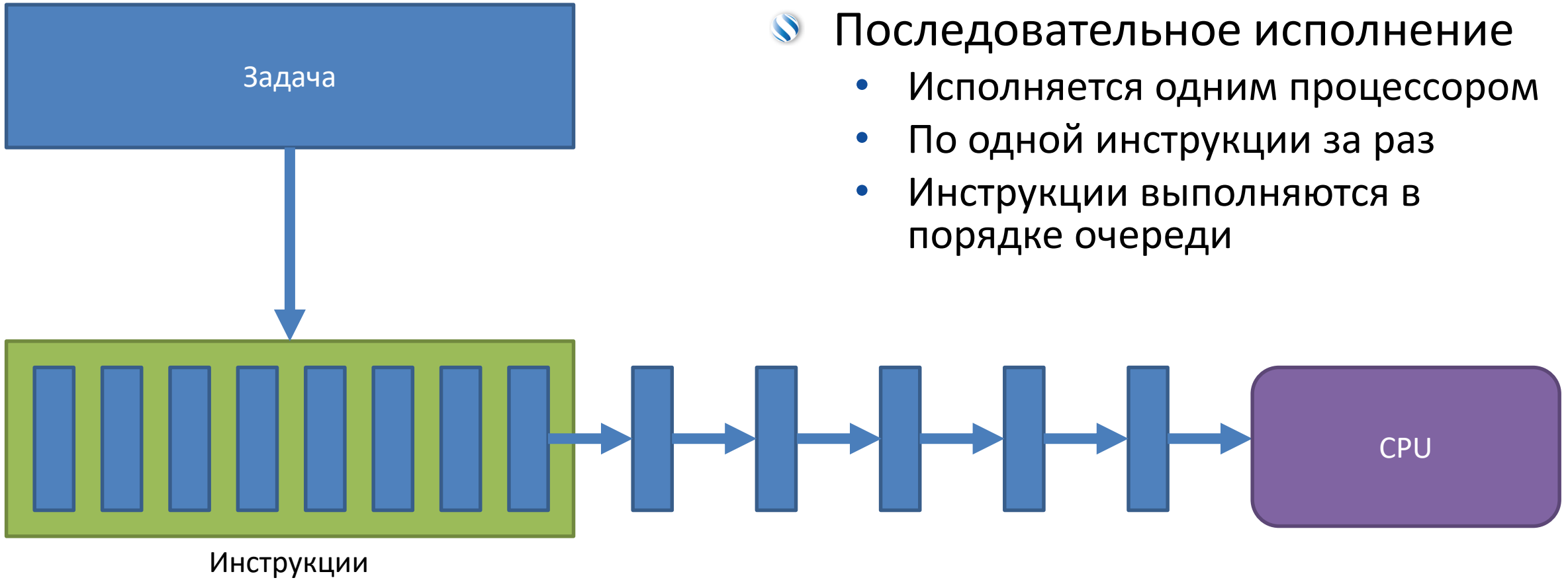
Applied Parallel Computing

parallel-computing.pro

Введение в OpenMP

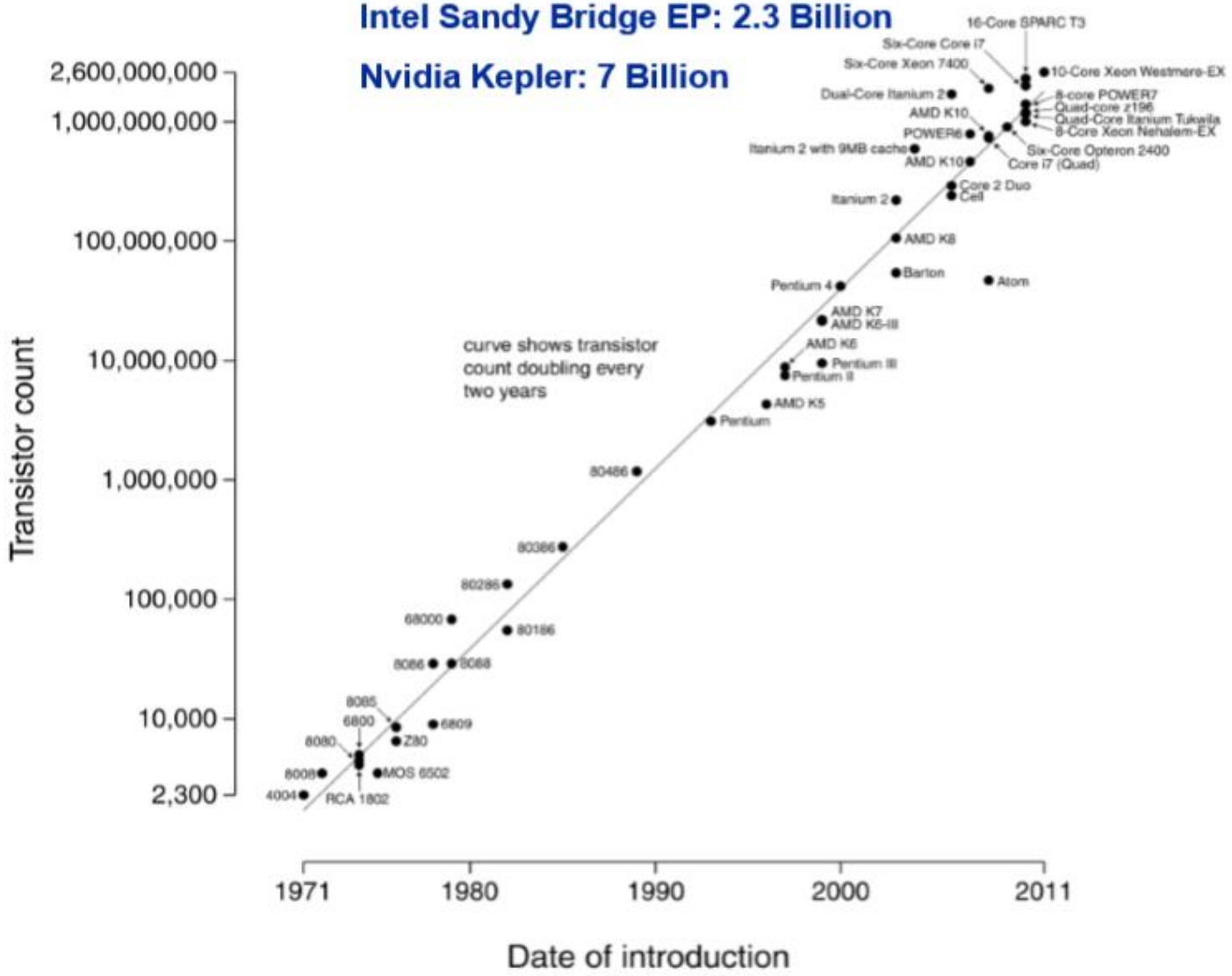
к.т.н. Алексей Ивахненко

Последовательное исполнение



- Последовательное исполнение
  - Исполняется одним процессором
  - По одной инструкции за раз
  - Инструкции выполняются в порядке очереди

Закон Мура


Intel Sandy Bridge EP: 2.3 Billion
Nvidia Kepler: 7 Billion
curve shows transistor count doubling every two years
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
Date of introduction
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
AMD K10
8-Core Xeon Nehalem-EX
POWER6
Six-Core Opteron 2400
Itanium 2 with 9MB cache
Core i7 (Quad)
Core 2 Duo
Itanium 2
Cell
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

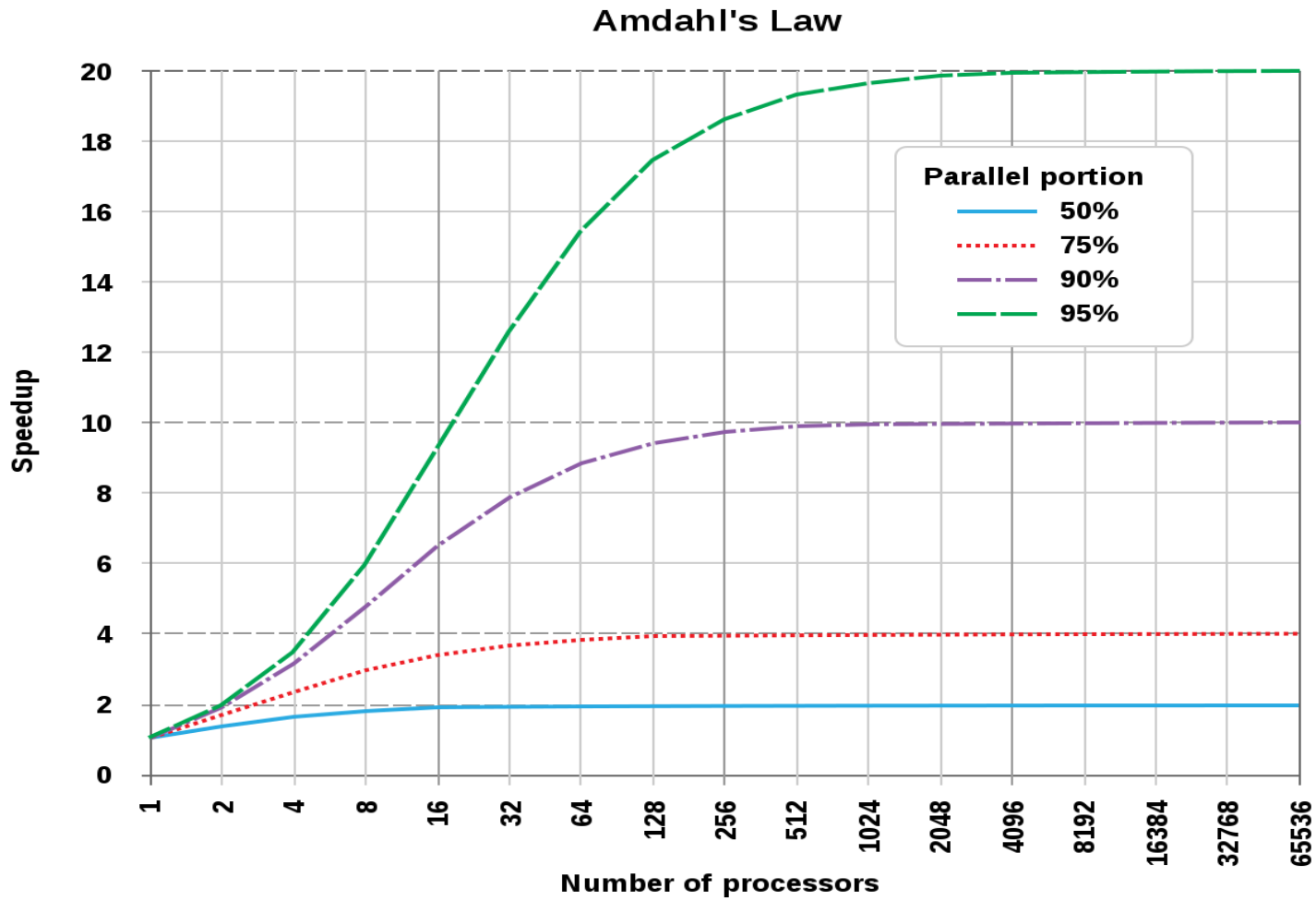
Закон Амдаля

$S(p) \leq 1/(f + (1 - f)/p)$

p – количество процессоров

f – доля программы, которая должна быть выполнена последовательно

S - ускорение



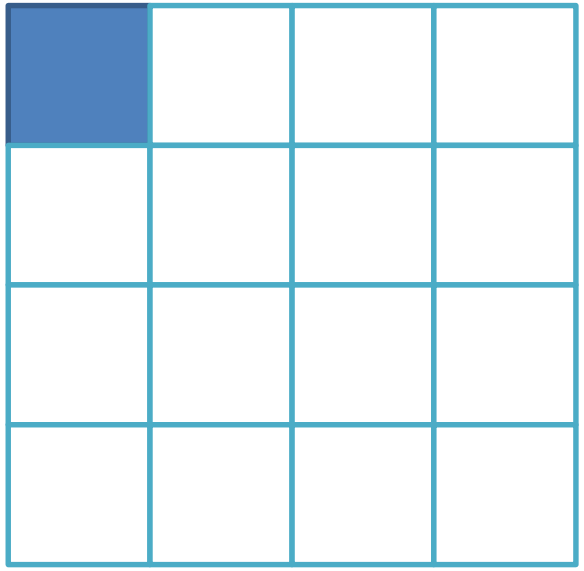
Последовательное исполнение

$C_0 = A_0 + B_0$

$C_1 = A_1 + B_1$

. . . . . .

$C_{n-1} = A_{n-1} + B_{n-1}$

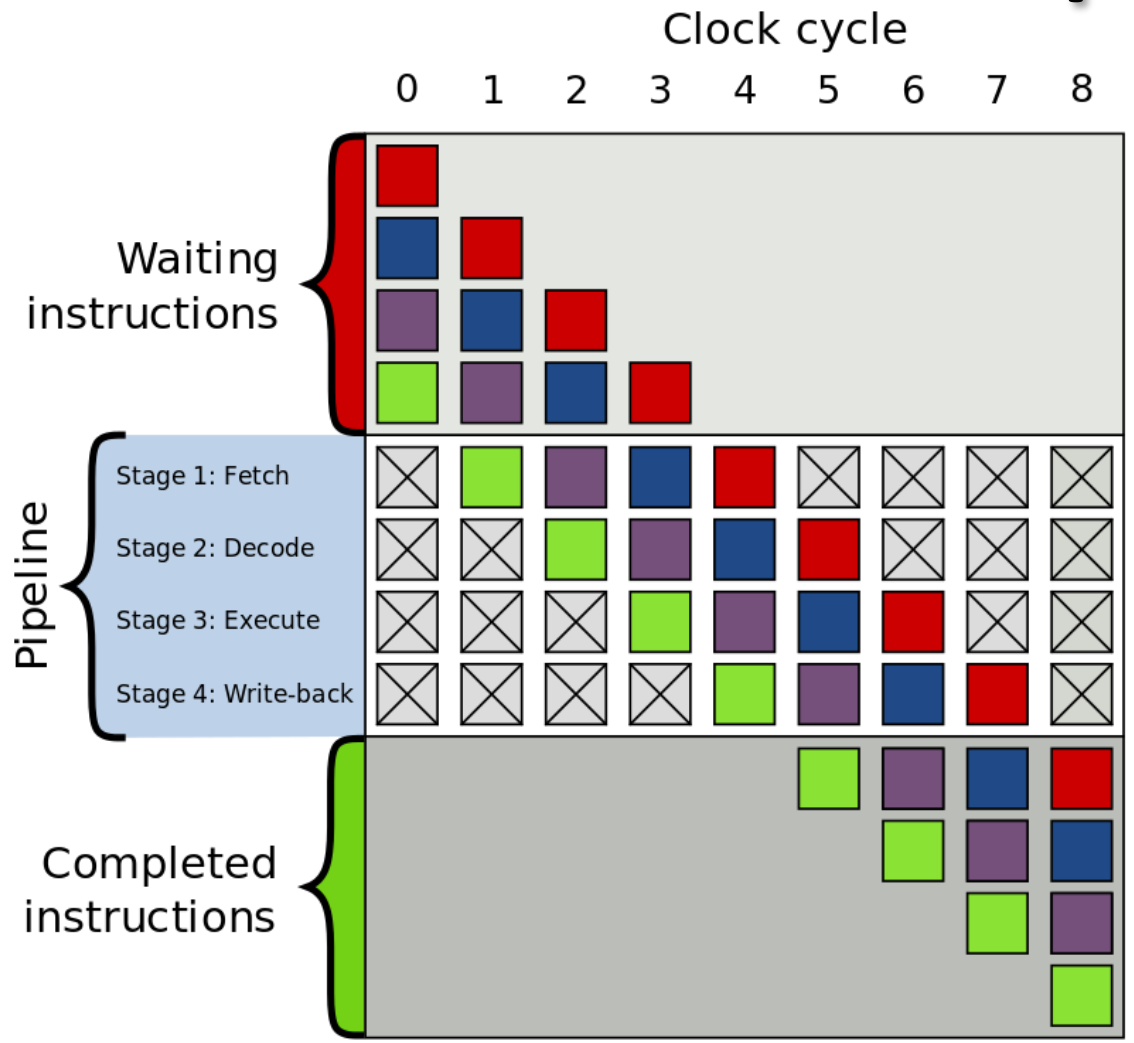


Последовательное исполнение

```
multiply_and_add(const float* a, const float* b, const float* c, float* d)
{
for(int i=0; i<8; i++)
 {
   d[i] = a[i] * b[i];
   d[i] = d[i] + c[i];
 }
}
```

Конвейер

- Общий конвейер с четырьмя стадиями работы:
  - Получение
  - Раскодирование
  - Выполнение
  - Запись результата

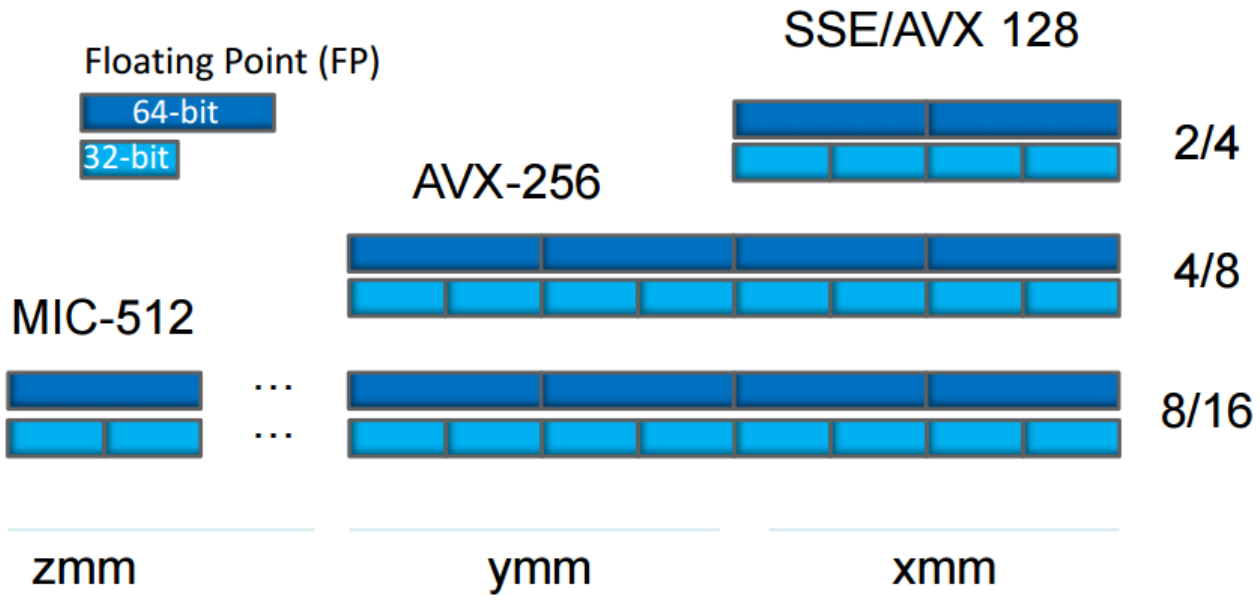


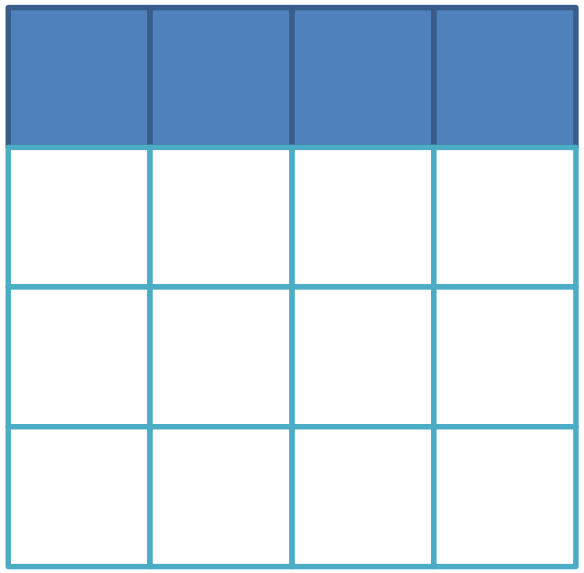
ILP

```
multiply_and_add(const float* a, const float* b, const float* c, float* d)
{
 #pragma unroll 4
 for(int i=0; i<8; i++)
 {
   d[i] = a[i] * b[i];
   d[i] = d[i] + c[i];
 }
}

__m256 multiply_and_add(__m256 a, __m256 b, __m256 c)
{
 return _mm256_fmadd_ps(a, b, c);
}
```

Векторные инструкции





Векторные инструкции

Векторизация

```
multiply_and_add(const float* a, const float* b, const float* c, float* d)
{
 for(int i=0; i<8; i++)
 {
   d[i] = a[i] * b[i];
   d[i] = d[i] + c[i];
 }
}
```

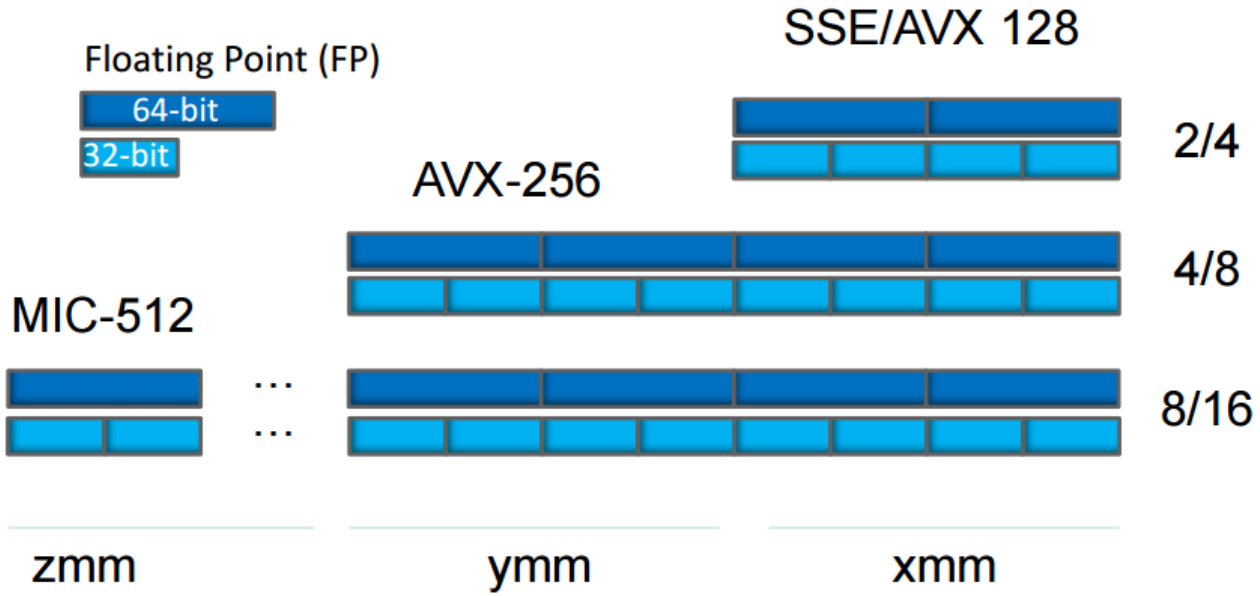
```
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c)
{
 return _mm256_fmadd_ps(a, b, c);
}
```

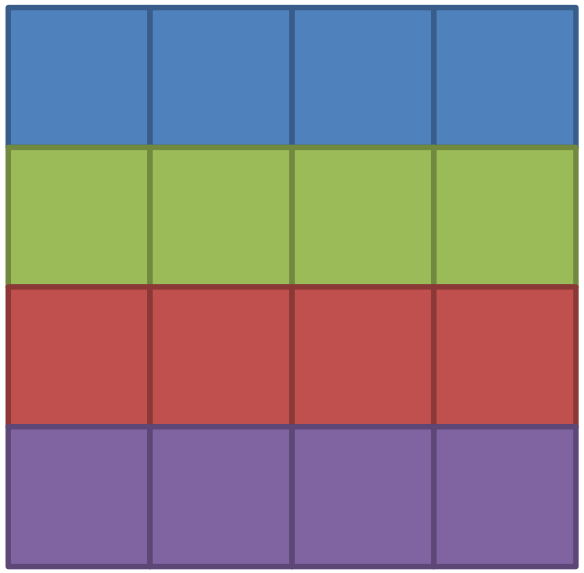
AVX/AVX2

- В общем виде инструкция AVX/AVX2 называется следующим образом:
  - _mm<bit_width>_<name>_<data_type>
- <data_type>:
  - ps - vectors contain floats (ps stands for packed single-precision)
  - pd - vectors contain doubles (pd stands for packed double-precision)
  - epi8/epi16/epi32/epi64 - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers
  - epu8/epu16/epu32/epu64 - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers
  - si128/si256 - unspecified 128-bit vector or 256-bit vector
  - m128/m128i/m128d/m256/m256i/m256d - identifies input vector types when they're different than the type of the returned vector

Data Type	Description
__m128	128-bit vector containing 4 floats
__m128d	128-bit vector containing 2 doubles
__m128i	128-bit vector containing integers
__m256	256-bit vector containing 8 floats
__m256d	256-bit vector containing 4 doubles
__m256i	256-bit vector containing integers

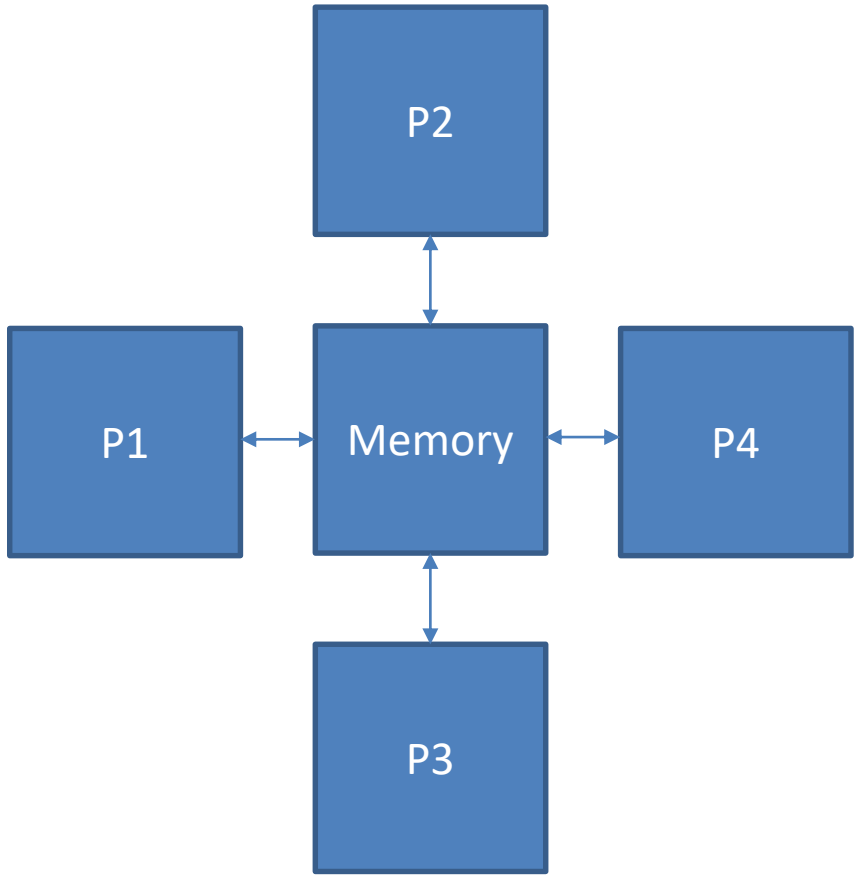
Векторные инструкции и параллельное исполнение





Общая память

- Все процессоры имеют доступ к памяти, как глобальному адресному пространству
- Процессоры работают независимо, но делят ресурсы памяти



Общая память

- Преимущества
  - Глобальное адресное пространство – упрощает программирование
  - Позволяет распараллеливать код по частям
  - Унифицирует и ускоряет обмен данными между CPU
- Недостатки
  - Память не масштабируется вместе с количеством процессоров
  - Накладные расходы на организацию общего доступа к памяти
  - Программист сам отвечает за синхронизацию доступов к памяти
  - Soaring expense of internal network.

План

- OpenMP.
- Программная модель и модель исполнения.
- Параллельные регионы: teams и threads.
- Синтаксис.
- Данные: private и shared переменные.
- Переменные среды.
- Директивы распределения работы.
- Определение индексов.

Программная модель

- Множество архитектур
  - AMD, Intel, IBM, Nvidia и др.
- Множество спецификаций
  - версии 1.0 (Fortran ’97, C ’98), 1.1 и 2.0 (Fortran ’00, C/C++ ’02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015)
- Состоит из директив, функций и переменных среды
- Требует поддержки компилятора
  - -fopenmp (gcc) –openmp (icc)

Модель исполнения

- OpenMP потоки
  - Fork / join
    - ✓ Программа начинается с 1-го потока (initial)
    - ✓ Разветвляется в зависимости от указаний программиста
  - Может исполняться на нескольких устройствах
    - ✓ Контролируется host
    - ✓ Можно передать часть программы ускорителю (device)
    - ✓ Потоки не мигрируют между устройствами

Hello World!

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

int main (int argc, char *argv[])
{
#pragma omp parallel num_threads(2)
  {
        int nthreads, tid;
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        printf("Hello World from thread = %d\n of %d\n", tid, nthreads);
  }
}
```

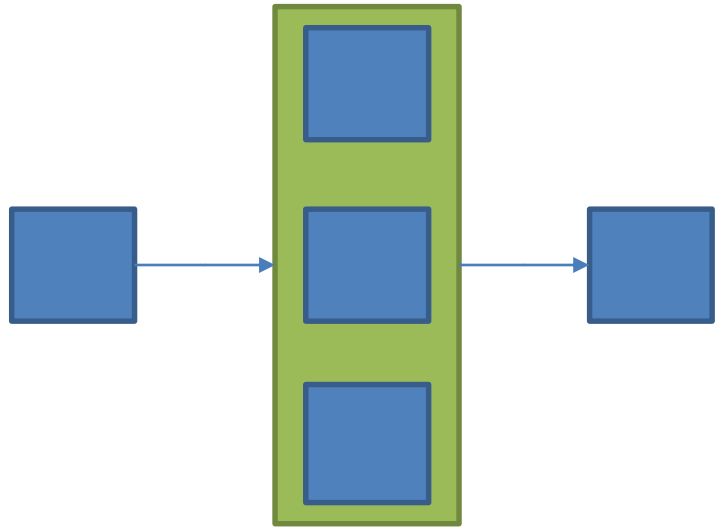
Синтаксис

- **Fortran**

```
!$omp directive [clause [, clause] …]
        structured block
!$omp end directive
```

- **C**

```
#pragma omp directive [clause [, clause] …]
        structured block
```



Директивы

- Parallel
- Sections
- For
- Single
- Simd
- Declare simd
- Task
- Taskloop
- Taskyield

Директивы

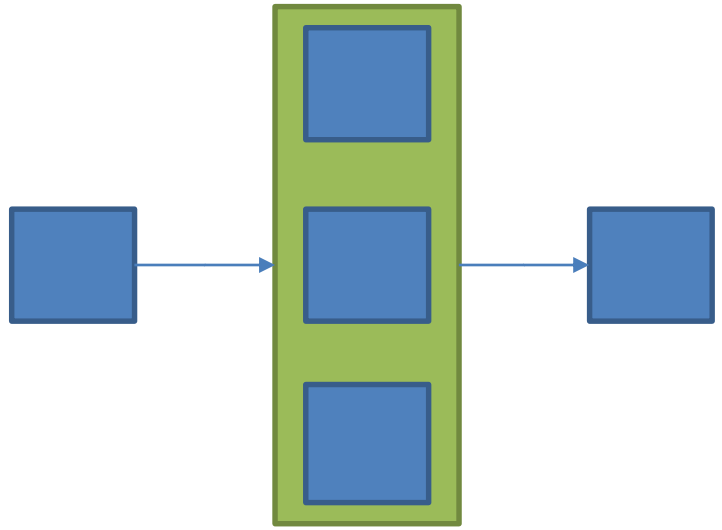
- Target data
- Target enter data
- Target exit data
- Target
- Target update
- Declare target
- Teams
- Distribute

Директивы

- Master
- Critical
- Barrier
- Taskwait
- Taskgroup
- Atomic
- Flush
- Ordered
- Cancel
- Cancellation point
- Threadprivate
- Declare reduction

Parallel

```
#pragma omp parallel
  {
    // Code inside this region runs in parallel.
    printf("Hello!\n");
  }
```



Clauses

- if([parallel :] scalar-expression
- num_threads(integer-expression)
- default(shared | none)
- private(list)
- firstprivate(list)
- shared(list)
- copyin(list)
- reduction(reduction-identifier: list)
- proc_bind(master | close | spread)

For

```
#pragma omp for [clause[ [, ]clause] ...]
        for-loops

clause:
        private(list)
        firstprivate(list)
        lastprivate(list)
        linear(list [ : linear-step])
        reduction(reduction-identifier : list)
        schedule( [modifier [, modifier] : ] kind[, chunk_size])
        collapse(n)
        ordered[ (n) ]
        nowait
```

Schedule kind

- kind:
  - static: Iterations are divided into chunks of size chunk_size and assigned to threads in the team in round-robin fashion in order of thread number.
  - dynamic: Each thread executes a chunk of iterations then requests another chunk until none remain.
  - guided: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned.
  - auto: The decision regarding scheduling is delegated to the compiler and/or runtime system.
  - runtime: The schedule and chunk size are taken from the run-sched-var ICV.

Schedule modifier

- modifier:
  - monotonic: Each thread executes the chunks that it is assigned in increasing logical iteration order.
  - nonmonotonic: Chunks are assigned to threads in any order and the behavior of an application that depends on execution order of the chunks is unspecified.
  - simd: Ignored when the loop is not associated with a SIMD construct, otherwise the new_chunk_size for all except the first and last chunks is chunk_size/ simd_width * simd_width where simd_width is an implementation-defined value.

OpenMP Pi calculation

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    // Get the N argument value.
    size_t n = (size_t)strtoul(argv[1], NULL, 0);
    double w = 1.0 / n;
    double sum=0.0;
    #pragma omp parallel for schedule (static) reduction (+:sum)
    for (size_t i =0; i < n; ++i)
    {
        double x = w * ((double)i + 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    double pi=w* sum;
    printf("sum = %24.16g\n", pi);
    return 0;
}
```

Teams vs threads

- While the parallel construct creates a team of threads, the teams construct creates a league of teams.
- This directive can be only used directly inside a target construct.
  - The optional attribute num_teams can be used to specify the maximum number of teams created.
  - The actual number of teams may be smaller than this number. The master thread of each team will execute the code inside that team.

```
#include <stdio.h>

int main(void)
{
  #pragma omp target teams
  {
    printf("Hello World!\n");
  }
  return 0;
}
```

Shared vs Private

```
#pragma omp parallel for private(temp) shared(n,a,b,c)
{
   for(i=1; i<=n; i++){
      temp = 2.0*a[i];
      a[i] = temp;
      b[i] = c[i]/temp;
   }
}
```

Shared vs Private

- The default context of a variable is determined by the following rules:
  - Variables with static storage duration are shared.
  - Dynamically allocated objects are shared.
  - Variables with automatic storage duration that are declared in a parallel region are private.
  - Variables in heap allocated memory are shared. There can be only one shared heap.
  - All variables defined outside a parallel construct become shared when the parallel region is encountered.
  - Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
  - Memory allocated within a parallel loop by the alloca function persists only for the duration of one iteration of that loop, and is private for each thread.

Shared vs Private

```
int E1;                          /* shared static     */

void main (argvc,...) {          /* argvc is shared   */
   int i;                         /* shared automatic  */

void *p = malloc(...);          /* memory allocated by malloc   */
                                   /* is accessible by all threads */
                                   /* and cannot be privatized     */



int E2;                          /*shared static */

void foo (int x) {               /* x is private for the parallel */
                                   /* region it was called from     */

int c;                          /* the same */
 ...
```

Shared vs Private

```
#pragma omp parallel firstprivate (p)
   {
     int b;                    /* private automatic  */
     static int s;             /* shared static      */

     #pragma omp for
     for (i =0;...) {
       b = 1;                  /* b is still private here !    */
       foo (i);                /* i is private here because it */
                               /* is an iteration variable     */
      }


#pragma omp parallel
     {
       b = 1;                  /* b is shared here because it  */
                               /* is another parallel region   */
     }
   }
 }
…
}
```

Sections

```
#pragma omp sections
 {
   { Work1(); }
   #pragma omp section
   { Work2();
     Work3(); }
   #pragma omp section
   { Work4(); }
 }
```

SIMD

```
float a[8], b[8];
...
#pragma omp simd
for(int n=0; n<8; ++n) a[n] += b[n];

#pragma omp declare simd aligned(a,b:16)
 void add_arrays(float *__restrict__ a, float *__restrict__ b)
 {
   #pragma omp simd aligned(a,b:16)
   for(int n=0; n<8; ++n) a[n] += b[n];
 }
```

Environment variables

- The environment variables are as follows:
  - OMP_SCHEDULE sets the run-sched-var ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
  - OMP_NUM_THREADS sets the nthreads-var ICV that specifies the number of threads to use for parallel regions.
  - OMP_DYNAMIC sets the dyn-var ICV that specifies the dynamic adjustment of threads to use for parallel regions.
  - OMP_PROC_BIND sets the bind-var ICV that controls the OpenMP thread affinity policy.
  - OMP_PLACES sets the place-partition-var ICV that defines the OpenMP places that are available to the execution environment.
  - OMP_NESTED sets the nest-var ICV that enables or disables nested parallelism.
  - OMP_STACKSIZE sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation.

Environment variables

- OMP_WAIT_POLICY sets the wait-policy-var ICV that controls the desired behavior of waiting threads.
- OMP_MAX_ACTIVE_LEVELS sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.
- OMP_THREAD_LIMIT sets the thread-limit-var ICV that controls the maximum number of threads participating in a contention group.
- OMP_CANCELLATION sets the cancel-var ICV that enables or disables cancellation.
- OMP_DISPLAY_ENV instructs the runtime to display the OpenMP version number and the initial values of the ICVs, once, during initialization of the runtime.
- OMP_DEFAULT_DEVICE sets the default-device-var ICV that controls the default device number.
- OMP_MAX_TASK_PRIORITY sets the max-task-priority-var ICV that specifies the maximum value that can be specified in the priority clause of the task construct.

Индексы

- `int omp_get_num_threads( );`
  - Количество потоков
- `int omp_get_thread_num( );`
  - Номер потока

План

- Атомарные операции и критические секции.
- OpenMP tasks и sections.
- Nesting и Binding.
- Редукция.
- Измерение времени.
- Примеры.

Atomics

```
#pragma omp atomic
      counter += value;
```

- Clauses
  - Read

    ```
    #pragma omp atomic read
    var = x;
    ```
  - Write

    ```
    #pragma omp atomic write
    x = expr;
    ```

Atomics

- Clauses
  - Update

```
#pragma omp atomic update // The word "update" is optional
// One of these:
++x; --x; x++; x--;
x += expr;  x -= expr;  x *= expr;   x /= expr;  x &= expr;
x = x+expr; x = x-expr; x = x*expr;  x = x/expr; x = x&expr;
x = expr+x; x = expr-x; x = expr*x;  x = expr/x; x = expr&x;
x |= expr;  x ^= expr;  x <<= expr;  x >>= expr;
x = x|expr; x = x^expr; x = x<<expr; x = x>>expr;
x = expr|x; x = expr^x; x = expr<<x; x = expr>>x;
```

  - Capture

```
#pragma omp atomic capture
// One of these:
var = x++;  /* Or any other of the update expressions listed above */
{ var = x; x++; /* Or any other of of the update expressions listed above */ }
{ x++; /* Or any other of of the update expressions listed above */; var = x; }
{ var = x; x = expr; }
```

Critical

```
#pragma omp critical(dataupdate)
 {
   datastructure.reorganize();
 }
 ...
 #pragma omp critical(dataupdate)
 {
   datastructure.reorganize_again();
 }
```

- The critical construct restricts the execution of the associated statement / block to a single thread at time.
- The critical construct may optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.
- If the name is omitted, a default name is assumed.

Reduction

```
int factorial(int number)
 {
   int fac = 1;
   #pragma omp parallel for reduction(*:fac)
   for(int n=2; n<=number; ++n)
     fac *= n;
   return fac;
 }
```

Reduction

- reduction(operator:list)
  - +, -, |, ^, || 0
  - *, && 1
  - & ~0
  - min largest representable number
  - max smallest representable number

Custom reduction

```
#pragma omp declare reduction(name:type:expression)

#pragma omp declare reduction(name:type:expression) initializer(expression)
```

- The name is the name you want to give to the reduction method.
- The type is the type of your reduction result.
- Within the reduction expression, the special variables omp_in and omp_out are implicitly declared, and they stand for the input and output expressions respectively.
- Within the initializer expression, the special variable omp_priv is implicitly declared and stands for the initial value of the reduction result.

Timing

- The omp_get_wtime routine returns elapsed wall clock time in seconds.

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

- The omp_get_wtime routine returns a value equal to the elapsed wall clock time in seconds since some “time in the past”. The actual “time in the past” is arbitrary, but it is guaranteed not to change during the execution of the application program. The time returned is a “per-thread time”, so it is not required to be globally consistent across all threads participating in an application.

Timing

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

Полезные ссылки и использованные источники

- https://bisqwit.iki.fi/story/howto/openmp/
- https://software.intel.com/en-us/node/692395
- http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf
- http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
- https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.cbcpx01/cuppvars.htm