академия больших данных

mail.ru group

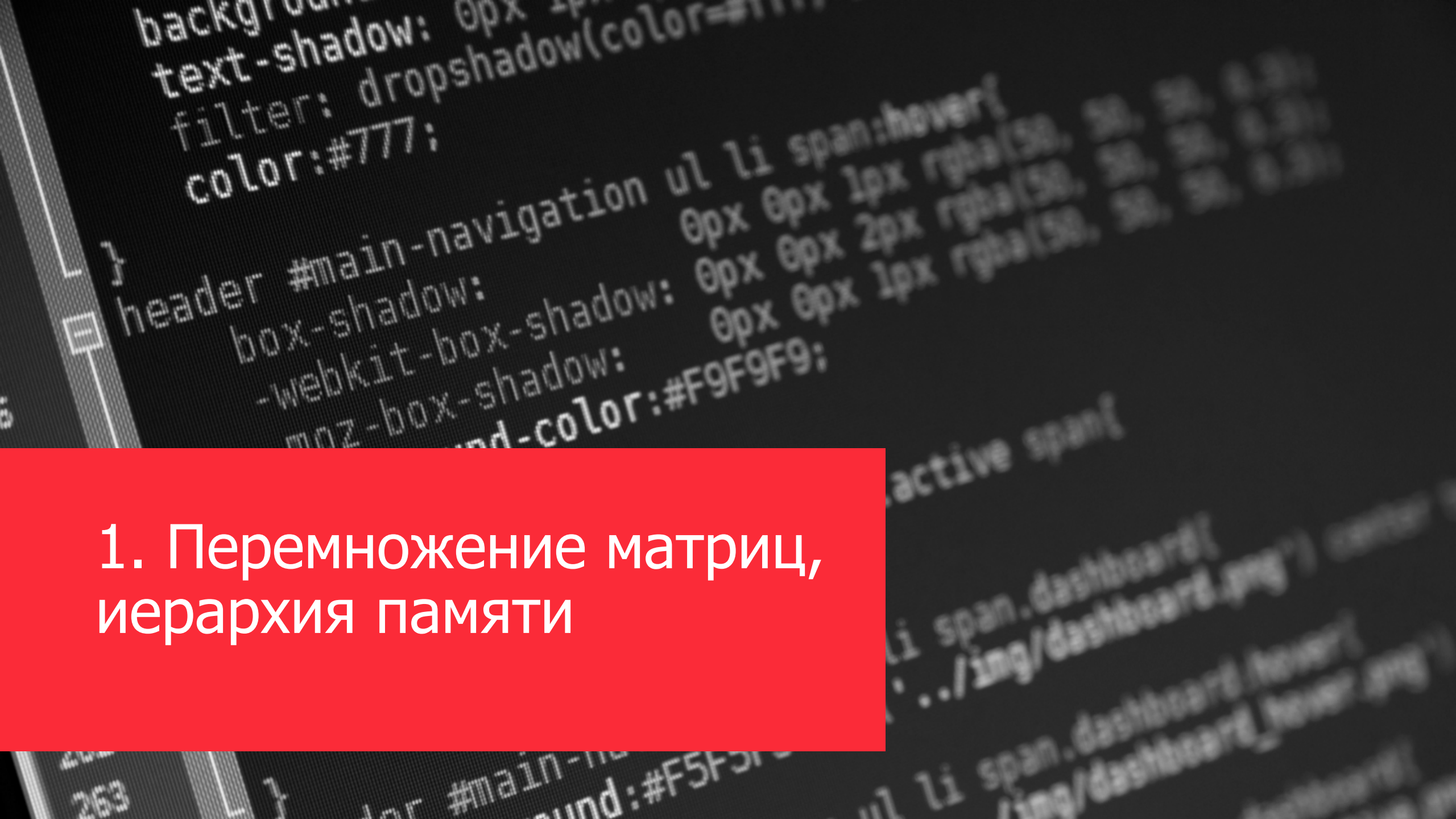
Высокопроизводительные вычисления. Лекция 3. OpenMP

Рыкованов Сергей

доцент Сколковского института науки и технологий

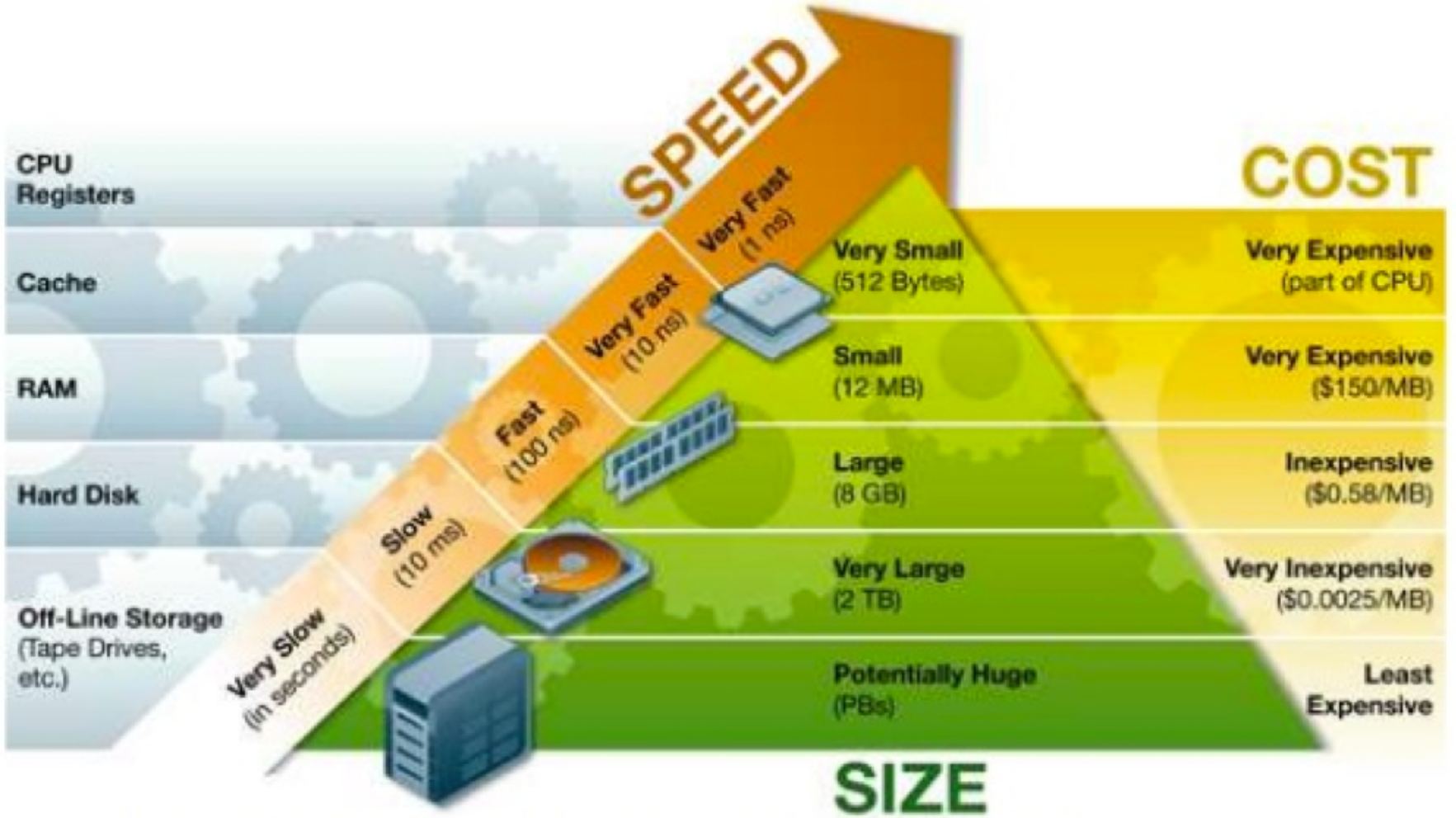
Матвеев Сергей

научный сотрудник Сколковского института науки и технологий



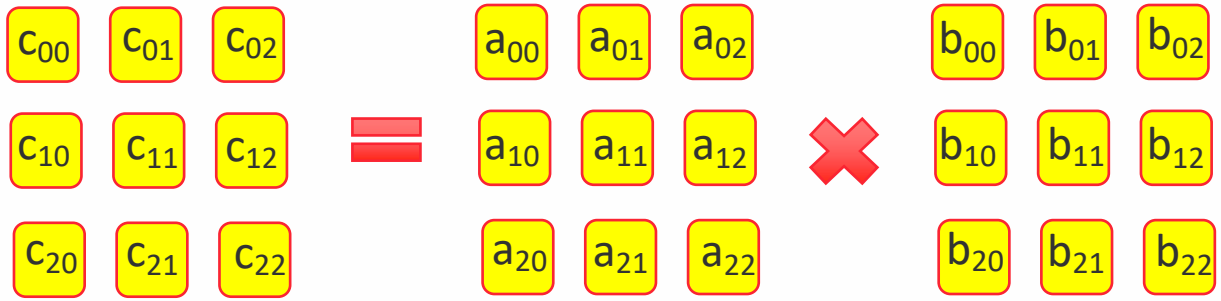
1. Перемножение матриц, иерархия памяти

Иерархия памяти в компьютере



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

«Наивное» перемножение матриц



NxN matrices: A, B, C
N >> 1

C = A * B

$$c_{ij}=\sum_{n=0}^{n=N-1} a_{in}*b_{nj}$$

```
for i from 0 to N-1:
      for j from 0 to N-1:
            for n from 0 to N-1:

      C[i][j]+=A[i][n]*B[n][j]
```

https://bit.ly/339RAhp

Основные элементы кода: аллокация памяти и адресация

```
double *A, *B, *C;
A = (double *) malloc(N * N * sizeof(double));
B = (double *) malloc(N * N * sizeof(double));
C = (double *) malloc(N * N * sizeof(double));
free(A); free(B); free(C);
```

матрица NxN

```
void RandomMatrix(double * A, size_t N)
{
    srand(time(NULL));
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            A[i * N + j] = rand() / RAND_MAX;
        }
    }
}
```

адресация к элементу A_{ij}

Основные элементы кода: matmul и измерение времени

matmul:

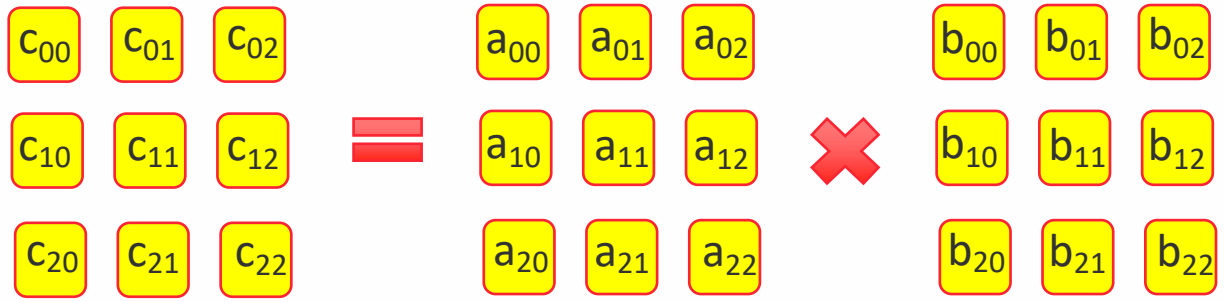
```
for (i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            for(k = 0; k < N; k++)
                C[i * N + j] = C[i * N + j] + A[i * N + k] * B[k * N + j];
```

```
struct timeval start, end;
double r_time = 0.0;
gettimeofday(&start, NULL);
... работа
... еще работа
gettimeofday(&end, NULL);
r_time = end.tv_sec - start.tv_sec + ((double)(end.tv_usec - start.tv_usec)) / 1000000;
```

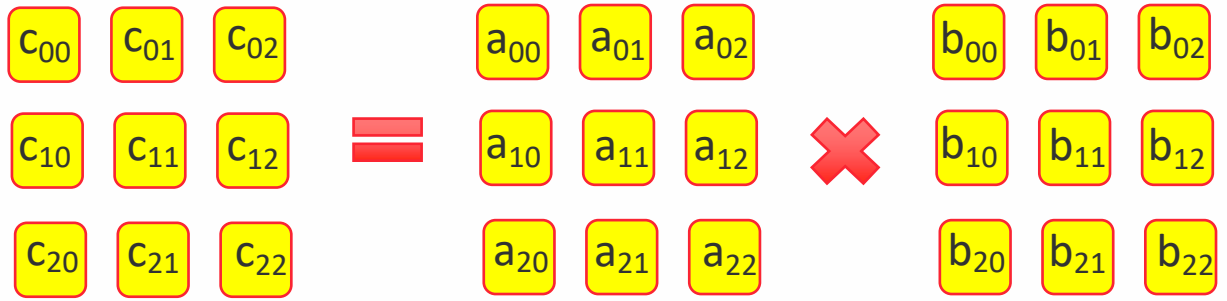
секунды

микросекунды

другие опции с разным разрешением: clock(); clock_gettime()

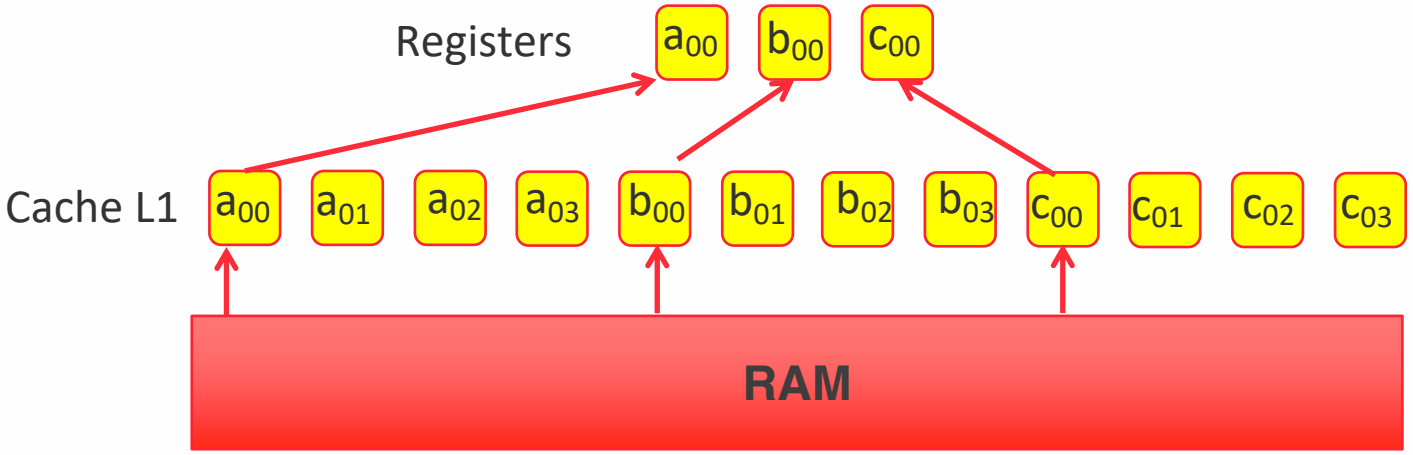


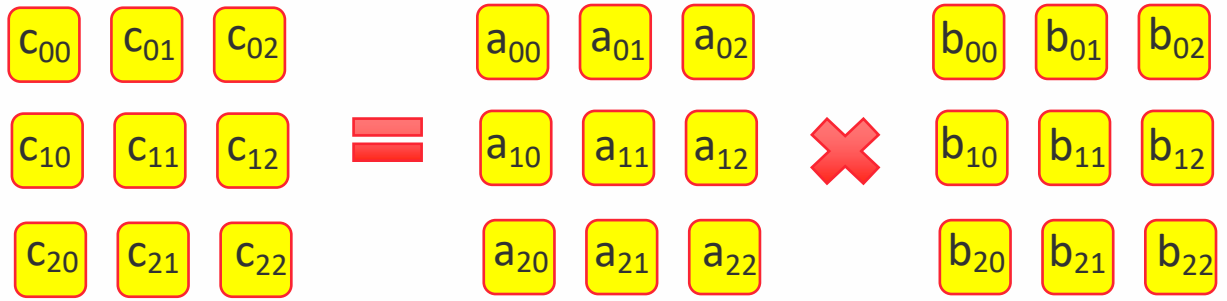

```
for i from 0 to N-1:
       for j from 0 to N-1:
              for n from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```




```
for i from 0 to N-1:
       for j from 0 to N-1:
              for n from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```

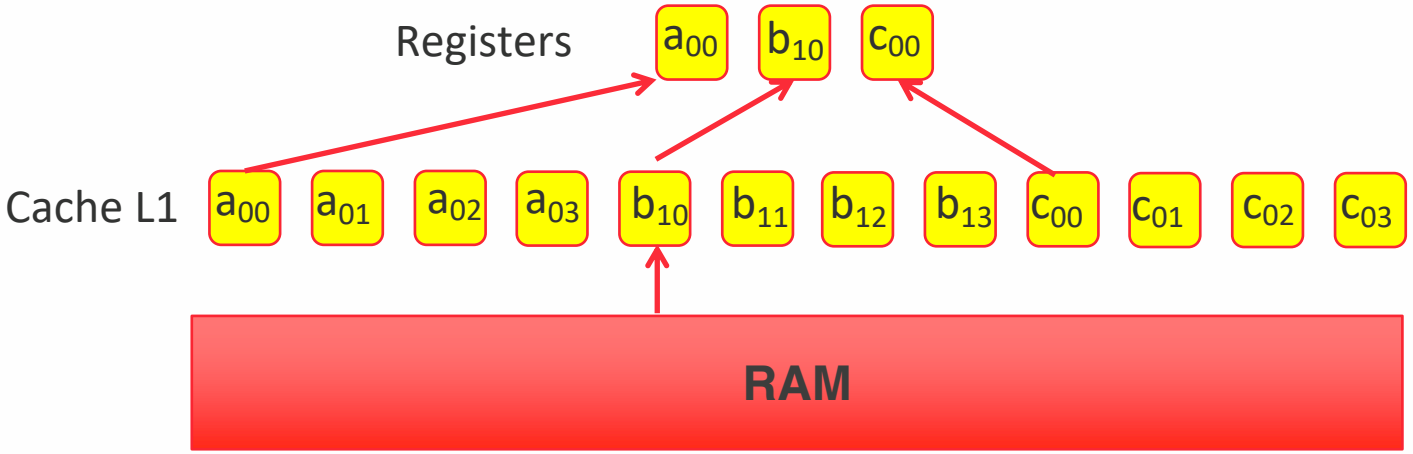
```
i=0
j=0
n=0
C[0][0]+=A[0][0]*B[0][0]
```

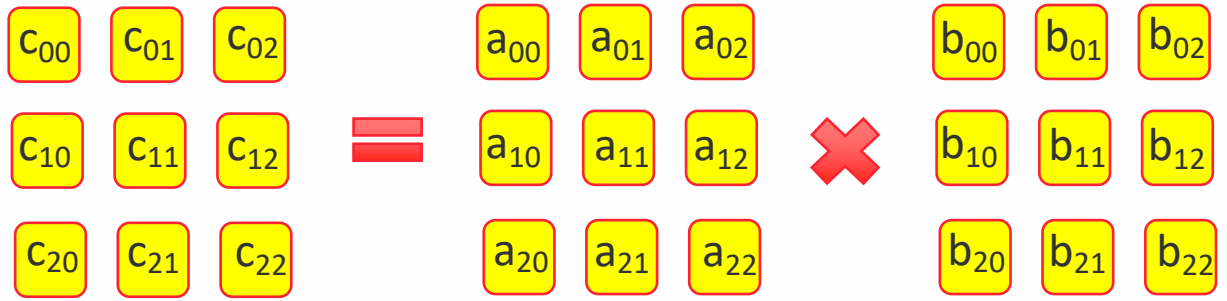





```
for i from 0 to N-1:
       for j from 0 to N-1:
              for n from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```

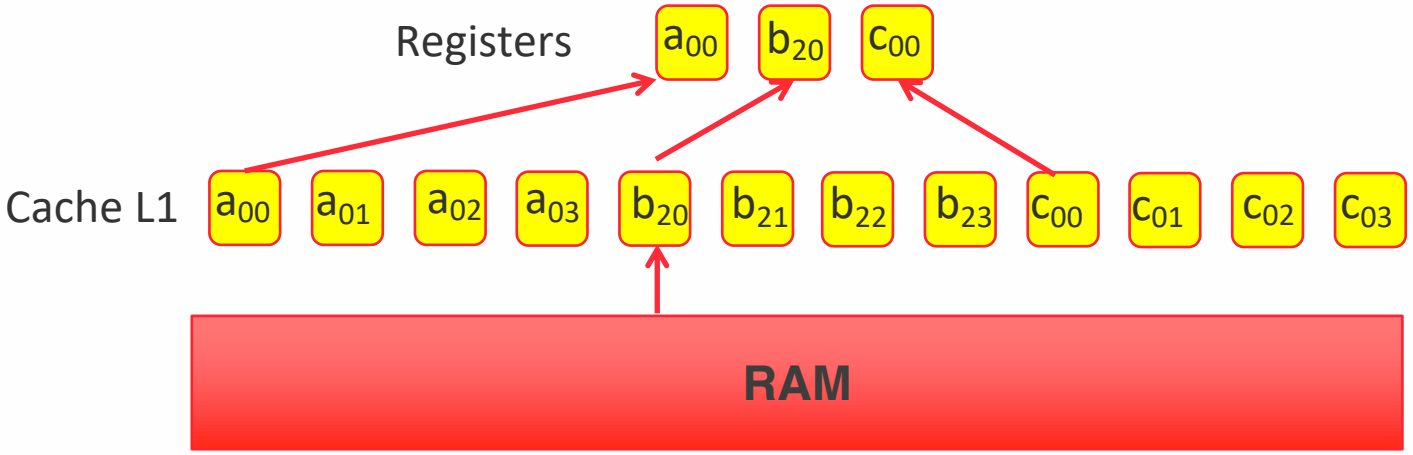
```
i=0
j=0
n=1
C[0][0]+=A[0][1]*B[1][0]
```

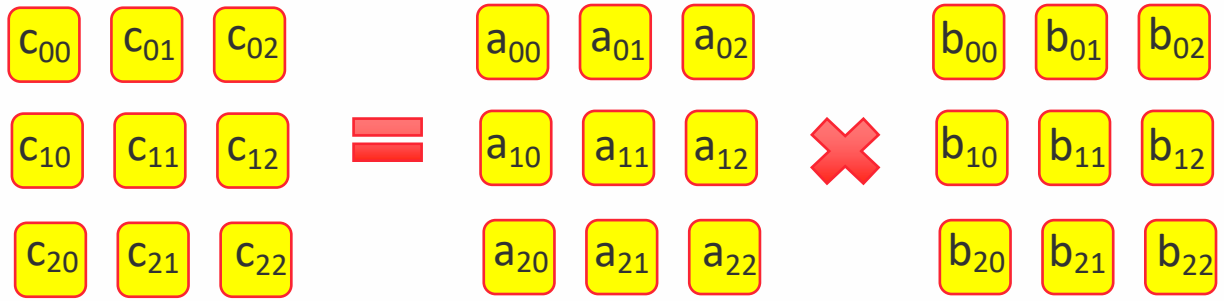




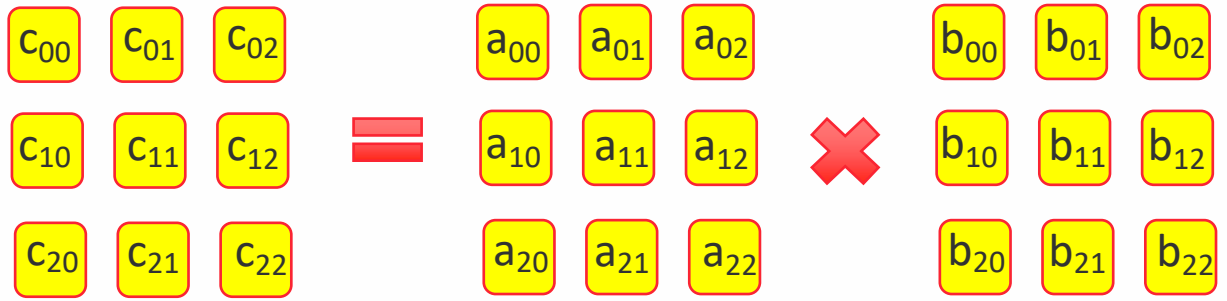

```
for i from 0 to N-1:
       for j from 0 to N-1:
              for n from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```

```
i=0
j=0
n=2
C[0][0]+=A[0][2]*B[2][0]
```



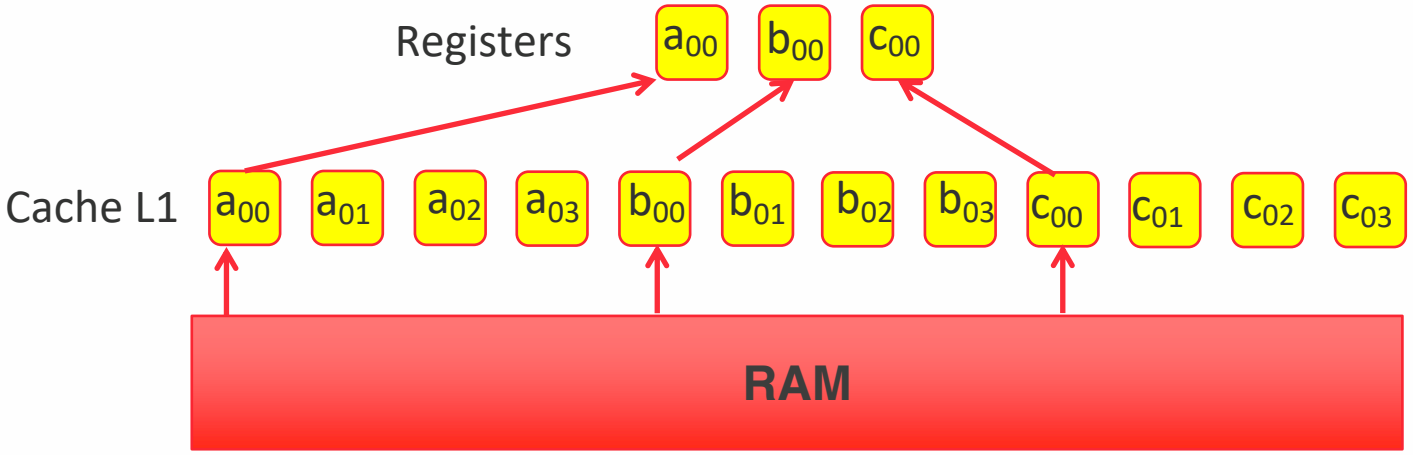


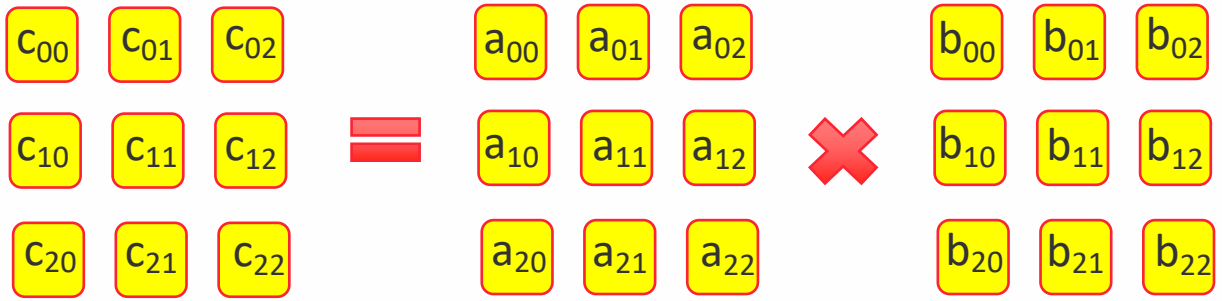

```
for n from 0 to N-1:
       for i from 0 to N-1:
              for j from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```




```
for n from 0 to N-1:
       for i from 0 to N-1:
              for j from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```

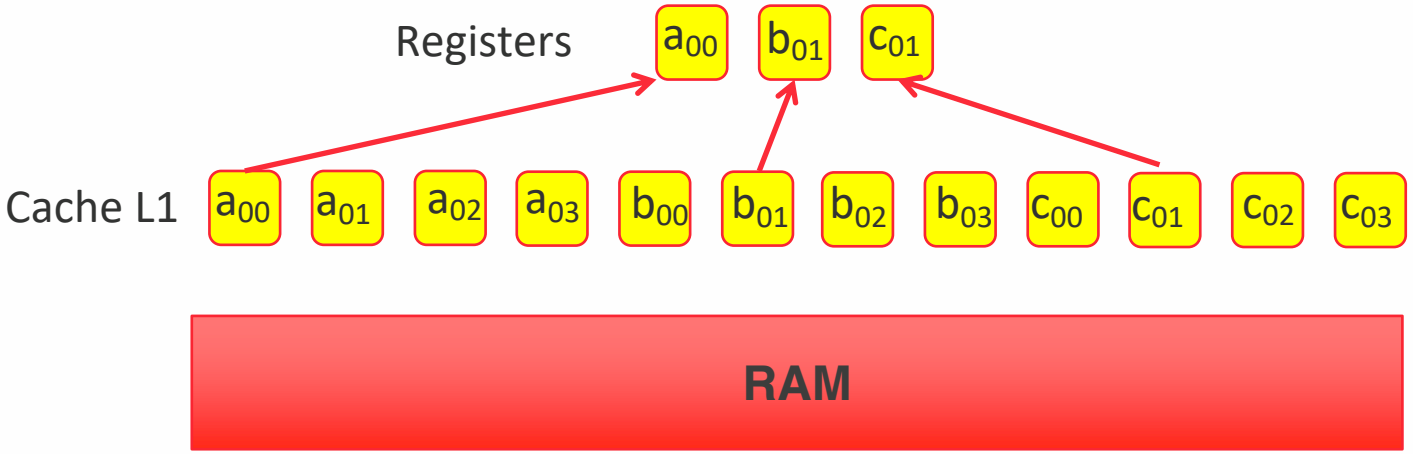
```
n=0
i=0
j=0
C[0][0]+=A[0][0]*B[0][0]
```

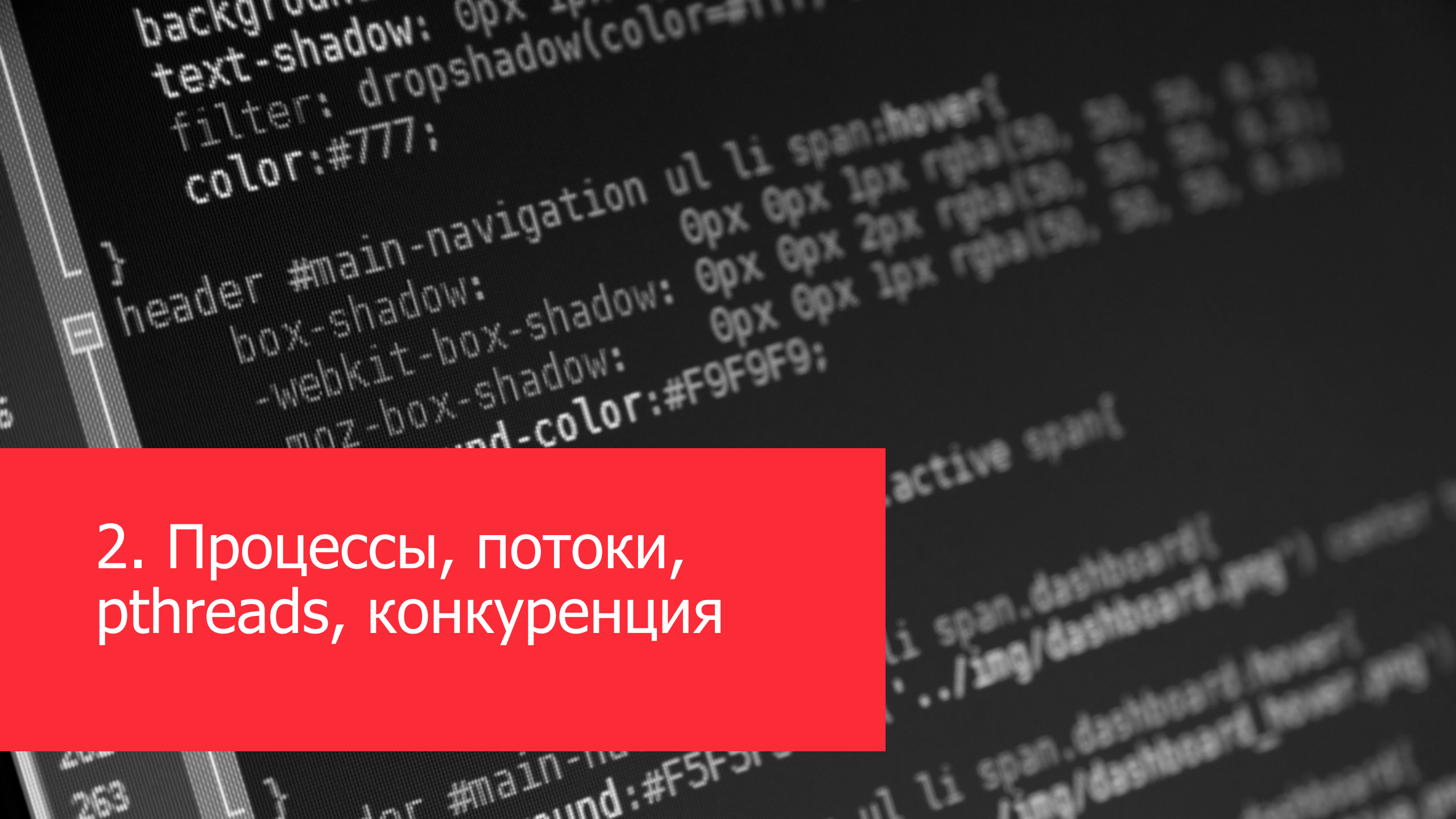





```
for n from 0 to N-1:
       for i from 0 to N-1:
              for j from 0 to N-1:

       C[i][j]+=A[i][n]*B[n][j]
```

```
n=0
i=0
j=1
C[0][1]+=A[0][0]*B[0][1]
```





2. Процессы, потоки, pthreads, конкуренция

Processes and threads

Processes and threads

Thread

Thread

Task

Address space

Program

pthreads

POSIX

The Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

First published in 1988.

Among other things the committee introduced **C POSIX library** – a bunch of header files that need to work on an OS for **POSIX-compliance**.

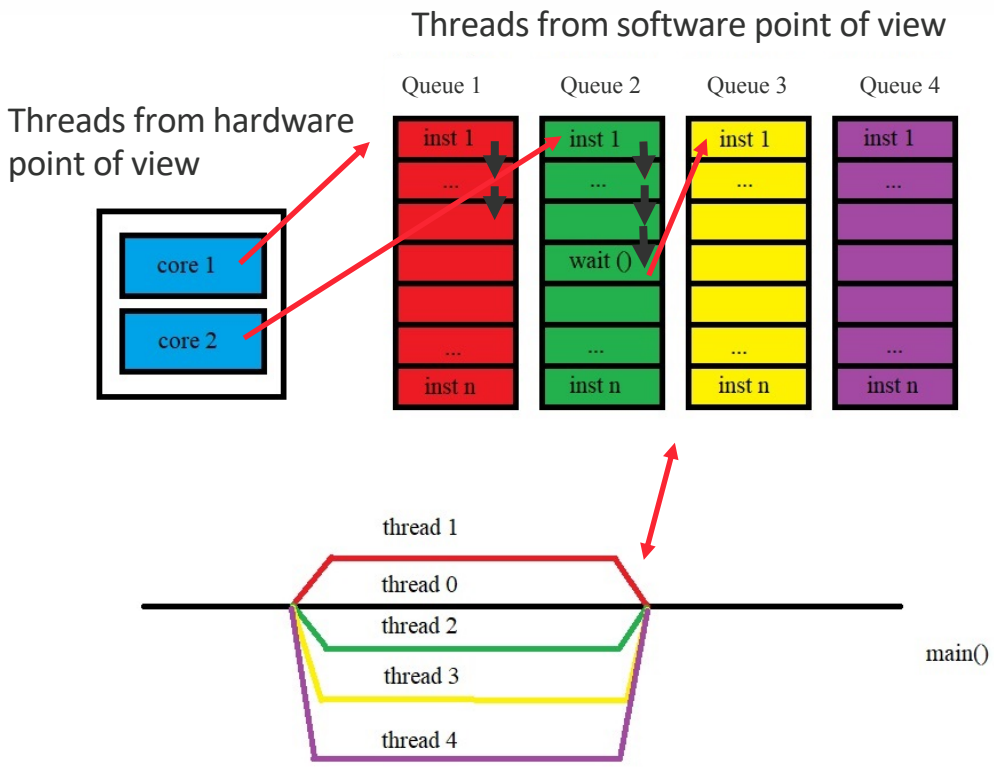
C POSIX library

<aio.h>	Asynchronous input and output
<arpa/inet.h>	Functions for manipulating numeric IP addresses (part of Berkeley sockets)
<assert.h>	Verify assumptions
<complex.h>	Complex Arithmetic, see C mathematical functions
<cpio.h>	Magic numbers for the cpio archive format
<ctype.h>	Character types
<dirent.h>	Allows the opening and listing of directories
<dlfcn.h>	Dynamic linking
<errno.h>	Retrieving Error Number
<fcntl.h>	File opening, locking and other operations
<fenv.h>	Floating-Point Environment (FPE), see C mathematical functions
<float.h>	Floating-point types, see C data types
<fmtmsg.h>	Message display structures
<fnmatch.h>	Filename matching
<ftw.h>	File tree traversal
<glob.h>	Pathname "globbing" (pattern-matching)
<grp.h>	User group information and control
<iconv.h>	Codeset conversion facility
<inttypes.h>	Fixed sized integer types, see C data types
<iso646.h>	Alternative spellings, see C alternative tokens
<langinfo.h>	Language information constants – builds on C localization functions
<libgen.h>	Pathname manipulation
<limits.h>	Implementation-defined constants, see C data types
<locale.h>	Category macros, see C localization functions
<math.h>	Mathematical declarations, see C mathematical functions
<monetary.h>	String formatting of monetary units
<mqueue.h>	Message queue
<ndbm.h>	NDBM database operations
<net/if.h>	Listing of local network interfaces
<netdb.h>	Translating protocol and host names into numeric addresses (part of Berkeley sockets)
<netinet/in.h>	Defines Internet protocol and address family (part of Berkeley sockets)
<netinet/tcp.h>	Additional TCP control options (part of Berkeley sockets)
<nl_types.h>	Localization message catalog functions
<poll.h>	Asynchronous file descriptor multiplexing
<pthread.h>	Defines an API for creating and manipulating POSIX threads
<pwd.h>	passwd (user information) access and control
<regex.h>	Regular expression matching
<sched.h>	Execution scheduling
<search.h>	Search tables
<semaphore.h>	POSIX semaphores
<setjmp.h>	Stack environment declarations
<signal.h>	Signals, see C signal handling
<spawn.h>	Process spawning
<stdarg.h>	Handle Variable Argument List
<stdbool.h>	Boolean type and values, see C data types
<stddef.h>	Standard type definitions, see C data types
<stdint.h>	Integer types, see C data types
<stdio.h>	Standard buffered input/output, see C file input/output
<stdlib.h>	Standard library definitions, see C standard library
<string.h>	Several String Operations, see C string handling
<strings.h>	Case-insensitive string comparisons
<stropts.h>	Stream manipulation, including ioctl
<sys/ipc.h>	Inter-process communication (IPC)
<sys/mman.h>	Memory management, including POSIX shared memory and memory mapped files
<sys/msg.h>	POSIX message queues
<sys/resource.h>	Resource usage, priorities, and limiting
<sys/select.h>	Synchronous I/O multiplexing
<sys/sem.h>	XSI (SysV style) semaphores
<sys/shm.h>	XSI (SysV style) shared memory
<sys/socket.h>	Main Berkeley sockets header
<sys/stat.h>	File information (stat et al.)
<sys/statvfs.h>	File System information
<sys/time.h>	Time and date functions and structures
<sys/times.h>	File access and modification times
<sys/types.h>	Various data types used elsewhere
<sys/uio.h>	Vectored I/O operations
<sys/un.h>	Unix domain sockets
<sys/utsname.h>	Operating system information, including uname
<sys/wait.h>	Status of terminated child processes (see wait)
<syslog.h>	System error logging
<tar.h>	Magic numbers for the tar archive format
<termios.h>	Allows terminal I/O interfaces
<tgmath.h>	Type-Generic Macros, see C mathematical functions
<time.h>	Type-Generic Macros, see C date and time functions
<trace.h>	Tracing of runtime behavior (DEPRECATED)
<ulimit.h>	Resource limiting (DEPRECATED in favor of <sys/resource.h>)
<unistd.h>	Various essential POSIX functions and constants
<utime.h>	inode access and modification times
<utmpx.h>	User accounting database functions
<wchar.h>	Wide-Character Handling, see C string handling
<wctype.h>	Wide-Character Classification and Mapping Utilities, see C character classification
<wordexp.h>	Word-expansion like the shell would perform

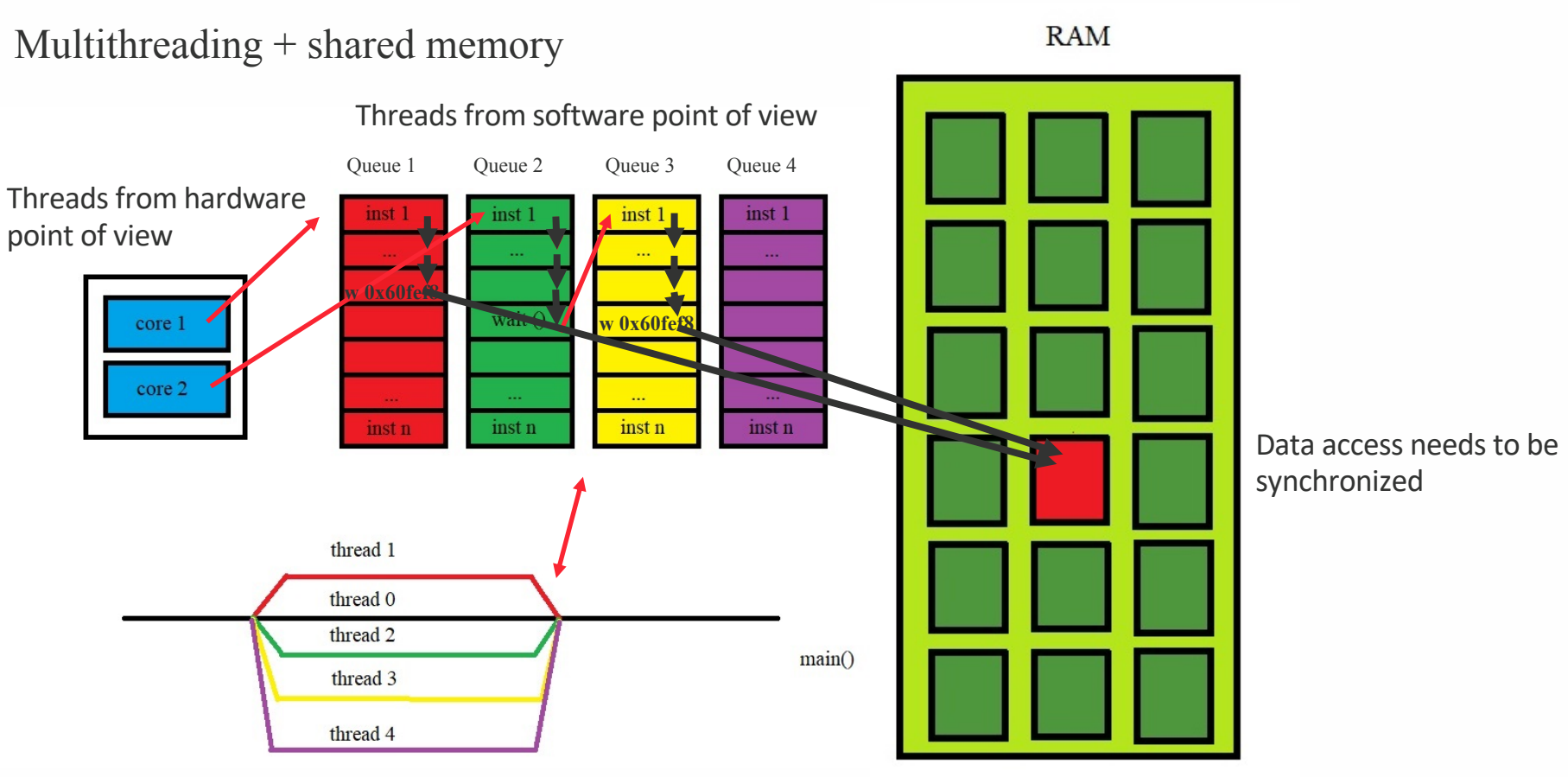
C POSIX library (Multithreading header files)

<aio.h>	Asynchronous input and output	<signal.h>	Signals, see C signal handling
<arpa/inet.h>	Functions for manipulating numeric IP addresses (part of Berkeley sockets)	<spawn.h>	Process spawning
<assert.h>	Verify assumptions	<stdarg.h>	Handle Variable Argument List
<complex.h>	Complex Arithmetic, see C mathematical functions	<stdbool.h>	Boolean type and values, see C data types
<cpio.h>	Magic numbers for the cpio archive format	<stddef.h>	Standard type definitions, see C data types
<ctype.h>	Character types	<stdint.h>	Integer types, see C data types
<dirent.h>	Allows the opening and listing of directories	<stdio.h>	Standard buffered input/output, see C file input/output
<dlfcn.h>	Dynamic linking	<stdlib.h>	Standard library definitions, see C standard library
<errno.h>	Retrieving Error Number	<string.h>	Several String Operations, see C string handling
<fcntl.h>	File opening, locking and other operations	<strings.h>	Case-insensitive string comparisons
<fenv.h>	Floating-Point Environment (FPE), see C mathematical functions	<stropts.h>	Stream manipulation, including ioctl
<float.h>	Floating-point types, see C data types	<sys/ipc.h>	Inter-process communication (IPC)
<fmtmsg.h>	Message display structures	<sys/mman.h>	Memory management, including POSIX shared memory and memory mapped files
<fnmatch.h>	Filename matching	<sys/msg.h>	POSIX message queues
<ftw.h>	File tree traversal	<sys/resource.h>	Resource usage, priorities, and limiting
<glob.h>	Pathname "globbing" (pattern-matching)	<sys/select.h>	Synchronous I/O multiplexing
<grp.h>	User group information and control	<sys/sem.h>	XSI (SysV style) semaphores
<iconv.h>	Codeset conversion facility	<sys/shm.h>	XSI (SysV style) shared memory
<inttypes.h>	Fixed sized integer types, see C data types	<sys/socket.h>	Main Berkeley sockets header
<iso646.h>	Alternative spellings, see C alternative tokens	<sys/stat.h>	File information (stat et al.)
<langinfo.h>	Language information constants – builds on C localization functions	<sys/statvfs.h>	File System information
<libgen.h>	Pathname manipulation	<sys/time.h>	Time and date functions and structures
<limits.h>	Implementation-defined constants, see C data types	<sys/times.h>	File access and modification times
<locale.h>	Category macros, see C localization functions	<sys/types.h>	Various data types used elsewhere
<math.h>	Mathematical declarations, see C mathematical functions	<sys/uio.h>	Vectored I/O operations
<monetary.h>	String formatting of monetary units	<sys/un.h>	Unix domain sockets
<mqueue.h>	Message queue	<sys/utsname.h>	Operating system information, including uname
<ndbm.h>	NDBM database operations	<sys/wait.h>	Status of terminated child processes (see wait)
<net/if.h>	Listing of local network interfaces	<syslog.h>	System error logging
<netdb.h>	Translating protocol and host names into numeric addresses (part of Berkeley sockets)	<tar.h>	Magic numbers for the tar archive format
<netinet/in.h>	Defines Internet protocol and address family (part of Berkeley sockets)	<termios.h>	Allows terminal I/O interfaces
<netinet/tcp.h>	Additional TCP control options (part of Berkeley sockets)	<tgmath.h>	Type-Generic Macros, see C mathematical functions
<nl_types.h>	Localization message catalog functions	<time.h>	Type-Generic Macros, see C date and time functions
<poll.h>	Asynchronous file descriptor multiplexing	<trace.h>	Tracing of runtime behavior (DEPRECATED)
<pthread.h>	Defines an API for creating and manipulating POSIX threads	<ulimit.h>	Resource limiting (DEPRECATED in favor of <sys/resource.h>)
<pwd.h>	passwd (user information) access and control	<unistd.h>	Various essential POSIX functions and constants
<regex.h>	Regular expression matching	<utime.h>	inode access and modification times
<sched.h>	Execution scheduling	<utmpx.h>	User accounting database functions
<search.h>	Search tables	<wchar.h>	Wide-Character Handling, see C string handling
<semaphore.h>	POSIX semaphores	<wctype.h>	Wide-Character Classification and Mapping Utilities, see C character classification
<setjmp.h>	Stack environment declarations	<wordexp.h>	Word-expansion like the shell would perform

Multithreading



Multithreading + shared memory



Data access needs to be synchronized

A short example

Global variables. They become automatically shared between threads.

```c
int counter;
pthread_mutex_t lock;

void* doSomeThing(void *arg)
{
    pthread_mutex_lock(&lock);
    counter += 1;
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    int N = 500;
    pthread_t tid[N];
    int i = 0;
    int err;
    pthread_mutex_init(&lock, NULL);
    for(i=0; i<N; i++) pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
    for(i=0; i<N; i++) pthread_join(tid[0], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

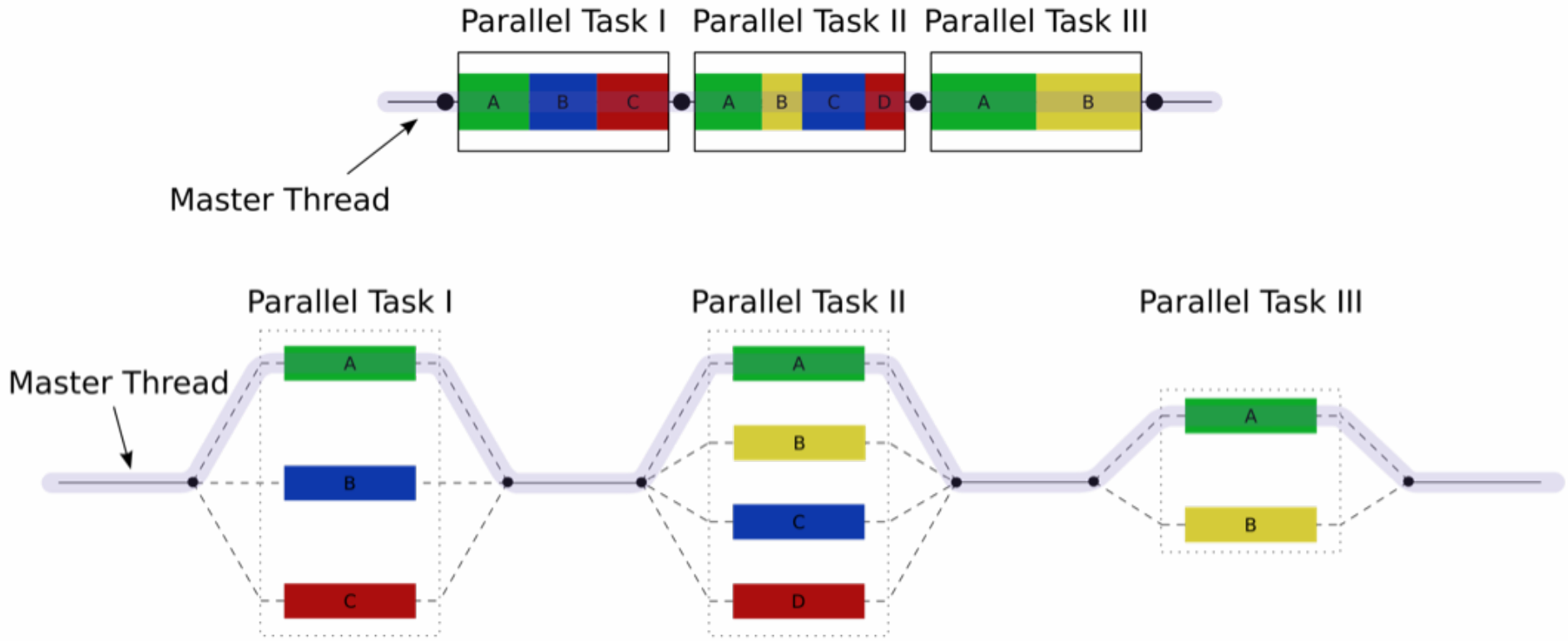
Notice the definition. The function has to have 1 argument, which is pointer to void and return pointer to void. Before using *arg* cast it to the needed type. For ex: *(int) arg*.

You can also write pthread_exit(0); instead of return NULL;

4th argument is arguments that are passed to doSomeThing. We can pass there whatever we want but we need to cast it as poniter to void. For ex: *(void*) i*.

Compile command looks something like this: gcc prog.c -o prog.exe -pthread -lm

Fork-join concept



https://en.wikipedia.org/wiki/Fork-join_model

Race condition (состояние гонки)

Example: calculate the sum of array elements
“REDUCTION(SUM)”

s = sum(a[i])

s=s+a[i]

1. Read s from memory
2. Add a value
3. Write the updated value back

memory element that holds the variable ’s’ which is the sum of the elements

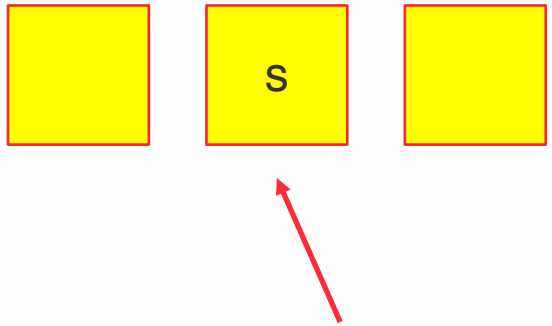
Race condition (состояние гонки)

Example: calculate the sum of array elements
“REDUCTION(SUM)”

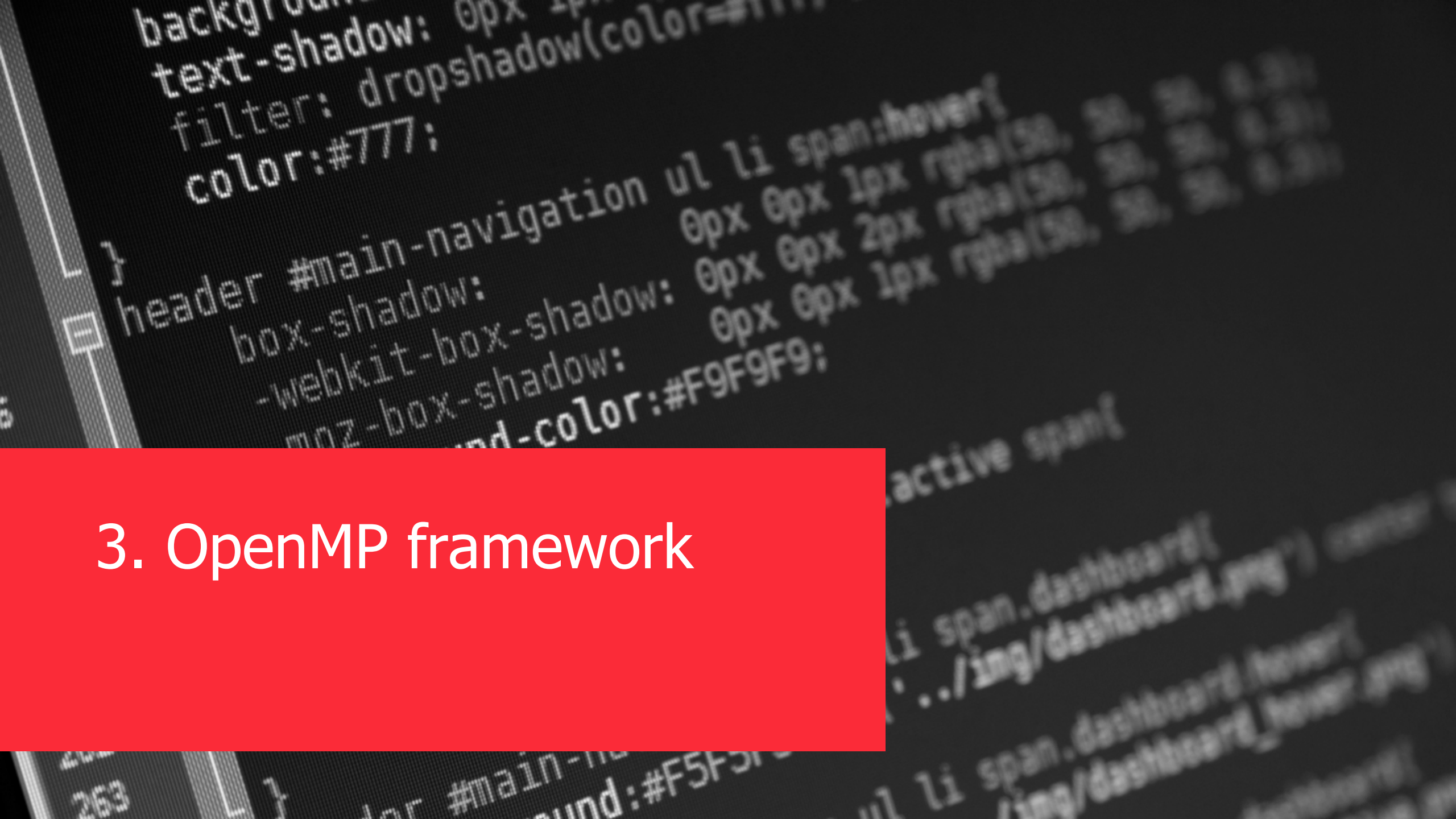
s = sum(a[i])

s=s+a[i] →

1. Read s from memory
2. Add a value
3. Write the updated value back

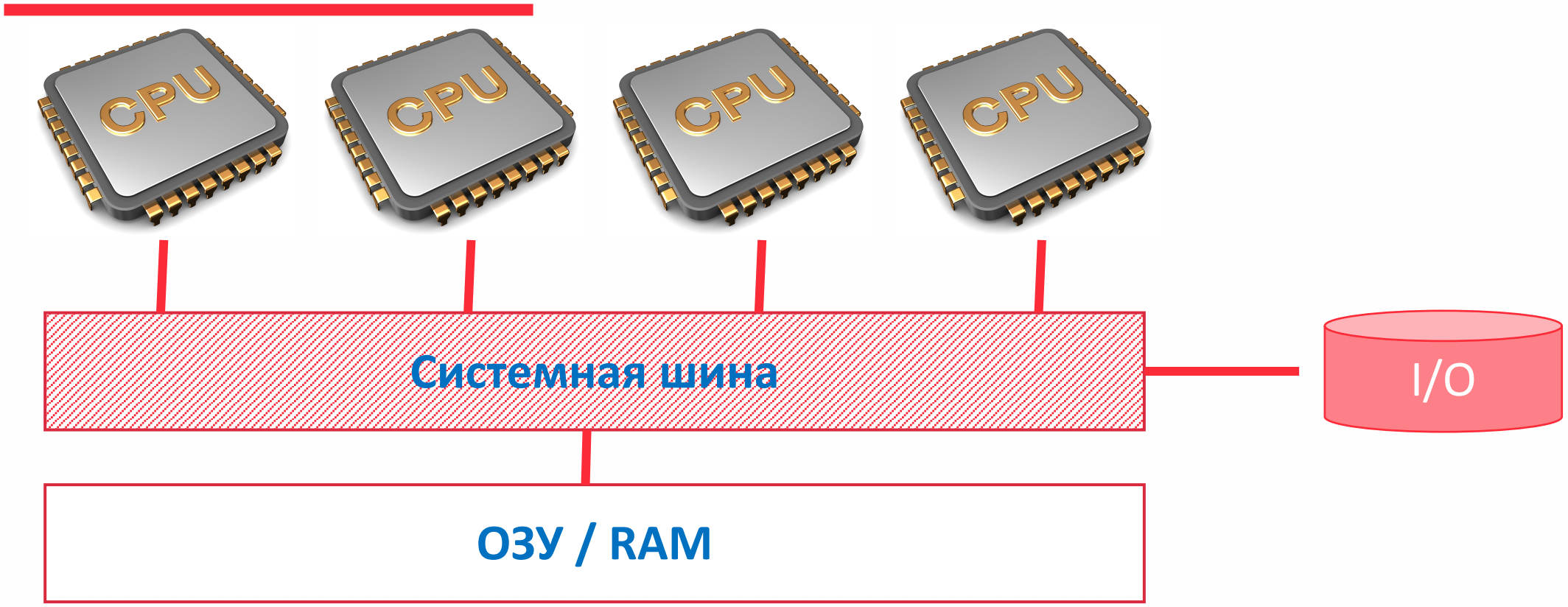


memory element that holds the variable ’s’ which is the sum of the elements



3. OpenMP framework

OpenMP. Системы с общей памятью



OpenMP:

- API для написания многопоточных приложений
- Набор директив компилятора, функций и переменных окружения
- Значительно упрощает разработку многопоточных приложений (особенно в HPC)

OpenMP. Директивы компилятора

В большей степени OpenMP – это директивы компилятора. В общем виде:

```
#pragma omp construct [clause [clause]…]
#pragma omp parallel num_threads(4)
```

Прототипы функций и типы данных находятся в header-файле:

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

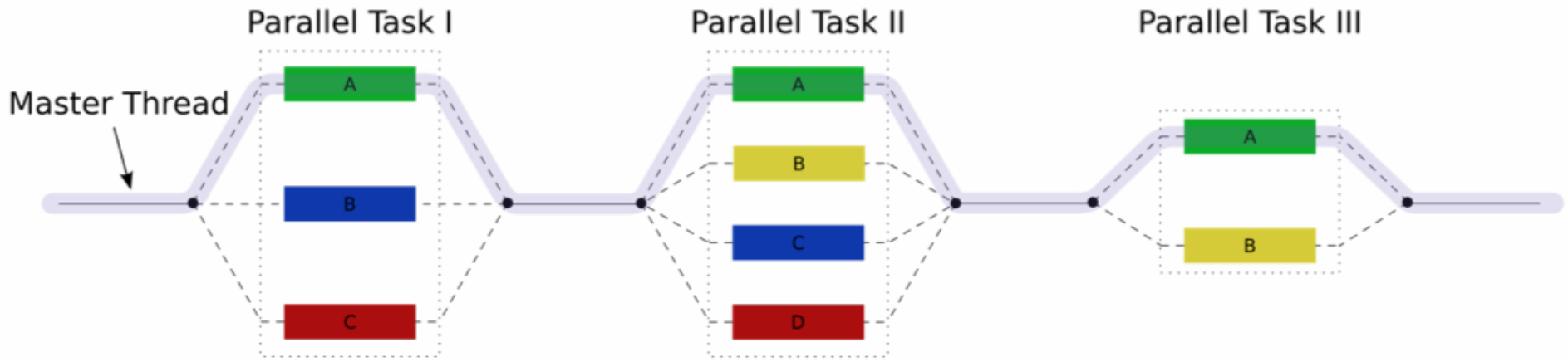
Переменные окружения можно задать на системном уровне, например:

```
OMP_NUM_THREADS
```

OpenMP. Концепция Fork-Join

У разработчика есть явный контроль параллелизации за счет модели параллелизма типа fork-join:

- все OpenMP программы начинаются как один процесс (мастер-поток, master thread), который выполняется до директивы компилятора (parallel region construct)
- **FORK**: мастер-поток создает команду параллельных потоков
- **JOIN**: когда потоки заканчивают выполнение параллельных инструкций, они синхронизуются и завершают свою жизнь, остается только мастер-поток.



OpenMP. Концепция Fork-Join

Разработчик вставляет в код директивы, которые поясняют компилятору:

- какие части программы должны исполняться параллельно
- какие задания будут исполнять различные потоки
- какие данные общие (shared), какие данные приватные (private)
- `#pragma omp`

Компилятор генерирует многопоточный код

Негласное правило: один поток на ядро (2 или 4 если используется HT)

Работа с данными в параллельных частях программы приводит к необходимости **синхронизации потоков**.

OpenMP. Hello, world!

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

int main() {

#pragma omp parallel
printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Compile line:

```
gcc -fopenmp helloWorld.c
```

OpenMP. Как задать кол-во потоков

Через переменную окружения:

```
setenv OMP_NUM_THREADS 2          (cshell)
export OMP_NUM_THREADS=2          (bash)
```

Функция библиотеки OpenMP:

```
omp_set_num_threads(2);
```

В директиве компилятору:

```
#pragma omp parallel num_threads(2)
```

OpenMP. Конструкция parallel

```
#include <omp.h>

int main(){
  int var1, var2, var3;

  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
 {
    ...parallel section
 }

 ...resume serial code

}
```

OpenMP. Конструкция parallel

- Когда поток достигает директивы parallel, он становится мастер-потоком и его номер становится равен 0 (остальные потоки имеют уникальные номера)
- Оптимально если все потоки исполняют один и тот же код, но можно делать параллелизм по задачам (task parallelism, work sharing)
- В конце конструкции parallel всегда **барьер**! Только мастер-поток продолжает исполнение программы
- Если какой-то поток аварийно останавливается (terminate) внутри параллельной части, то все потоки останавливаются и результат не определен.
- Если программа скомпилирована без использования OpenMP – она компилируется (возможно, за исключением части функций) и работает последовательно.
- Простота параллелизации!
- Простота поддержки кода (code maintenance) – одна версия для параллельной и последовательной программы.

```
static void critical_example(int v) {

    // Shared variables
    int a = 0;
    int b = v;

    #pragma omp parallel
    {
        // Private variable - one for each thread
        int c;

        // Reading from "b" is safe: it is read-only
        // Writing to "c" is safe: it is private
        c = b;

        // Reading from "c" is safe: it is private
        // Writing to "c" is safe: it is private
        c = c * 10;

        #pragma omp critical
        {
            // Any modifications of "a" are safe:
            // we are inside a critical section
            a = a + c;
        }
    }

    // Reading from "a" is safe:
    // we are outside parallel region
    std::cout << a << std::endl;
}
```

shared
private
critical

OpenMP:Небольшое сравнение с pthreads

```
void thunk ()
{
        foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
          &tid[i],0,thunk, 0);
think();

for (int i = 1; i < 4; ++i)
   pthread_join (tid[i]);
```

```
#pragma omp parallel num_threads(4)
{
      foobar ();
}
```

OpenMP. Work-sharing constructs

- Не создает новых потоков, только разделяет работу между всеми потоками в команде
- **На входе нет барьера, на выходе есть**! (но можно отменить барьер на выходе)

3 типа work-sharing constructs:

- **for loop:** параллелизация циклов (data parallelism)
- **sections:** разделение на параллельные участки, в каждом участке свои инструкции
- **single:** часть кода, которая исполняется строго одним потоком
- **master:** то же, что и single, но выполняет только поток с номером 0 (мастер)

OpenMP. Work-sharing constructs

- Не создает новых потоков, только разделяет работу между всеми потоками в команде
- **На входе нет барьера, на выходе есть**! (но можно отменить барьер на выходе)

3 типа work-sharing constructs:

- **for loop:** параллелизация циклов (data parallelism)
- **sections:** разделение на параллельные участки, в каждом участке свои инструкции
- **single:** часть кода, которая исполняется строго одним потоком
- **master:** то же, что и single, но выполняет только поток с номером 0 (мастер)

OpenMP. Пример parallel for

```
int main()
{
    int sum=0;
    int a[N], i;

    #pragma omp parallel for
    for(i=0;i<N;i++)
    {
        a[i]=i;
        printf("iterate i=%3d by thread %3d\n",i, omp_get_thread_num());
    }
    return 0;

}
```

OpenMP. Пример parallel for

```
int main()
{
    int sum=0;
    int a[N], i;

    #pragma omp parallel for
    for(i=0;i<N;i++)
    {
        a[i]=i;
        printf("iterate i=%3d by thread %3d\n",i, omp_get_thread_num());
    }
    return 0;

}
```

```
#pragma omp parallel
{
    #pragma omp for
```

OpenMP. for или не for?

```
#pragma omp parallel for
for(i=0;i<N;i++)
{
        a[i]=i;
        printf("iterate i=%3d by thread %3d\n", i, omp_get_thread_num());
}
```

```
#pragma omp parallel
for(i=0;i<N;i++)
{
        a[i]=i;
        printf("iterate i=%3d by thread %3d\n", i, omp_get_thread_num());
}
```

OpenMP. Общий вид директив

```
#pragma omp directive  [clause ...]
                        if (scalar_expression)
                       private (list)
                       shared (list)
                       default (shared | none)
                       firstprivate (list)
                       reduction (operator: list)
                       copyin (list)
                       num_threads (integer-expression)
```

- case sensitive
- одна директива в конструкции
- если следуют больше одной инструкции, надо обернуть в {}
- длинную строку можно переносить обратным слэшем (“\”)

OpenMP. parallel for

```
#pragma omp for [clause ...]
                    schedule (type [,chunk])
                    private (list)
                    firstprivate(list)
                    lastprivate(list)
                    shared (list)
                    reduction (operator: list)
                    nowait
```

SCHEDULE: описывает как разделять между потоками итерации цикла:

- static = статично разделять на части размера chunk, по умолчанию chunk примерно одинаковый для всех потоков
- dynamic = разделено на части chunk, но кто первый пришел, тот и забрал очередной chunk.
- guided = размер chunk уменьшается со временем
- runtime = определяется переменной окружения OMP_SCHEDULE

OpenMP. Пример parallel for

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

   #pragma omp for nowait
   for (i=0;i<n;i++)
     b[i] = += a[i];

   #pragma omp for nowait
   for (i=0;i<n;i++)
     x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

fork занимает время – не выходите без надобности из конструкции parallel
nowait - отменяет барьер на выходе из work-sharing конструкции

OpenMP. Пример parallel sections

```
#pragma omp sections [clause ...]
                     private (list)
                     firstprivate(list)
                     lastprivate(list)
                     reduction (operator: list)
                     nowait
{
 #pragma omp section
   structured block
 #pragma omp section
   structured block
}
```

nowait - отменяет барьер на выходе из work-sharing конструкции
возможна расходимость потоков (thread divergence)
возможно неравномерное распределение работы

OpenMP. Пример single

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate(list)
                   nowait
 structured block
```

- только один поток выполняет конструкцию single (кто первый пришел)
- важно если часть кода не thread safe (например, I/O)
- в конце барьер, но его можно отменить (nowait)

OpenMP. Пример firstprivate

Что здесь не так?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

OpenMP. Пример firstprivate

Что здесь не так?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

By default, `x` is shared variable (i is private).

Could have: Thread 0 set `x` for some `i`.
Thread 1 sets `x` for different `i`.
Thread 0 uses `x` but it is now incorrect.

OpenMP. Пример firstprivate

Что здесь не так?

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

Что насчет i, dx, y?

By default `dx, n, y` shared.
`dx, n` used but not changed. `y` changed, but independently for each `i`

OpenMP. Пример firstprivate

Что здесь не так?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

OpenMP. Пример firstprivate

Что здесь не так?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

private(dx) создает новую приватную переменную для каждого потока, но не гарантирует инициализацию

firstprivate clause creates private variables and initializes to the value from the master thread before the loop.

lastprivate copies last value computed by a thread (for *i*=*n*) to the maser thread copy to continue execution.

OpenMP. clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. this gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (`none` | `shared`). In fortran can also have private. Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 `for` loops where second depends on data from first that same threads execute same iterates)

More clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

```
#pragma omp parallel for default(none) \
                         shared(n,a) \
                         reduction(+:sum)
  for (i=0;i<n;i++)
    sum += a[i]
  /* end of parallel reduction  */
```

Also other arithmetic and logical ops., `min`,`max` instrinsics in Fortan only.

Потоковая безопасность (thread safety)

- `rand()` is not thread-safe, can lead to Heisenbug (Schrödinbug)
- rand_r is thread safe is because the function is entirely pure.
- Thread **safe** list of functions here
- Functions **unsafe** to call the function concurrently :

1. Rand()
2. Strtok()
3. Basename()
4. Others here

Генерация случайных чисел

omp_picalc.c

```
#pragma omp parallel ...
{
  unsigned int seed =
    123456789 *
    thread_id;
  ...
  double x =
    ((double)
    rand_r(&seed))...
 }
```

TIMING

```
> gcc omp_picalc.c -fopenmp
>time -p a.out 100000000
npoints:  100000000
hits:   78541717
pi_est:   3.141669
```

omp_picalc_rand_r.c:

```
unsigned int seed =
  123456789;
#pragma omp parallel...
{
  ...
  double x =
    ((double) rand_r(&seed))...
}
```

TIMING

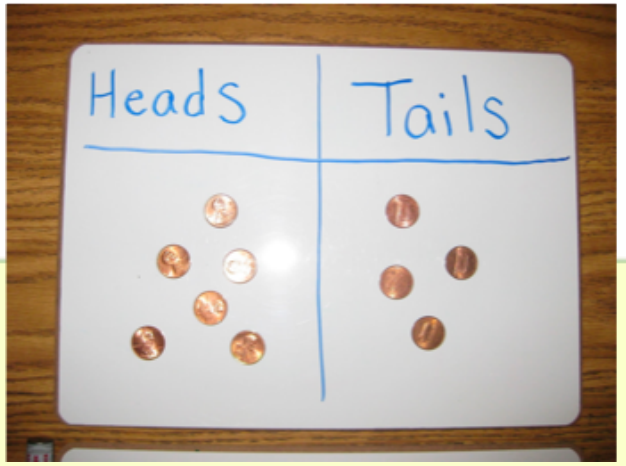
```
> gcc omp_picalc_rand_r.c -fopenmp
>time -p a.out 100000000
npoints:  100000000
hits:   77951102
pi_est:   3.118044
```

```
/***  OMP ***/
const int nThreads = 4;  // number of threads to use
unsigned int seeds[nThreads];

void seedThreads() {
    int my_thread_id;
    unsigned int seed;
    #pragma omp parallel private (seed, my_thread_id)
    {
        my_thread_id = omp_get_thread_num();

        //create seed on thread using current time
        unsigned int seed = (unsigned) time(NULL);

        //munge the seed using our thread number so that each thread has its
        //own unique seed, therefore ensuring it will generate a different set of numbers
        seeds[my_thread_id] = (seed & 0xFFFFFFF0) | (my_thread_id + 1);

        printf("Thread %d has seed %u\n", my_thread_id, seeds[my_thread_id]);
    }

}
/***  OMP ***/
```

```
    /***  OMP ***/
    omp_set_num_threads(nThreads);
    seedThreads();
    /***  OMP ***/
```

Link to the file

Пример использование массива seed




```
#pragma omp parallel num_threads(nThreads) default(none) \
        private(numFlips, tid, seed) \
        shared(trialFlips, seeds) \
        reduction(+:numHeads, numTails)
    {
        tid = omp_get_thread_num();    // my thread id
        seed = seeds[tid];             // it is much faster to keep a private copy of our seed
                srand(seed);                    //seed rand_r or rand

        #pragma omp for
        for (numFlips=0; numFlips<trialFlips; numFlips++) {
//          in Windows, can use rand()
//            if (rand()%2 == 0) // if random number is even, call it heads
            // linux: rand_r() is thread safe, to be run on separate threads concurrently
            if (rand_r(&seed)%2 == 0) // if random number is even, call it heads
                numHeads++;
            else
                numTails++;
        }

    }
```

Note: in Linux we need to use the rand_r() function for thread-safe generation of the numbers. However, in Windows, the rand() function is thread-safe.

for loop scheduling

Loop Scheduling - 4 Types

Static

- So far only done static scheduling with fixed size chunks
- Threads get fixed size chunks in rotating fashion
- Great if each iteration has same work load

Dynamic

- Threads get fixed chunks but when done, request another chunk
- Incurs more overhead but balances uneven load better

Guided

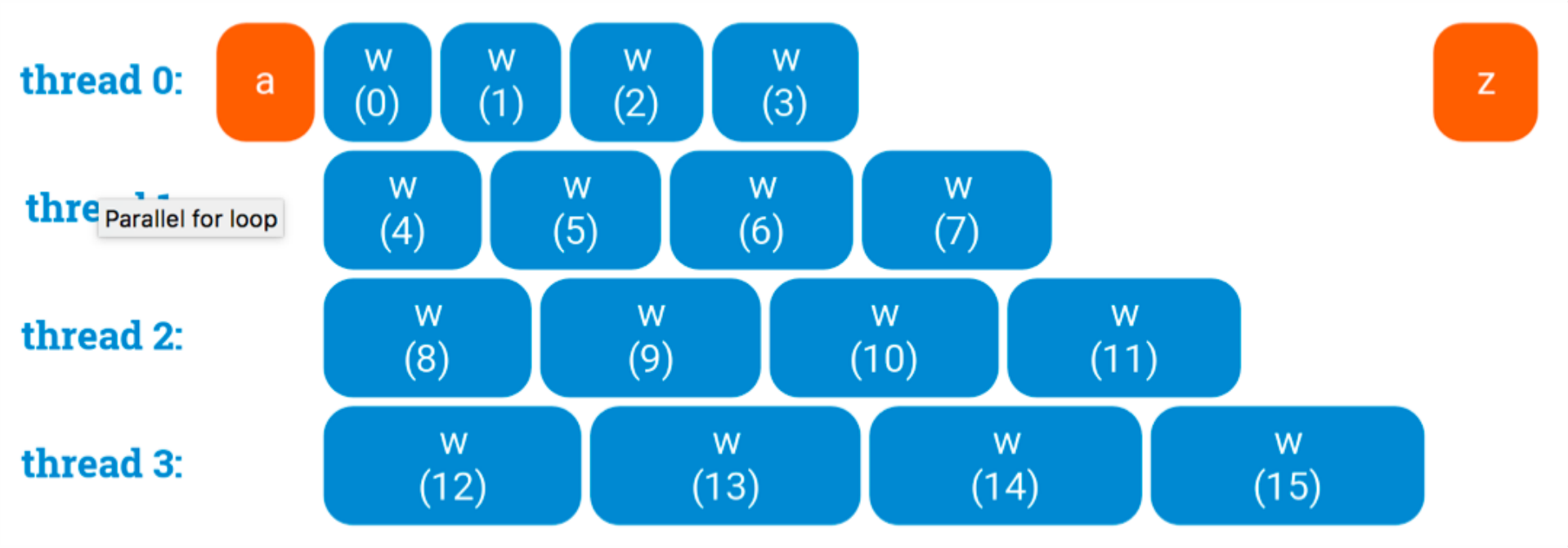
- Hybrid between static/dynamic, start with each thread taking a "big" chunk
- When a thread finishes, requests a "smaller" chunk, next request is smaller

Runtime

- Environment variables used to select one of the others
- Flexible but requires user awareness: *What's an environment variable?*

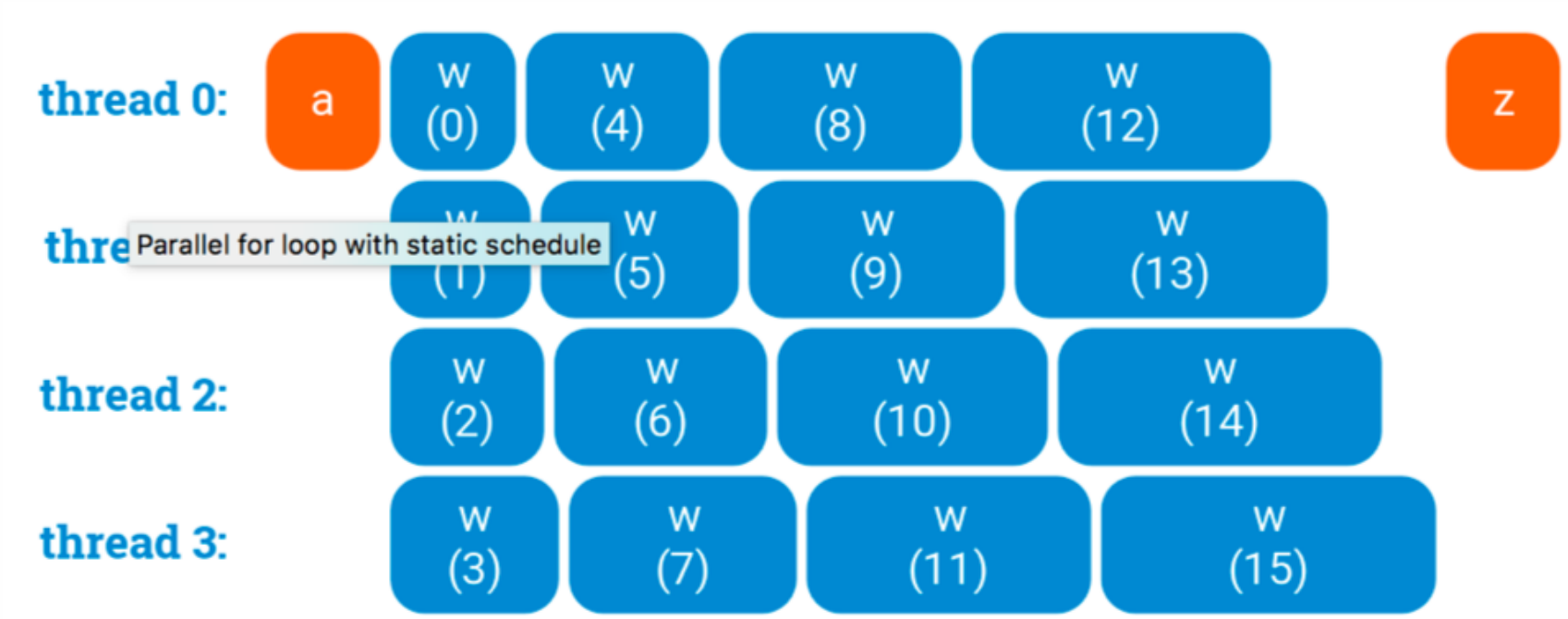
for loop scheduling

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



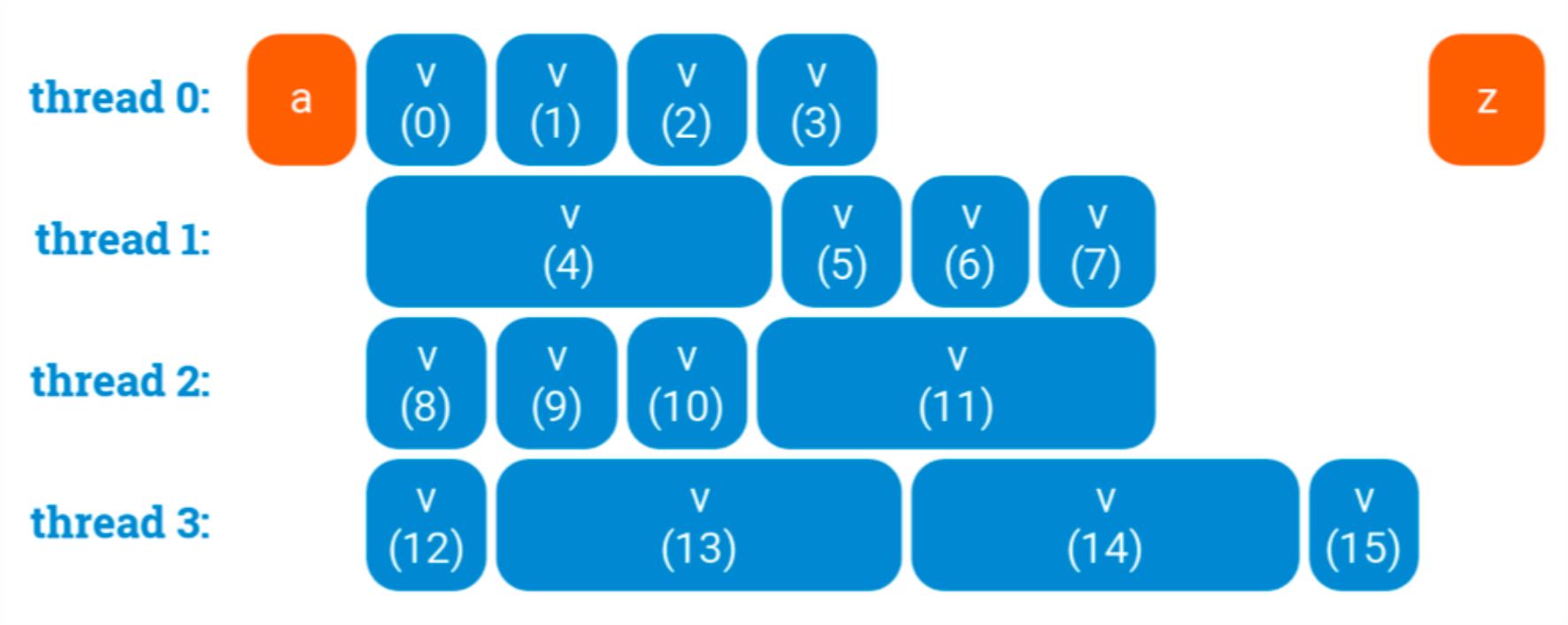
for loop scheduling (static)

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



for loop scheduling

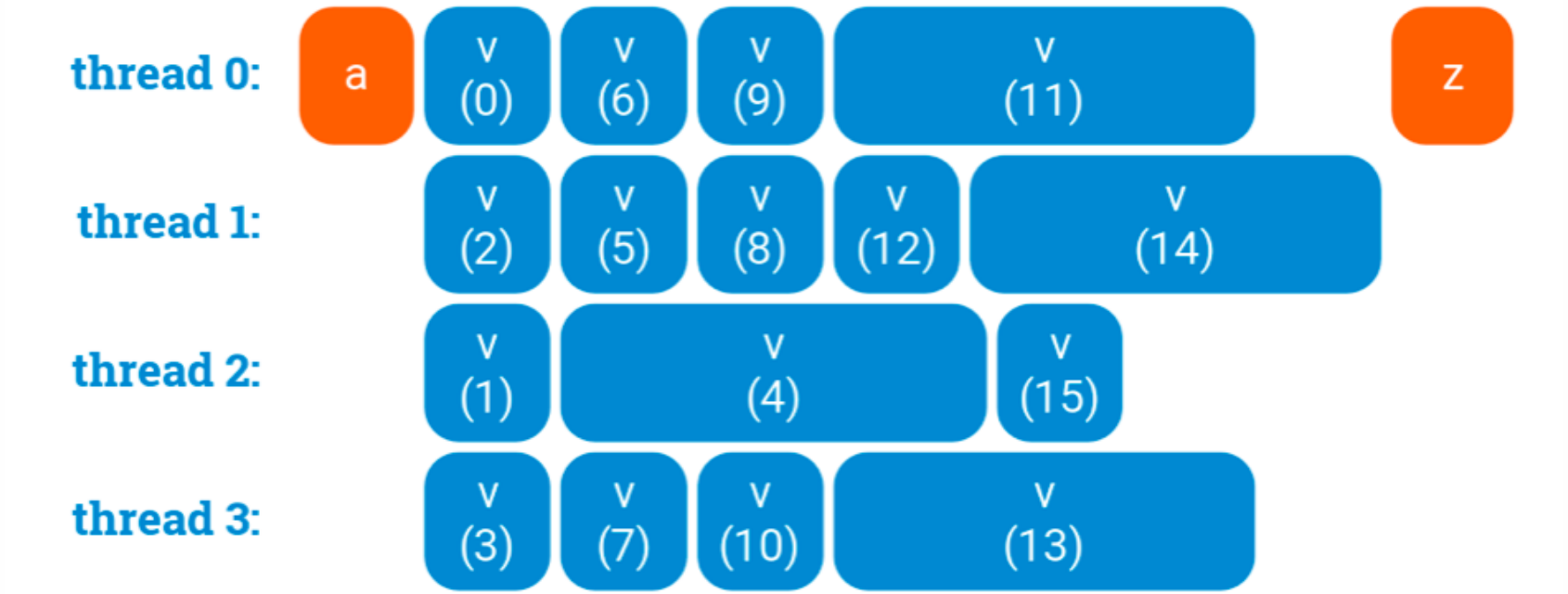
```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



for loop scheduling (dynamic)

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```

- dynamic scheduling is expensive
- no guarantee to be optimal

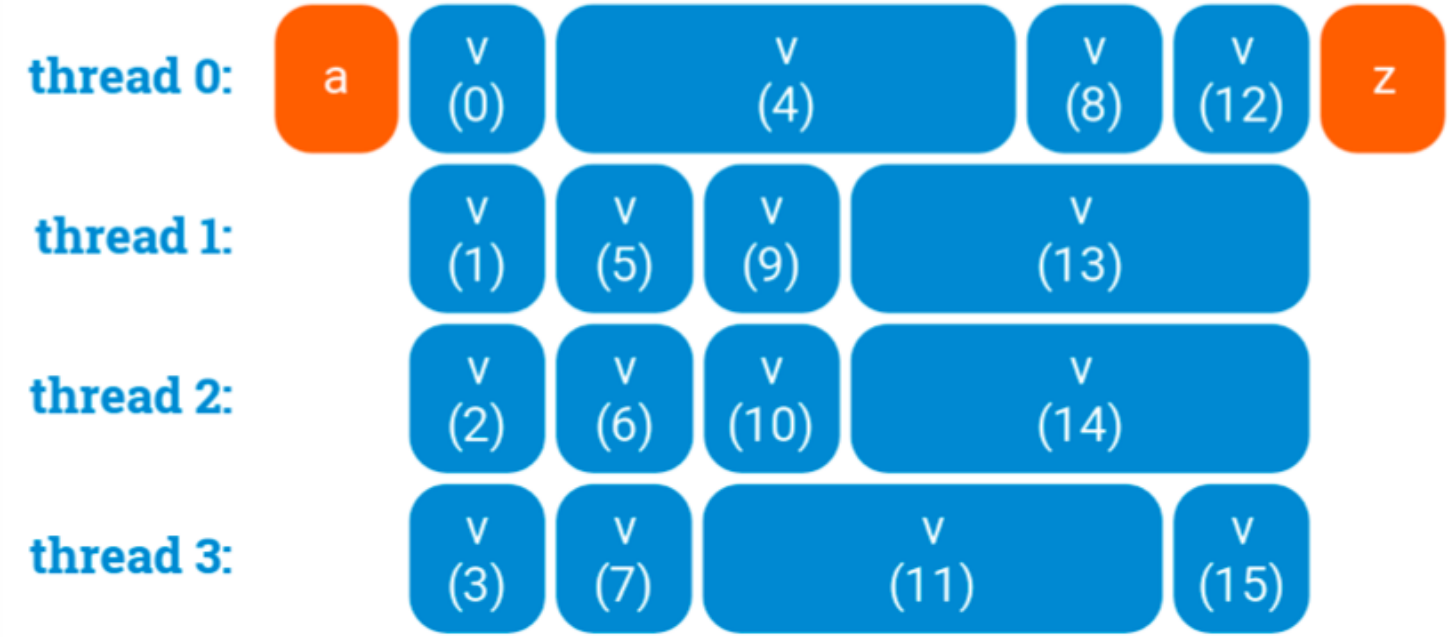


for loop scheduling (dynamic)

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```

- dynamic scheduling is expensive
- no guarantee to be optimal

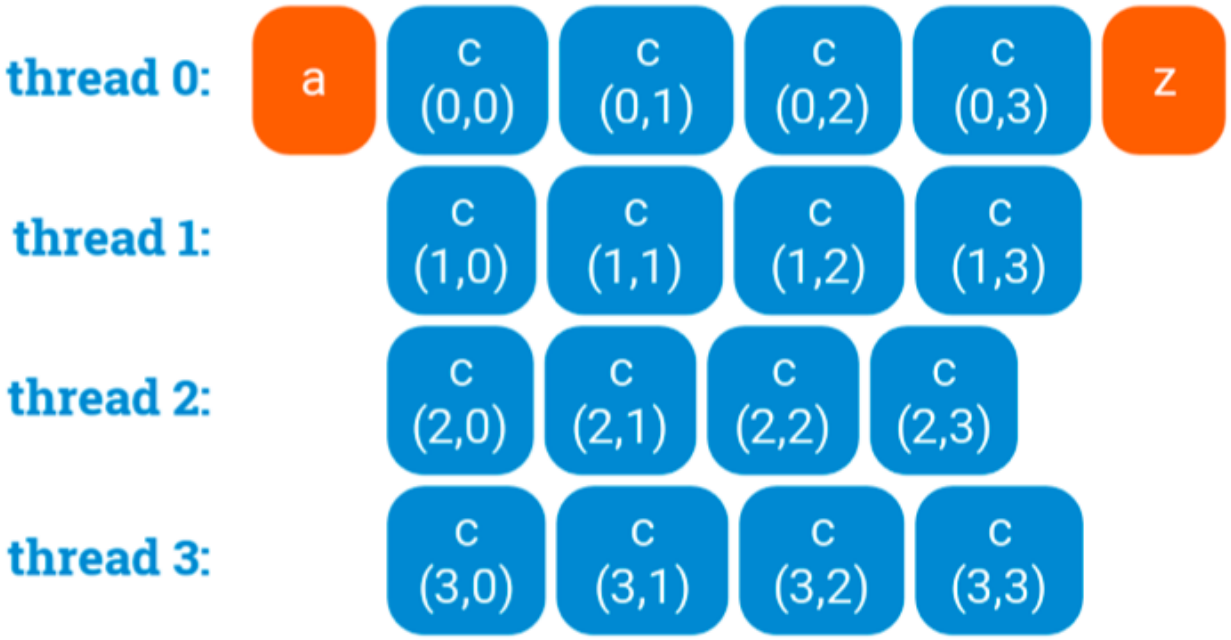
For example:



Parallelizing nested loops

Often easy enough to parallelize the inner loop

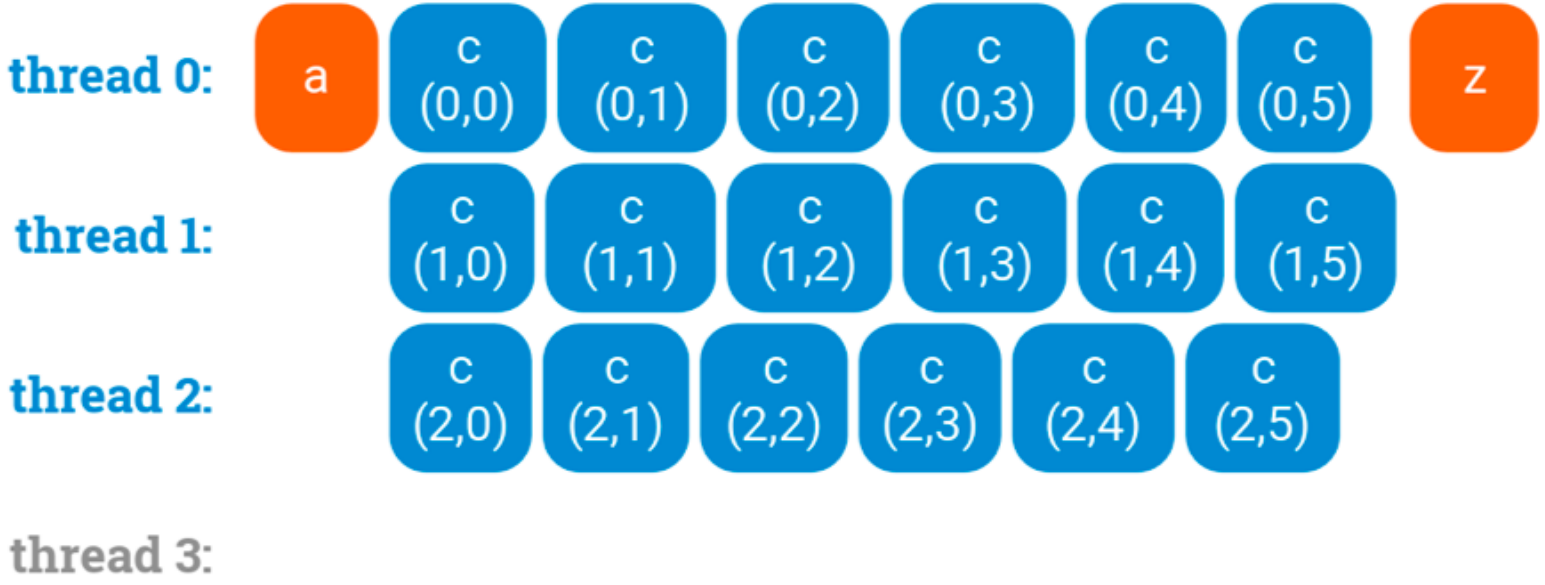
```
a();
#pragma omp parallel for
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        c(i, j);
    }
}
z();
```



Parallelizing nested loops

For example, the outer loop can be too short:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

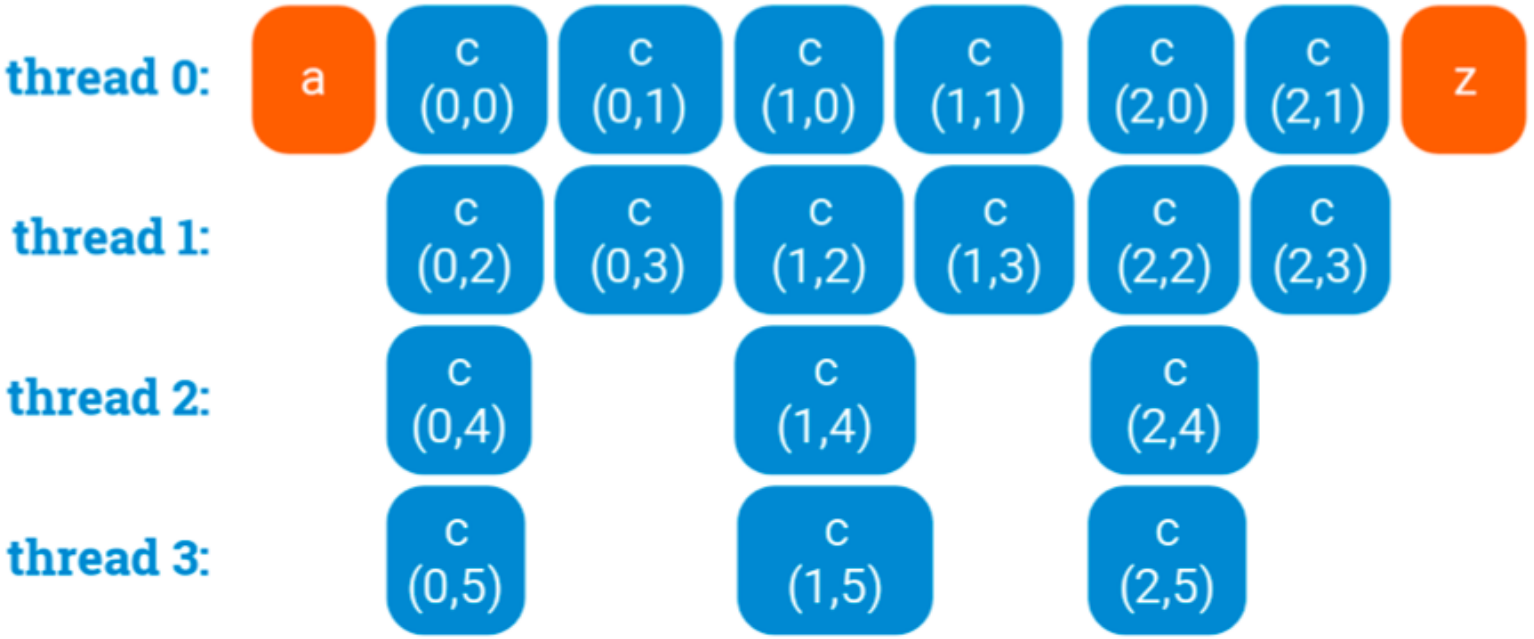


Parallelizing nested loops (bad way)

Idea: parallelize the inner loop

```
a();
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

However, no guarantee, that it will be any faster

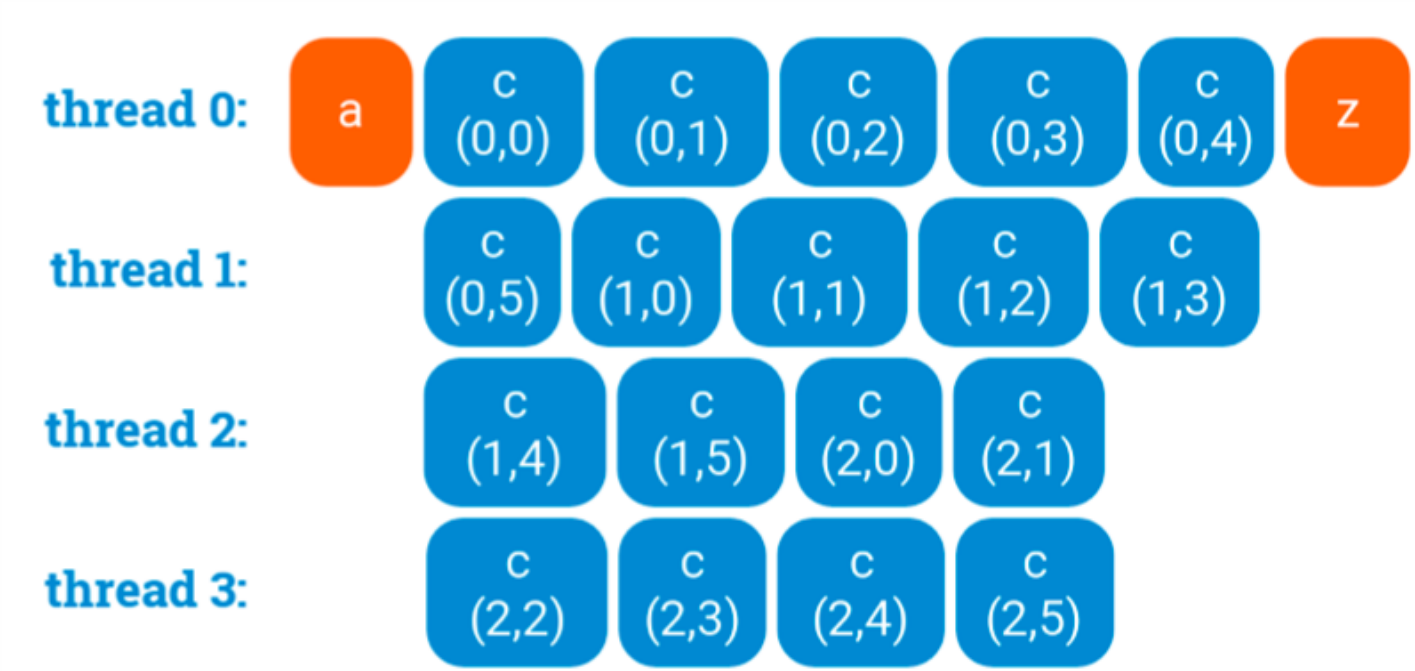


Parallelizing nested loops (good way 1)

```
a();
#pragma omp parallel for
for (int ij = 0; ij < 3 * 6; ++ij) {
    c(ij / 6, ij % 6);
}
z();
```

Parallelizing nested loops (good way 2)

```
a();
#pragma omp parallel for collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



Race condition example

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

int main(){

  int x = 2;

  #pragma omp parallel shared(x) num_threads(2)
  {
    if (1 == omp_get_thread_num()){
      x = 5;
    }
    else {
      printf("1: Thread %d has x = %d\n",omp_get_thread_num(),x);
    }

    #pragma omp barrier

    if (0 == omp_get_thread_num()) {
      printf("2: thread %d has x = %d\n",omp_get_thread_num(),x);
    }
    else {
      printf("3: thread %d has x = %x\n",omp_get_thread_num(),x);
    }

  }
  return 0;
}
```

Synchronization in OpenMP

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions .
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- locks: low-level run-time library routines (like mutex vars., semaphores)
- flush operation - forces the executing thread to make its values of shared data consistent with shared memory
- master (like single but not implied barrier at end), *ordered*, ...

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Atomic example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)   \
                    reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

Locks

Locks control access to shared resources. Up to implementation to use spin locks (busy waiting) or not.

- Lock variables must be accessed only through locking routines:

  ```
  omp_init_lock   omp_destroy_lock
  omp_set_lock    omp_unset_lock   omp_test_lock
  ```

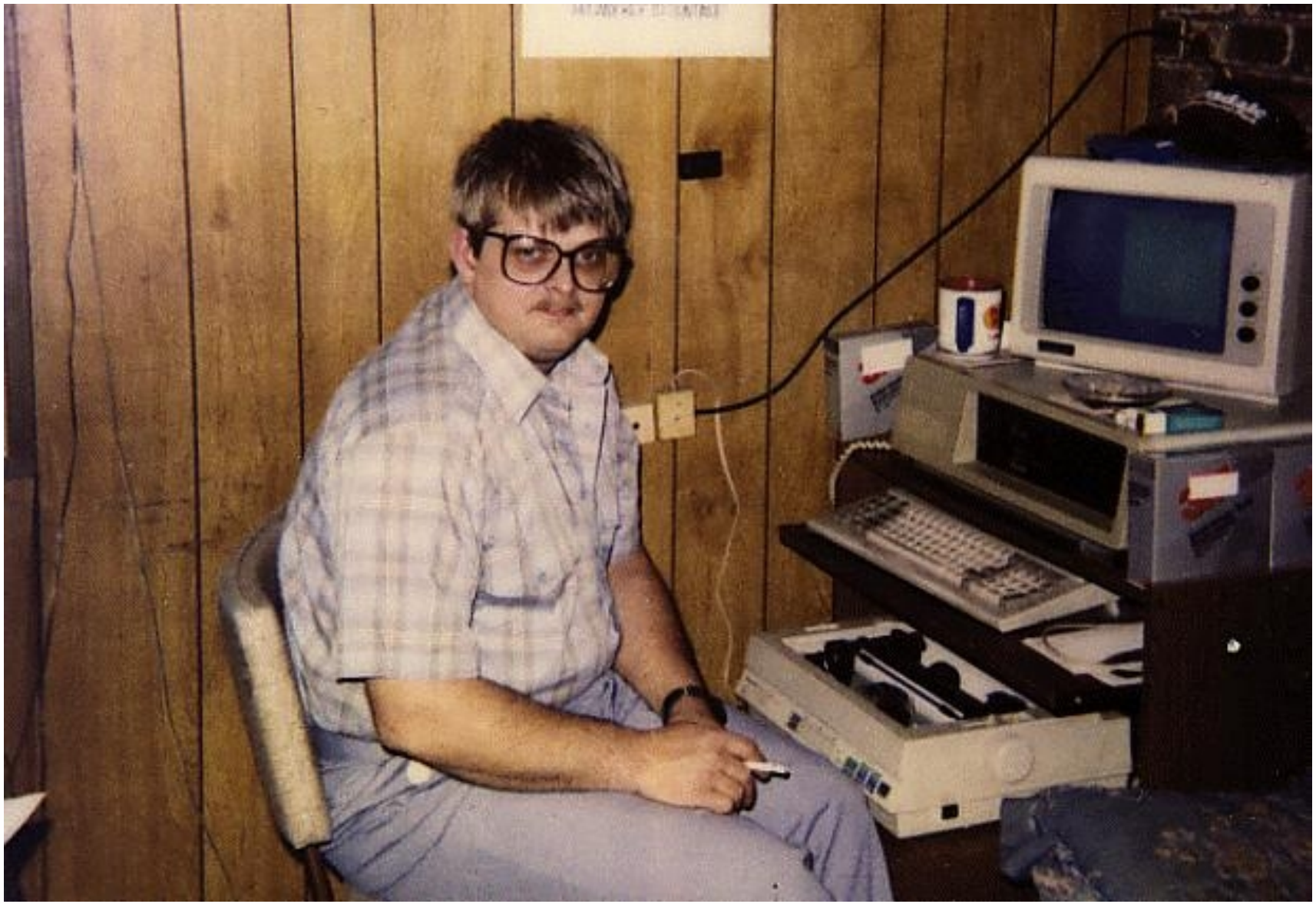
- In C, lock is a type `omp_lock_t` or `omp_nest_lock_t` (In Fortran lock variable is integer)
- initial state of lock is unlocked.
- `omp_set_lock(omp_lock_t *lock)` forces calling thread to wait until the specified lock is available. (Non-blocking version is `omp_test_lock`

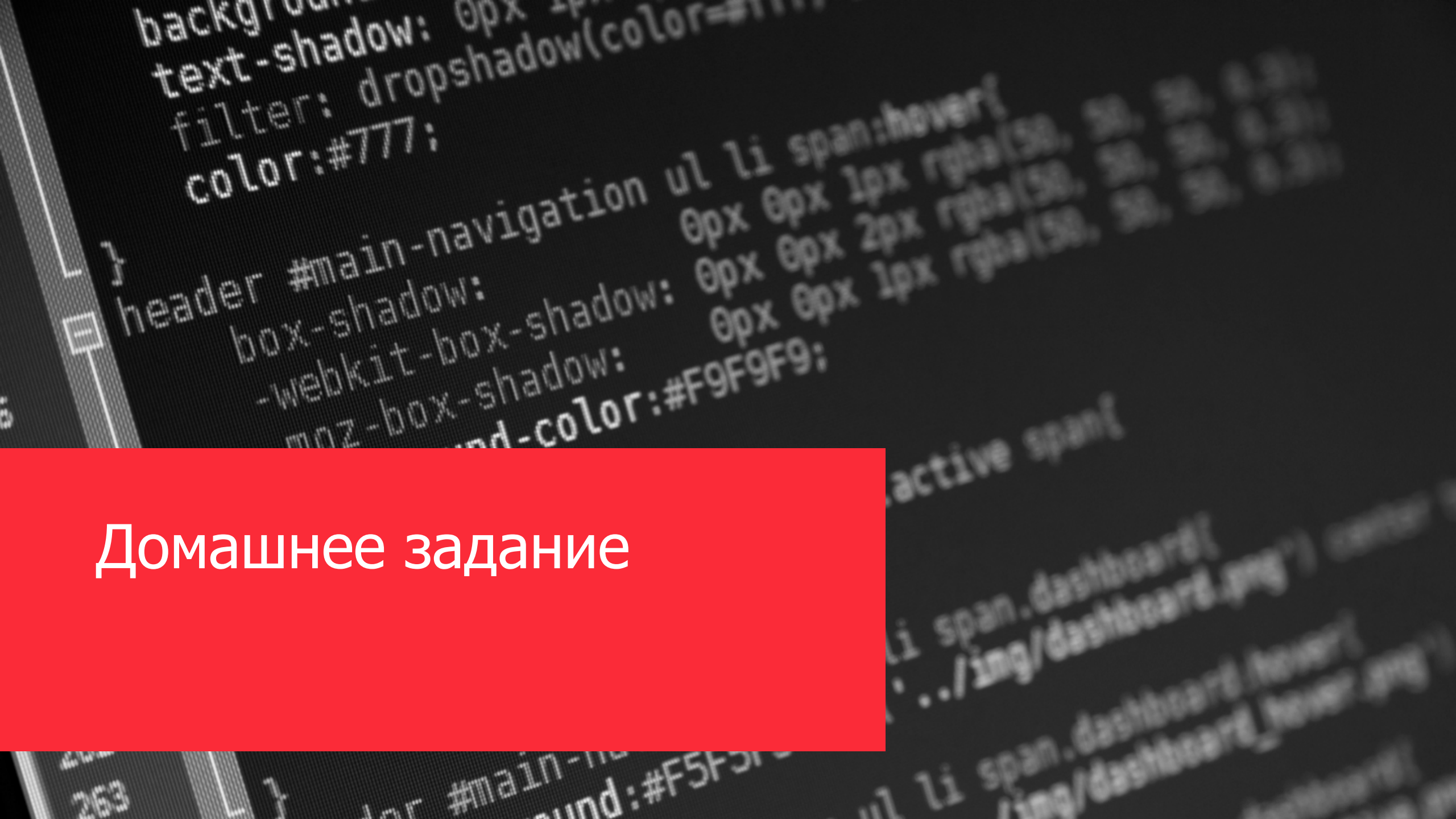
Locks

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

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Больше примеров использования:

- https://www.openmp.org/resources/tutorials-articles/





Домашнее задание