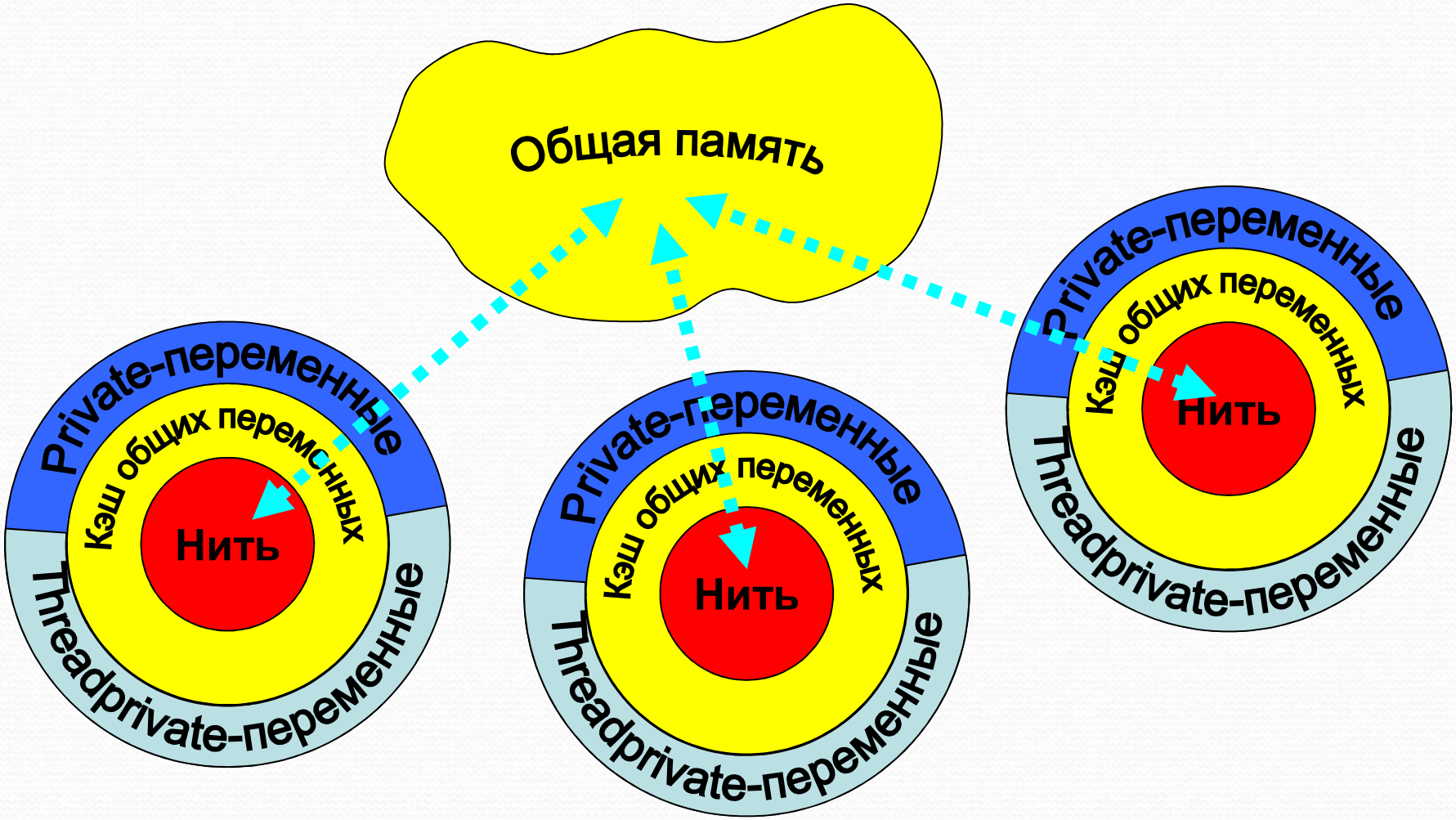
Выполнение OpenMP-программы

Fork-Join параллелизм:

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



Модель памяти в OpenMP



Фрагмент программы

```
int a, b, c, d, x, y;          // переменные
int *p, *q;                    // указатели
int f(int *p, int *q);         // прототип функции
...
a = x * (x - 1);
b = y * (y + 1);
c = a * a + a * b + b * b;
d = a * b * c;
p = &a;
q = &b;
x = f(p, q);
```

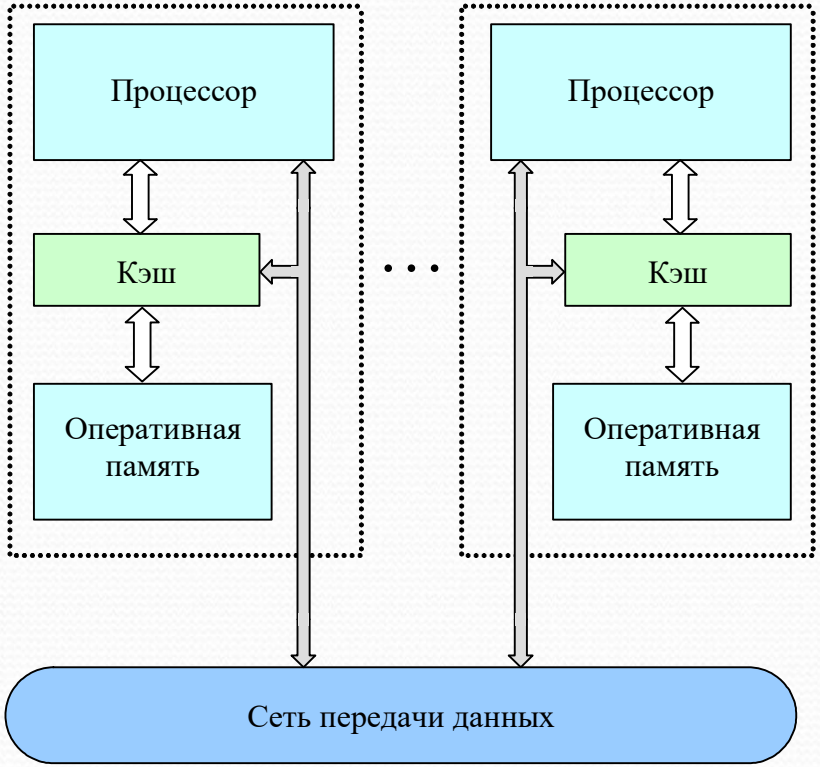
Фрагмент программы

```
int a, b, c, d, x, y;          // переменные
int *p, *q;                    // указатели
int f(int *p, int *q);         // прототип функции
...
a = x * (x - 1);               // a хранится в регистре
b = y * (y + 1);               // b хранится в регистре
c = a * a + a * b + b * b;     // будет использовано позднее
d = a * b * c;                 // будет использовано позднее
p = &a;                        // получает адрес a
q = &b;                        // получает адрес b
x = f(p, q);                   // вызов функции
```

Когерентность и консистентность памяти

P0: W(x)a R(y)b Время

P1: W(y)b R(y)b R(x)a W(x)b



Модели консистентности памяти

- **Модель консистентности** представляет собой некоторый договор между программами и памятью, в котором указывается, что при соблюдении программами определенных правил работа памяти будет корректной, если же требования к программе будут нарушены, то память не гарантирует правильность выполнения операций чтения/записи.

- Далее рассматриваются основные модели консистентности.

Строгая консистентность

- Операция чтения ячейки памяти с адресом **X** должна возвращать значение, записанное самой последней операцией записи с адресом **X**, называется моделью **строгой консистентности**.

а) строгая консистентность

б) нестрогая консистентность

Последовательная консистентность

- Впервые определил Lamport в 1979 г. в контексте совместно используемой памяти для мультипроцессорных систем.
- Результат выполнения должен быть тот-же, как если бы операторы всех процессоров выполнялись бы в некоторой последовательности, причем операции каждого отдельного процесса выполнялись бы в порядке, определяемой его программой.
- Последовательная консистентность не гарантирует, что операция чтения возвратит значение, записанное другим процессом наносекундой или даже минутой раньше, в этой модели только точно гарантируется, что все процессы знают последовательность всех записей в память.

Последовательная консистентность

- а) удовлетворяет последовательной консистентности

P1	**W(x)a**				
P2		**W(x)b**			
P3			**R(x)b**		**R(x)a**
P4				**R(x)b**	**R(x)a**

- б) не удовлетворяет последовательной консистентности

P1	**W(x)a**				
P2		**W(x)b**			
P3			**R(x)b**		**R(x)a**
P4				**R(x)a**	**R(x)b**

Причинная консистентность

- Предположим, что процесс P1 модифицировал переменную ***x***, затем процесс P2 прочитал ***x*** и модифицировал ***y***. В этом случае модификация ***x*** и модификация ***y*** потенциально причинно зависимы, так как новое значение ***y*** могло зависеть от прочитанного значения переменной ***x***. С другой стороны, если два процесса одновременно изменяют значения различных переменных, то между этими событиями нет причинной связи.
- Операции, которые причинно не зависят друг от друга называются параллельными.
- Причинная модель консистентности памяти определяется следующим условием: Последовательность операций записи, которые потенциально причинно зависимы, должна наблюдаться всеми процессами системы одинаково, параллельные операции записи могут наблюдаться разными процессами в разном порядке.

Причинная консистентность

P1	**W(x)a**				
P2		**R(x)a**	**W(x)b**		
P3				**R(x)b**	**R(x)a**
P4				**R(x)a**	**R(x)b**

Нарушение модели причинной консистентности

Корректная последовательность для модели причинной консистентности

P1	**W(x)a**			**W(x)c**		
P2		**R(x)a**	**W(x)b**			
P3		**R(x)a**			**R(x)c**	**R(x)b**
P4		**R(x)a**			**R(x)b**	**R(x)c**

Определение потенциальной причинной зависимости может осуществляться компилятором посредством анализа зависимости операторов программы по данным.

PRAM (Pipelined RAM) и процессорная консистентность

PRAM: Операции записи, выполняемые одним процессором, видны всем остальным процессорам в том порядке, в каком они выполнялись, но операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке.

Записи выполняемые одним процессором могут быть конвейеризованы: выполнение операций с общей памятью можно начинать не дожидаясь завершения предыдущих операций записи в память.

Процессорная: PRAM + когерентность памяти. Для каждой переменной **X** есть общее согласие относительно порядка, в котором процессоры модифицируют эту переменную, операции записи в разные переменные - параллельны. Таким образом, к упорядочиванию записей каждого процессора добавляется упорядочивание записей в переменные или группы.

Слабая консистентность (weak consistency)

- Пусть процесс в критической секции циклически читает и записывает значение некоторых переменных. Даже, если остальные процессоры и не пытаются обращаться к этим переменным до выхода первого процесса из критической секции, для удовлетворения требований рассматриваемых ранее моделей консистентности они должны видеть все записи первого процессора в порядке их выполнения, что, естественно, совершенно не нужно.
- Наилучшее решение в такой ситуации - это позволить первому процессу завершить выполнение критической секции и, только после этого, переслать остальным процессам значения модифицированных переменных, не заботясь о пересылке промежуточных результатов.

Слабая консистентность (weak consistency)

Модель слабой консистентности, основана на выделении среди переменных специальных синхронизационных переменных и описывается следующими правилами:

1. Доступ к синхронизационным переменным определяется моделью последовательной консистентности;
2. Доступ к синхронизационным переменным запрещен (задерживается), пока не выполнены все предыдущие операции записи;
3. Доступ к данным (запись, чтение) запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным.

Слабая консистентность (weak consistency)

- Первое правило определяет, что все процессы видят обращения к синхронизационным переменным в определенном (одном и том же) порядке.
- Второе правило гарантирует, что выполнение процессором операции обращения к синхронизационной переменной возможно только после выталкивания конвейера (полного завершения выполнения на всех процессорах всех предыдущих операций записи переменных, выданных данным процессором).
- Третье правило определяет, что при обращении к обычным (не синхронизационным) переменным на чтение или запись, все предыдущие обращения к синхронизационным переменным должны быть выполнены полностью. Выполнив синхронизацию перед обращением к общей переменной, процесс может быть уверен, что получит правильное значение этой переменной.

Слабая консистентность (weak consistency)

P1	W(x)a	W(x)b	S			
P2				R(x)a	R(x)b	S
P3				R(x)b	S	

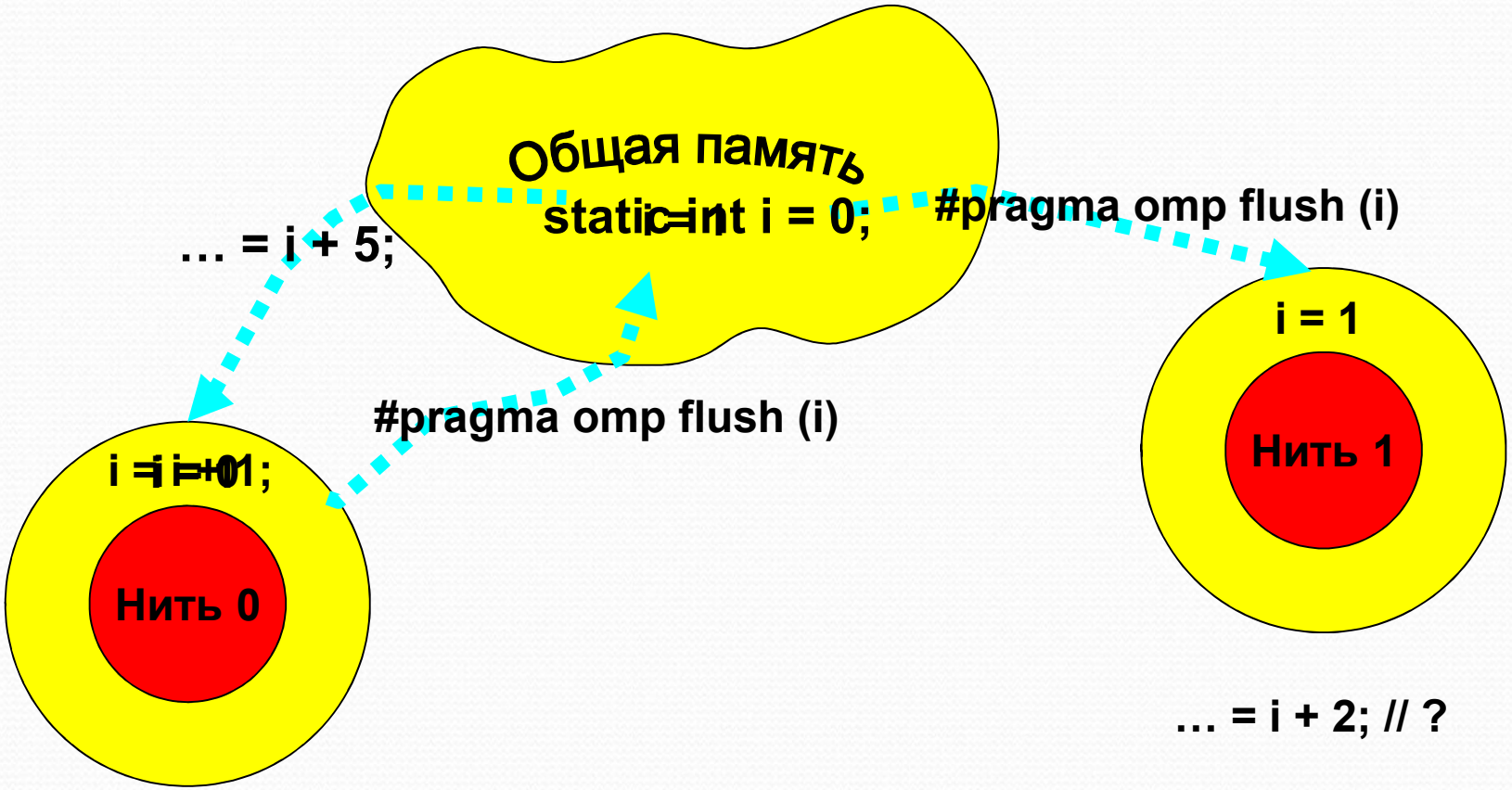
Допустимая последователь-ность событий

Недопустимая последовательность событий

P1	W(x)a	W(x)b	S		
P2				S	R(x)a

...

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**.

Консистентность памяти в OpenMP

1. Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).
2. Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке определяемом программой.
3. Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

Консистентность памяти в C++

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
enum class memory_order : /* unspecified */ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

Консистентность памяти в C++

```
std::atomic<bool> x, y;
std::atomic<int> z;
void thread_write_x() {
        x.store(true, std::memory_order_seq_cst);
}
void thread_write_y() {
        y.store(true, std::memory_order_seq_cst);
}
void thread_read_x_then_y() {
        while (!x.load(std::memory_order_seq_cst));
        if (y.load(std::memory_order_seq_cst)) { ++z; }
}
void thread_read_y_then_x() {
        while (!y.load(std::memory_order_seq_cst));
        if (x.load(std::memory_order_seq_cst)) { ++z; }
}
```

Консистентность памяти в C++

```
class mutex {
    public:
        void lock() {
            bool expected = false;
            while(!_locked.compare_exchange_weak(expected, true, std::memory_order_acquire)) {
                expected = false;
            }
        }
        void unlock() {
            _locked.store(false, std::memory_order_release);
        }
        private: std::atomic<bool> _locked;
};
```

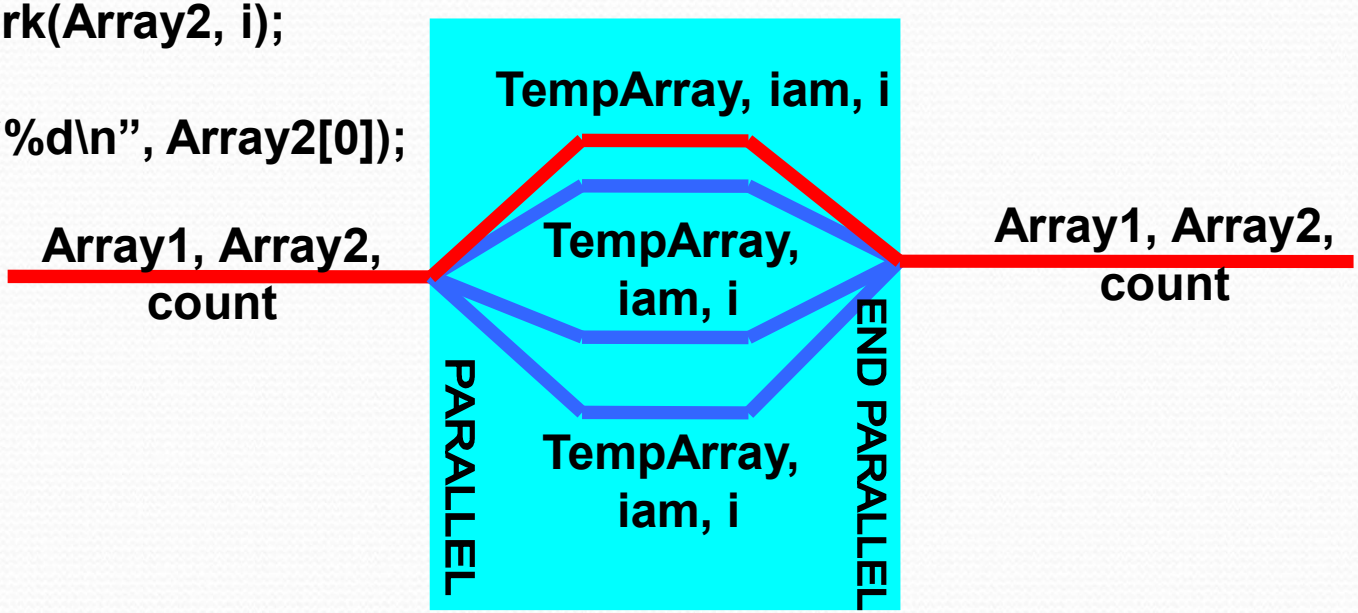
Классы переменных

- 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.

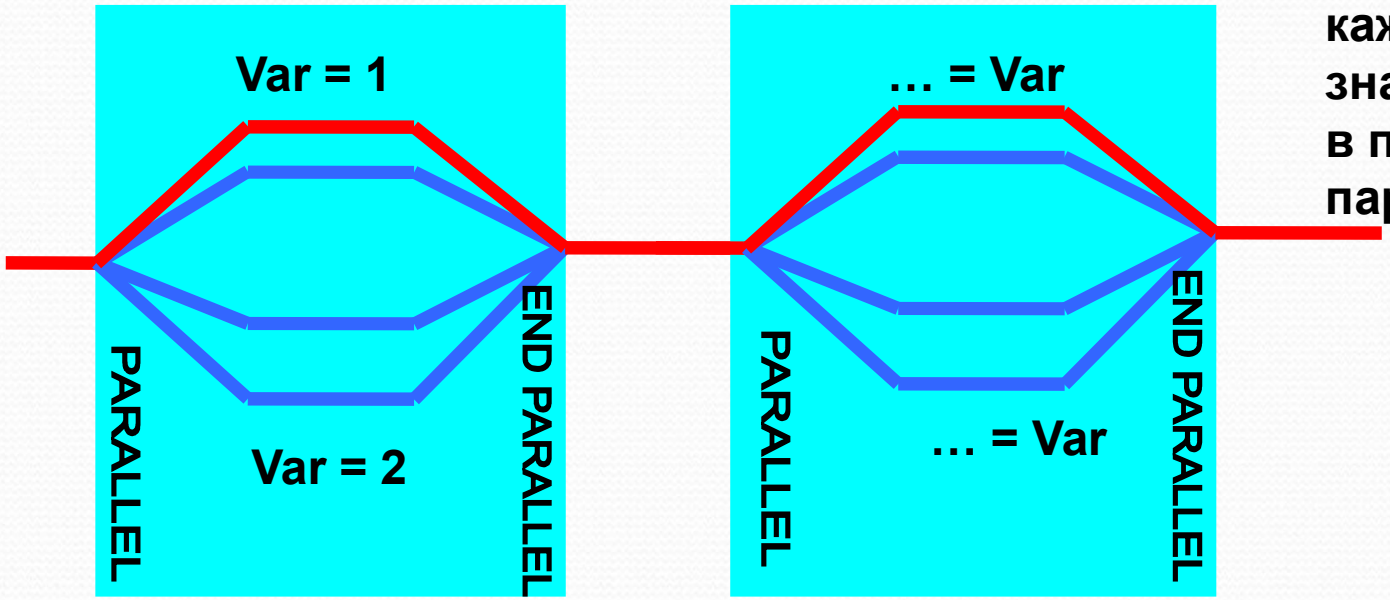
```
int i;
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];

}
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

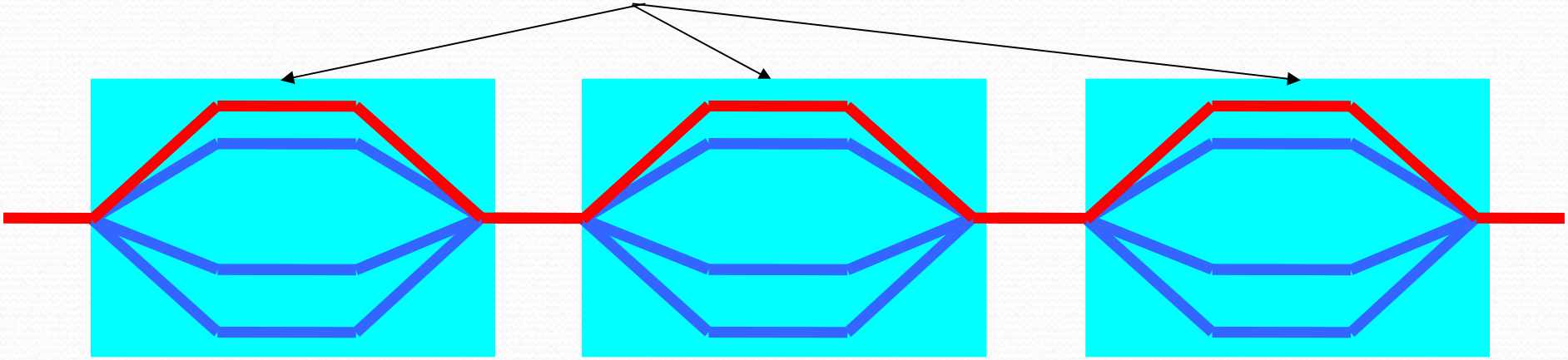
Меняет класс переменной по умолчанию:

- **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)

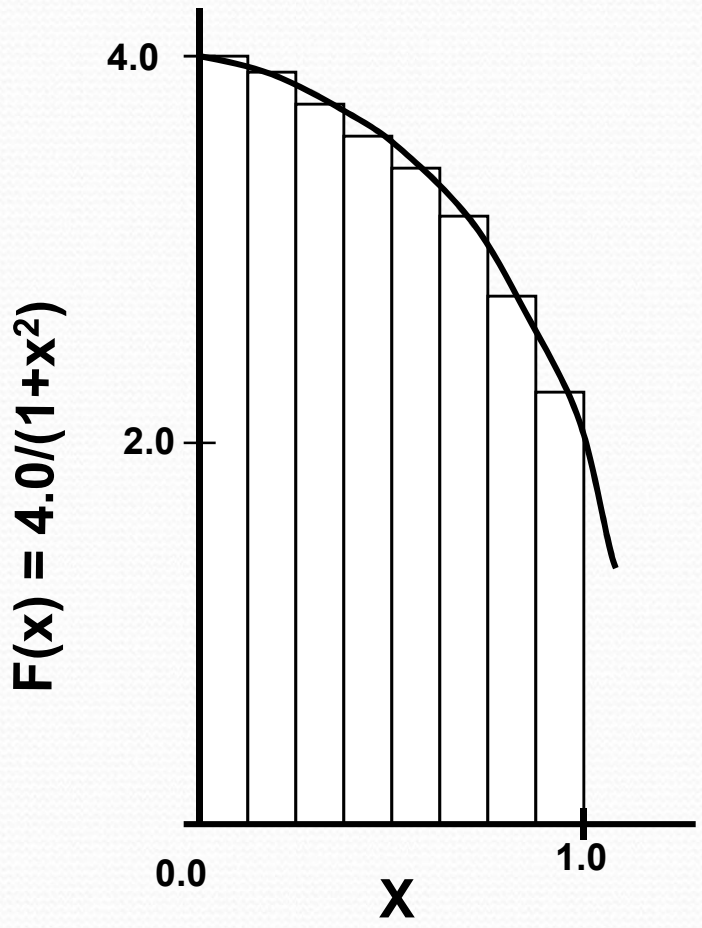


#pragma omp parallel *[клауза[[,] клауза] ...]*
структурный блок

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 Δx и высоту $F(x_i)$ в середине интервала

Вычисление числа π. Последовательная программа

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

Вычисление числа π. 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)
   {
      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;
}
```

Конфликт доступа к данным

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

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

Время	Thread 0	Thread 1
1	LOAD R1,sum	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R3,sum
4	STORE R1,sum	LOAD R4,val
5		ADD R3,R4
6		STORE R3,sum

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

Вычисление числа π. 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)
   {
      int id = omp_get_thread_num();
      int numt = omp_get_num_threads();
      for (i = id + 1; i <= n;  i=i+numt) {
            x = h * ((double)i - 0.5);
            #pragma omp critical
                  sum += (4.0 / (1.0 + x*x));
       }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π. 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 π на OpenMP. Клауза reduction

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

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

Клауза if

if(*scalar-expression*)

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

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

Клауза num_threads

num_threads(*integer-expression)*

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

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

Определение числа нитей в параллельной области

Число создаваемых нитей зависит от:

- клаузы if
- клаузы num_threads
- значений переменных, управляющих выполнением OpenMP-программы:
  - dyn-var (включение/отключение режима, в котором количество создаваемых нитей может изменяться динамически: **OMP_DYNAMIC, omp_set_dynamic()**)
  - nthreads-var (максимально возможное число нитей, создаваемых при входе в параллельную область: **OMP_NUM_THREADS, omp_set_num_threads()**)
  - thread-limit-var (максимально возможное число нитей, создаваемых для выполнения всей OpenMP-программы: **OMP_THREAD_LIMIT**)
  - nest-var (включение/отключение режима поддержки вложенного параллелизма:**OMP_NESTED, omp_set_nested()**)
  - max-active-level-var (максимально возможное количество вложенных параллельных областей: **OMP_MAX_ACTIVE_LEVELS, omp_set_max_active_levels()**)

Определение числа нитей в параллельной области

Пусть ***ThreadsBusy*** - *количество* OpenMP нитей, выполняемых в данный момент;

Пусть ***ActiveParRegions*** - *количество активных параллельных областей*;

if в директиве parallel существует клауза **if**

then *присвоить переменной* ***IfClauseValue*** *значение* выражения в клаузе **if**;

else ***IfClauseValue*** **=** ***true*****;**

if в директиве parallel существует клауза **num_threads**

then *присвоить переменной* ***ThreadsRequested*** *значение* выражения в **клаузе num_threads;**

else ***ThreadsRequested*** **=** ***nthreads-var*****;**

ThreadsAvailable **= (*thread-limit-var* - *ThreadsBusy* + 1);**

Определение числа нитей в параллельной области

if (*IfClauseValue* == *false*)
then number of threads = 1;
else if (*ActiveParRegions* >= 1) **and** (*nest-var* == *false*)
then number of threads = 1;
else if (*ActiveParRegions* == *max-active-levels-var*)
then number of threads = 1;
else if (*dyn-var* == *true*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)
then number of threads = [1 : *ThreadsRequested*];
else if (*dyn-var* == *true*) **and** (*ThreadsRequested* > *ThreadsAvailable*)
then number of threads = [1 : *ThreadsAvailable*];
else if (*dyn-var* == *false*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)
then number of threads = *ThreadsRequested*;
else if (*dyn-var* == *false*) **and** (*ThreadsRequested* > *ThreadsAvailable*)
then number of threads зависит от реализации компилятора;

Определение числа нитей в параллельной области

```
#include <stdio.h>
#include <omp.h>
int main (void)
{
     omp_set_nested(1);
     omp_set_max_active_levels(8);
     omp_set_dynamic(0);
     omp_set_num_threads(2);
     #pragma omp parallel
       {
          omp_set_num_threads(3);
          #pragma omp parallel
            {
              …
            }
       }
}
```