Advanced Computer Architecture

Thread Programming

Antoine Trouvé

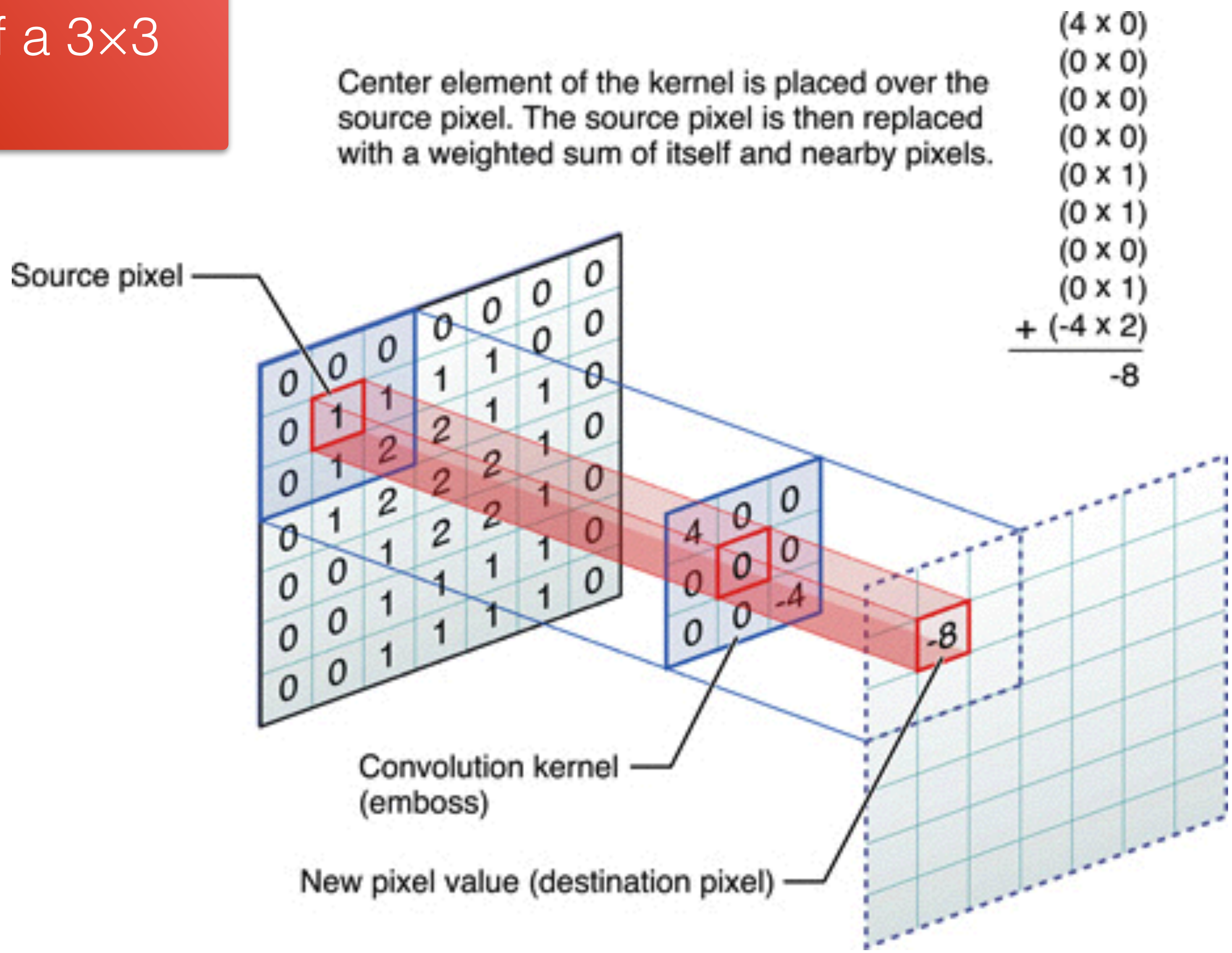
2015/06/01

Your very first **useful** program with Pthreads

Edge Detection Program

Principle

Case of a 3×3 matrix



Source: http://stats.stackexchange.com/questions/114385/what-is-the-difference-between-convolutional-neural-networks-restricted-boltzma

The Files

You can download a file from Linux command line with **wget.**

- The files are under **http://trouve.sakura.ne.jp/aca/img_kernel.step1**
  - **bmp.h**: prototypes of the utility functions
  - **bmp.c**: implementation of the utility functions
  - **img_kernel.step1.c**: the main function

Edge Detection Program Flow (in img_kernel.c)

Read the image file (format bmp) — read_bmp()

Creates a buffer image — variables **new_data** and **data** (defined in bmp.h)

Apply a convolution matrix (3×3) — apply_kernel_33() for 3×3 matrix

Saves the image — write_bmp()

How to Read/Write the Image File

In variable **info**

Format “bmp”

Header (54 bytes)

Pixels
(row major)

In variable **data**

Read: read_BMP()
Write: write_and_free_BMP()

Pixel (32 bits)

Red	Green	Blue	void
8 bits	8 bits	8 bits	8 bits

Always 0 in my images

The Serial Version of the Program

img_kernel.step1.c

```
#include "./bmp.h"

/* Edge detection */
double edge_kernel_matrix[3][3] = {
  {-1, -1, -1},
  {-1,  8, -1},
  {-1, -1, -1}
};

/* Identity */
double id_kernel_matrix[3][3] = {
  {0, 0, 0},
  {0, 1, 0},
  {0, 0, 0}
};

/* blur */
double blur_kernel_matrix[5][5] = {
  {1.0/273.0, 4.0/273.0, 7.0/273.0, 4.0/273.0, 1.0/273.0},
  {4.0/273.0, 16.0/273.0, 26.0/273.0, 16.0/273.0, 4.0/273.0},
  {7.0/273.0, 26.0/273.0, 41.0/273.0, 26.0/273.0, 7.0/273.0},
  {4.0/273.0, 16.0/273.0, 26.0/273.0, 16.0/273.0, 4.0/273.0},
  {1.0/273.0, 4.0/273.0, 7.0/273.0, 4.0/273.0, 1.0/273.0}
};

int main(int argc, char* argv[]) {
  if(argc!=3) { printf("Please specify the names of the input and output files in parameters:\n\t  %s
<input.bmp> <output.bmp>\n", argv[0]); exit(-1); }

  printf("Reads the file %s ...\n", argv[1]);

  unsigned char info[54];
  /* Reads the file and allocates the data in the heap */
  data = read_bmp(argv[1], info);

  if(data==NULL) { printf("Unable to open the file. Exit...\n"); return -1; }

  /* Does some stuff */
  printf("Applies the kernel ...\n");

  // extracts image height and width from header
  int width  = BMP_WIDTH(info);
  int height = BMP_HEIGHT(info);

  new_data = malloc(width*height*sizeof(bmp_pixel_t));

  applyMatrix_33((unsigned char*)BMP_PIXEL(data,0,1,width), (unsigned char*)BMP_PIXEL(new_data,
0,1,width), width, height-1, edge_kernel_matrix);

  printf("Writes the output file in %s ...", argv[2]);

  /* Writes the BMP to a file and frees the data from the heap */
  if(write_bmp(argv[2], new_data, info)==-1) {
    printf("Unable to write the file. Exit...\n"); return -1;
  }

  free(data);
  free(mew_data);

  return 0;
}
```

The main function only

```c
#include "./bmp.h"

/* Edge detection */
double edge_kernel_matrix[3][3] = {
  {-1, -1, -1},
  {-1,  8, -1},
  {-1, -1, -1}
};

/* Identity */
double id_kernel_matrix[3][3] = {
  {0, 0, 0},
  {0, 1, 0},
  {0, 0, 0}
};

/* blur */
double blur_kernel_matrix[5][5] = {
  {1.0/273.0, 4.0/273.0, 7.0/273.0, 4.0/273.0, 1.0/273.0},
  {4.0/273.0, 16.0/273.0, 26.0/273.0, 16.0/273.0, 4.0/273.0},
  {7.0/273.0, 26.0/273.0, 41.0/273.0, 26.0/273.0, 7.0/273.0},
  {4.0/273.0, 16.0/273.0, 26.0/273.0, 16.0/273.0, 4.0/273.0},
  {1.0/273.0, 4.0/273.0, 7.0/273.0, 4.0/273.0, 1.0/273.0}
};

int main(int argc, char** argv) {
  if(argc!=3) { printf("Please specify the names of the input and output files in parameters:\n\t  %s
<input.bmp> <output.bmp>\n", argv[0]); exit(-1); }

  printf("Reads the file %s ...\n", argv[1]);

  unsigned char info[54];
  /* Reads the file and allocates the data in the heap */
  unsigned char* data = read_BMP(argv[1], info);

  if(data==NULL) { printf("Unable to open the file. Exit...\n"); return -1; }

  /* Does some stuff */
  printf("Applies the kernel ...\n");

  // extracts image height and width from header
  int width  = BMP_WIDTH(info);
  int height = BMP_HEIGHT(info);

  unsigned char* new_data = malloc(width*height*sizeof(bmp_pixel_t));

  applyMatrix_33((unsigned char*)BMP_PIXEL(data,0,1,width), (unsigned char*)BMP_PIXEL(new_data,
0,1,width), width, height-1, edge_kernel_matrix);

  printf("Writes the output file in %s ...", argv[2]);

  /* Writes the BMP to a file and frees the data from the heap */
  if(write_and_free_BMP(argv[2], new_data, info)==-1) {
    printf("Unable to write the file. Exit...\n"); return -1;
  }

  free(data);

  return 0;
}
```

The Serial Version of the Program

img_kernel.c

Pre-defined kernel matrix

Reads the bmp file

Creates a new image buffer

Applies the matrix

Writes the output file and frees data

Compile / Link / Execute

① Compile and link the program

```
$> gcc img_kernel.step1.c bmp.c -lpthread –o img_kernel.step1.out
```

② Execute

```
$> ./img_kernel.step1.out ~/examples/img/afghan.bmp afghan.out.bmp
```

afghan.bmp

afghan.out.bmp

You can use a smaller image if the network is slow:
http://trouve.sakura.ne.jp/aca/afghan.small.bmp

Exercise 1

- Compile and execute the program. Try with afghan.bmp.
- Try with other matrices, maybe your own !

Exercise 2

- Modify the program so that it applies function **remove_red** before edge detection. Call it **img_kernel.step2**

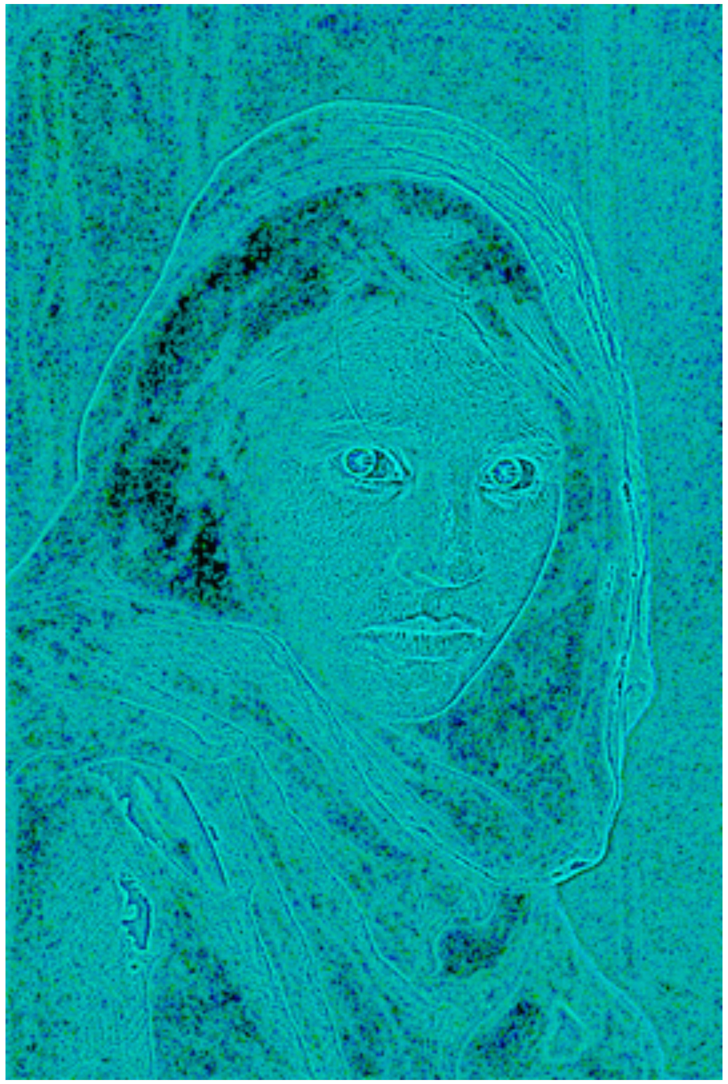
```
void remove_red(
  int width,
  int height,
  int starty);
```

The width/height of the image

From which line to start to apply the filter (0 for now)

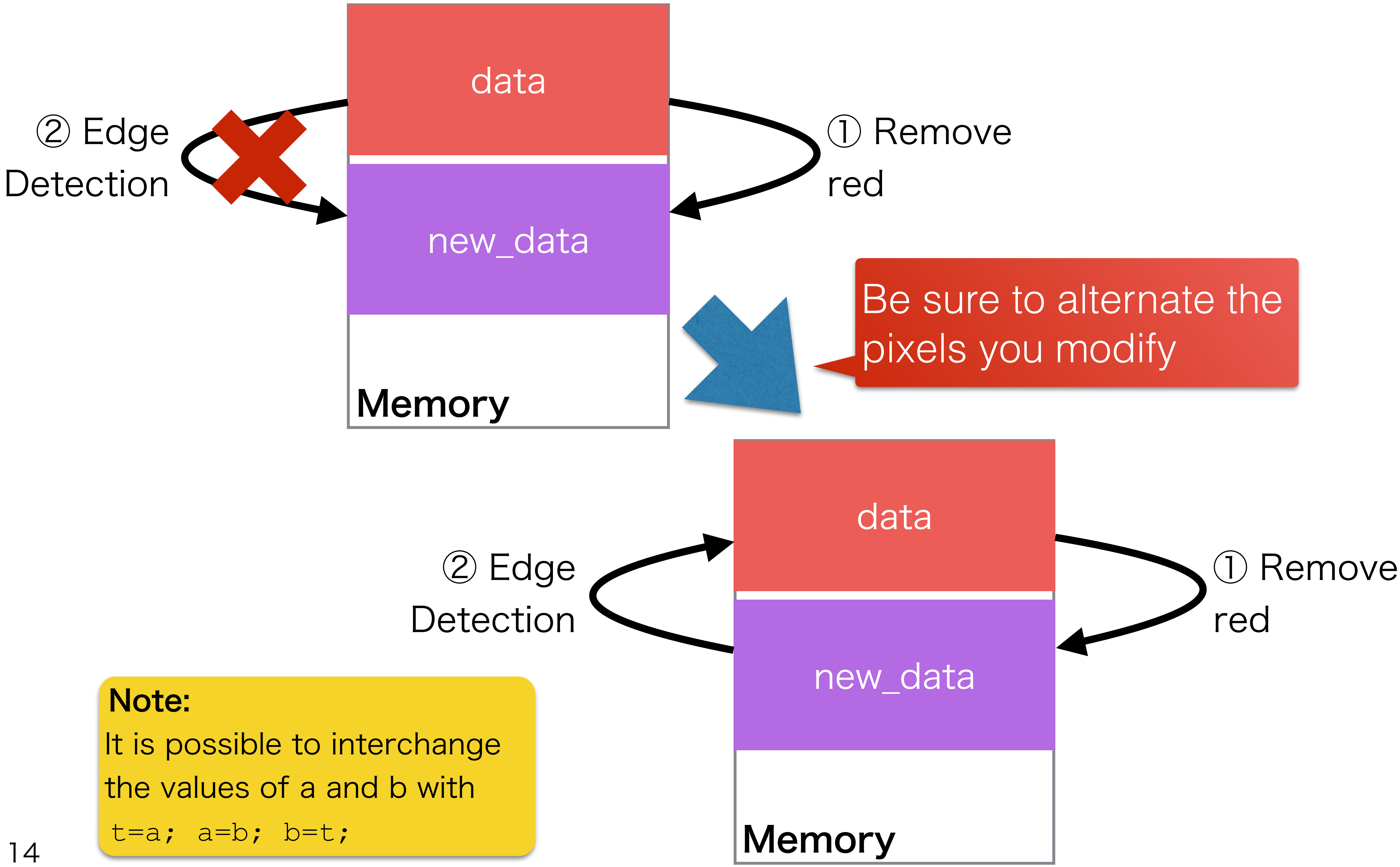
Check your output

Before



After

Hint for Exercise 2



Note:
It is possible to interchange the values of a and b with
`t=a; a=b; b=t;`

Let us Parallelize it

Reminder of a Simple Pthread Program

```
#include<stdio.h>    // printf()
#include<unistd.h>   // sleep()
#include<string.h>   // strerror(char*)
#include<pthread.h>

void* doSomeThing(void *arg)
{
  return NULL;
}

int main(void)
{
  int err;
  pthread_t thread;

  err = pthread_create(&thread, NULL, &doSomeThing, NULL);
  if (err != 0) {
    printf("\ncan't create thread :[%s]\n", strerror(err));
  }

  pthread_join(thread, NULL);

  return 0;
}
```

Create a variable to store information about the thread

Creates a thread, executing a given function

Wait for the child thread to finish

Affect a Rank to Thread

- It is common to affect a number to threads in order to identify them
  - We call it the **rank** of a thread
- In pthread, there is no automatic way to get the rank of a thread
- You can do it manually in pthread by passing arguments to the thread function

I use **unitptrt_t**, it could be **int**

```
void* do_thread(void* arg) {
   unitptr_t tid = (int)arg;
   [...]
   return NULL;
}

int main(void) {
   pthread_t thread;
   unitptr_t tid=0;
   pthread_create(
      &thread,
      NULL,
      &doSomeThing,
      (void*)tid);
   [...]
   return 0;
}
```

About Measuring Execution Time

- One of the major motivation for parallelization is performance improvement:

$$speedup = \frac{execution\ time\ after\ parallelization}{execution\ time\ before\ parallelization}$$

- There are two ways to measure execution time
  - **Wall clock time:** the actual time spent
  - **CPU time:** the amount of time the CPU was actually making calculations. If threads are used, sums up all the time spent in all threads
- We are interested in **wall clock time**.

Measuring Execution Time in Linux

Add **time** before the command to measure

```
$> time ./img_kernel.step3.out ~/examples/img/afghan.bmp afghan.out.bmp
```

Wall clock time

```
real 0m0.722s
user 0m1.326s
sys  0m0.032s
```

CPU time

Exercise 3

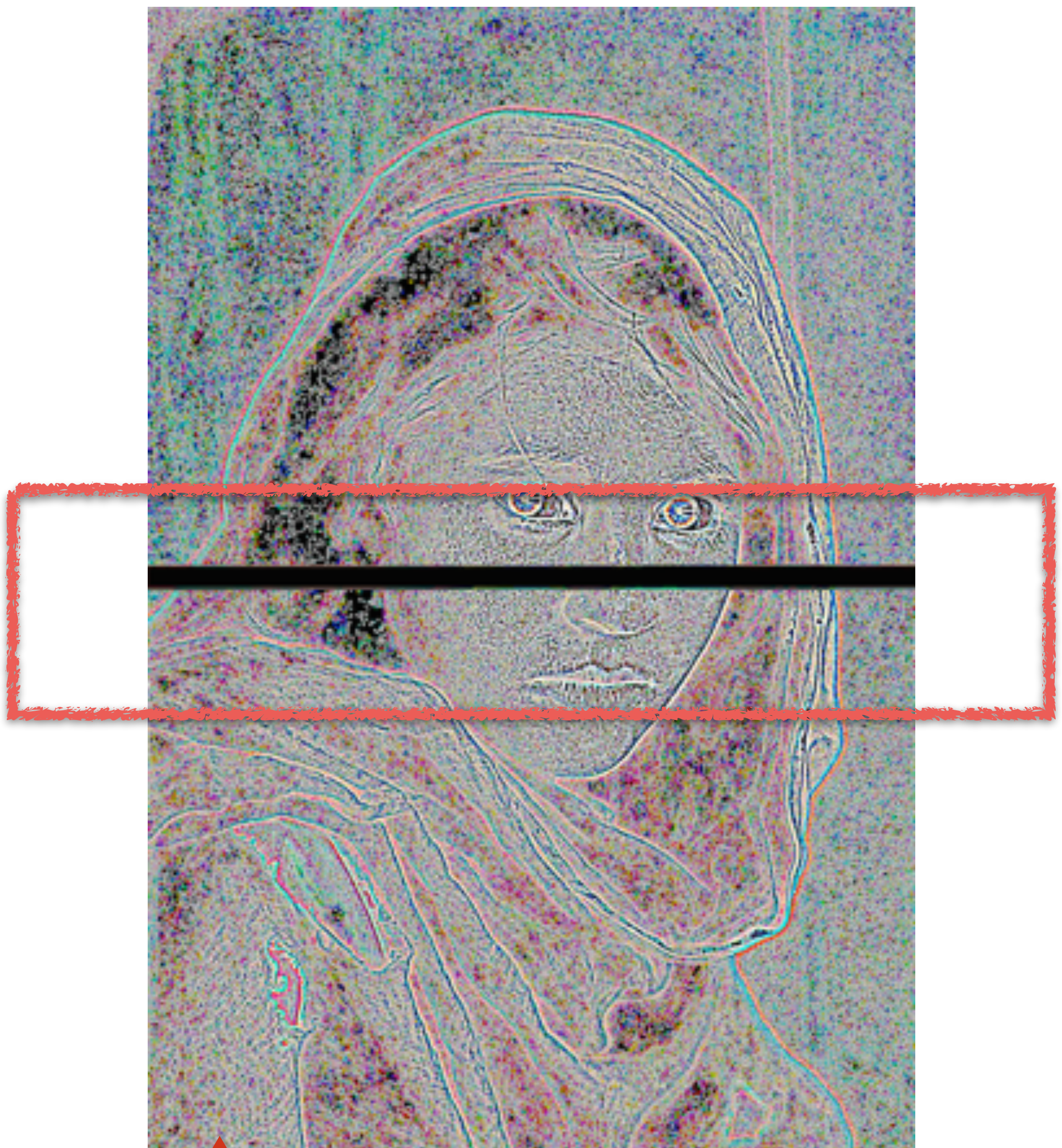
You can get step2 from **http://trouve.sakura.ne.jp/aca/img_kernel.step2** if you're not confident with yours

- Modify the program so that it executes with two worker threads. Use data-parallelism. Call it **img_kernel.step3**

- Is it faster than **img_kernel.step2** ?

Note:
・A possibility for thread to share variables is to use globals.
・Don't worry too much about artifacts

Who gets That ?



If you don't, force a thread to wait a little with a sleep()

Who gets That ?

Global variable mess

+

Race condition

If you don’t, force a thread to wait a little with a sleep()

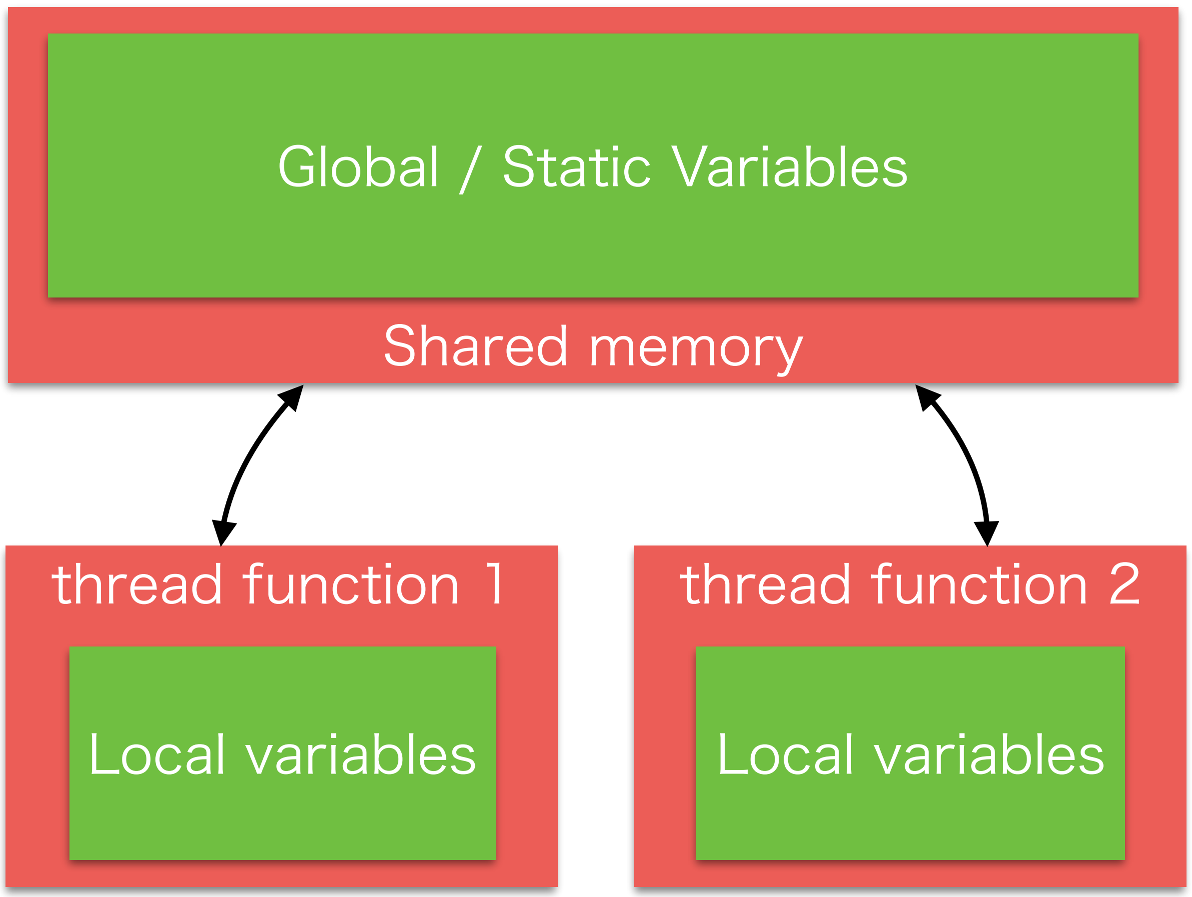
Global Variable Mess

```
void* do_thread(void* arg) {
  uintptr_t tid = (int)arg;

  int theight = height/nb_threads;
  int starty = theight*tid+1;
  if(tid==nb_threads-1) { theight -= 2; }

  remove_red(width, theight, starty);

  tmp_data = new_data;
  new_data = data;
  data = tmp_data;

  applyMatrix_33(width, theight, starty, edge_kernel_matrix);

  return NULL;
}
```

Modify global pointers inside the thread

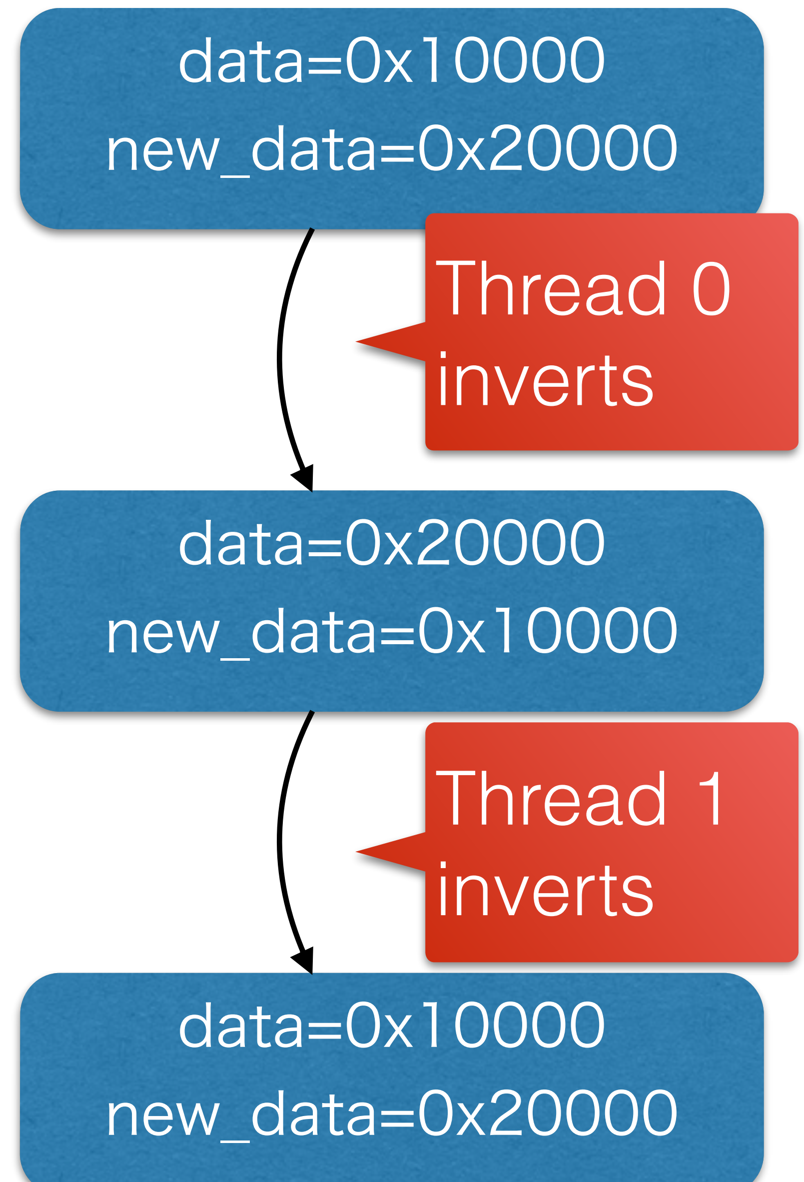
Reminder: The memory Model

- All threads share the same memory space
- All threads can access
  - The global variables
  - The memory dynamically allocated
- The can even access other thread's stack with pointers
  - This is often not a good idea



Global Variable Mess

- The problem
  - All threads share the global variables
  - Any modification in one thread impacts the other ones
- In our example
  - Both threads share **data** and **new_data**
  - Both threads invert **data** and **new_data**, that is, we invert two times
- Possible fix
  - **Only invert once** — We use this one
  - Use local variables



Exercise

Modify **img_kernel.step3.c** so that it only inverts the variables once

```
remove_red(…);

tmp_data = new_data;
new_data = data;
data = tmp_data;

applyMatrix_33(…);
```

```
remove_red(…);

if(tmp_data!=data) {
  tmp_data = new_data;
  new_data = data;
  data = tmp_data;
}

applyMatrix_33(…);
```

But the output is still strange…

Race Condition

- A race might condition occurs when one thread writes a data read by another one
- If we do not use any synchronization, we do not know in which order the read / write occurs

Example of Race Condition

Thread 0

```
remove_red(…);
tmp_data = new_data;
data = tmp_data;
data = tmp_data;
applyMatrix_33(…);
```

Thread 1

```
remove_red(…);
applyMatrix_33(…);
```

time

At this point
- thread 0 has modified **data**
- thread 1 is still running **remove_red**

In this time segment, remove_red is reading from the wrong data buffer: this is why you get the **black bar**.

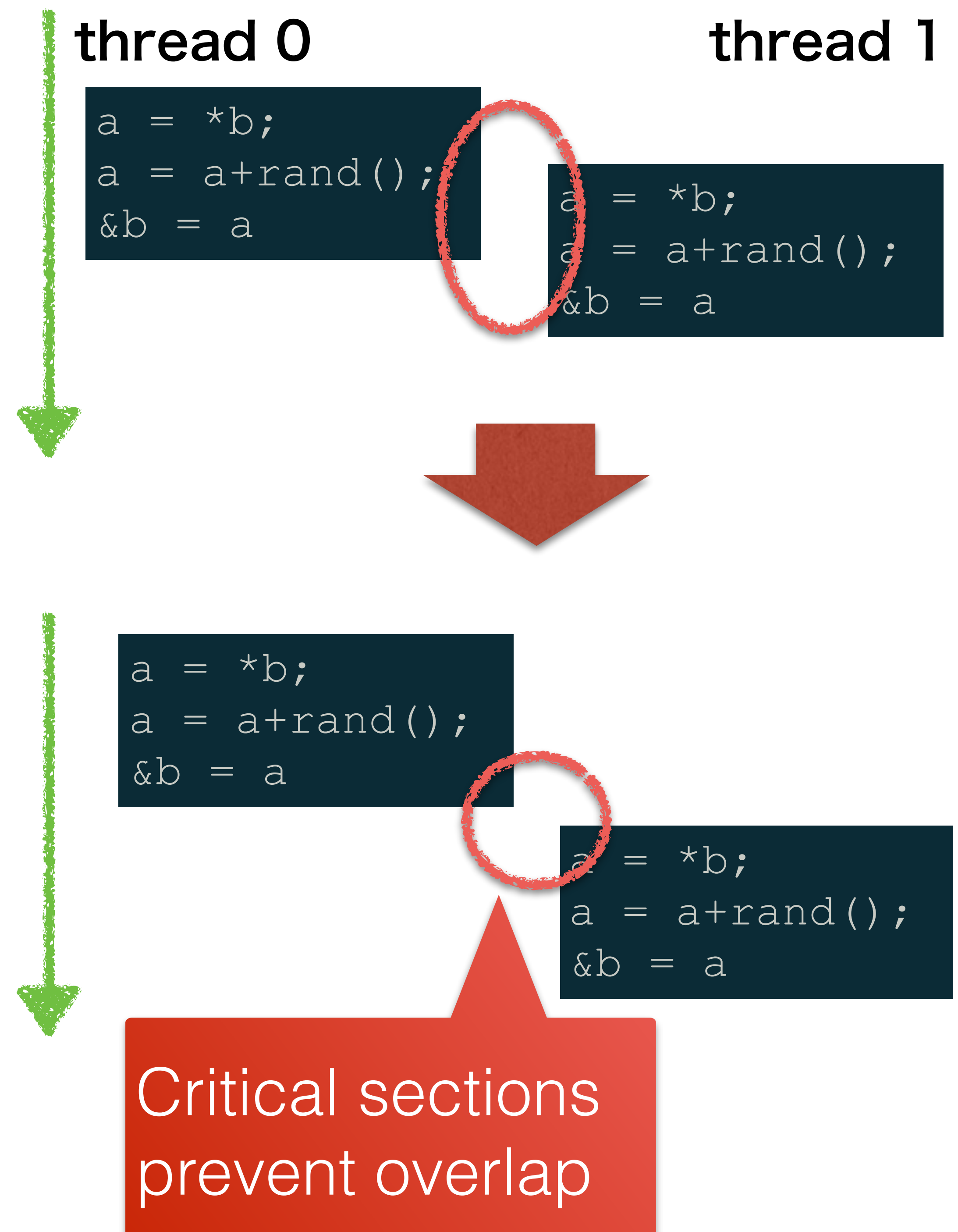
Thread Synchronization

What is Thread Synchronization

- A pool of **techniques** to control the order in which instructions in threads are executed
- We will look at two kinds of synchronization
  - Critical sections
  - Barriers
- Those are often implemented using
  - Mutex
  - Semaphore

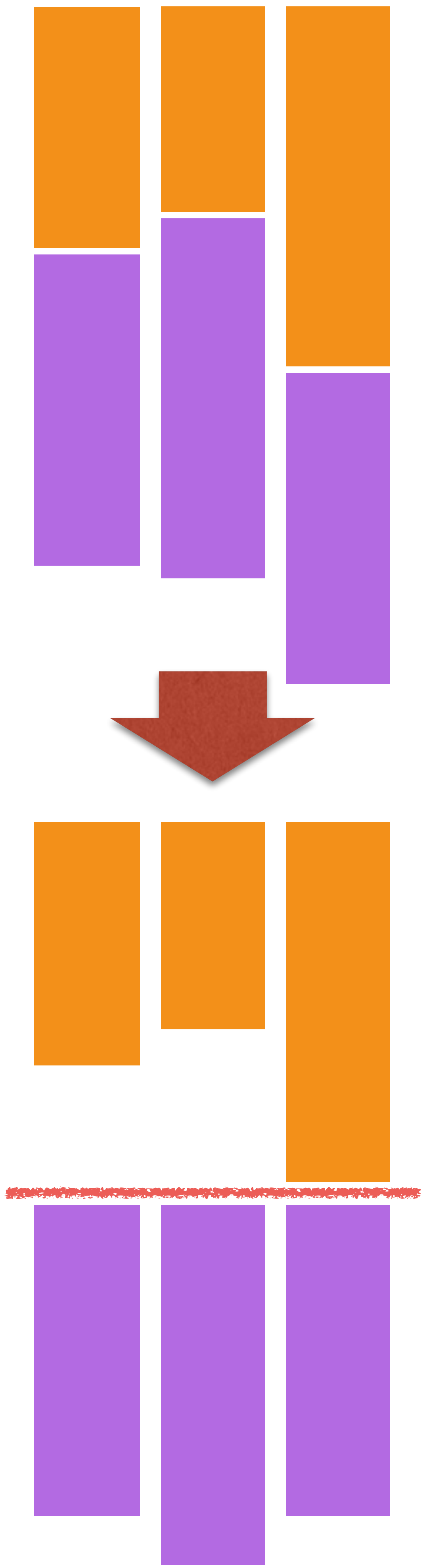
Critical Sections

- Critical sections make sure that at any time, at most one instance of a given code is under execution
- Some also use the term atomic sections
- There are no guarantee in which order critical sections are executes
- A critical section may execute concurrently to a non-critical one



Barrier

- A barrier is another method to synchronize threads
  - We say a thread ”hits” a barrier
- A barrier consists in making all the threads wait at a given point in the code
- There is no guarantee of the order in which threads hit a barrier



Mutual Exclusion Locks

- One can protect critical sections with mutual exclusion lock, or mutex
- A mutex is a resource that should be
  - Locked at the beginning of the critical section
  - Unlocked at the end of it
- If a thread tries to reserve a mutex already locked, it waits until it is unlocked
- Only the thread that locked a mutex can unlock it

Protect a Critical Section with a Mutex

```
some code
```

```
lock mutex
some code
unlock mutex
```

- Lock/unlock the mutex at the beginning/end of the critical section
- Any thread that try to lock the mutex before it unlock will have to wait

"Hidden" Race Condition

```
int nextId = 0;

void* do_thread(void* arg) {
  int myId = nextId++;

  [...]
}
```

This simple code affects an id to a thread

It is compiled in three assembly instructions

```
read myId from memory at &nextId
myId = myId + 1
write myId to memory at &nextId
```

- The above is one line of C code
  - It aims at affecting a different id (in myId) to threads
  - We want myId to be different for each thread
- Problem:
  - It is **not atomic**: two threads may end up with the same value in **myId**.

Example: if two threads run this way

Thread 0

```
read myId from memory at &nextId
```

```
myId = myId + 1
```

```
write myId to memory at &nextId
```

myId = 1

Thread 1

```
read myId from memory at &nextId
```

```
myId = myId + 1
```

```
write myId to memory at &nextId
```

myId = 2

No problem

Example: but if two threads run this way

Thread 0

```
read myId from memory at &nextId
myId = myId + 1
write myId to memory at &nextId
```

myId = 1

Thread 1

```
read myId from memory at &nextId
myId = myId + 1
write myId to memory at &nextId
```

myId = 1

Thread 1 reads before thread 0 writes !

The threads have the same value !

Quizz

How would you use mutex to fix this code so that all threads have a different myId ?

```
int nextId = 0;

void* do_thread(void* arg) {
  int myId = nextId++;

  [...]
}
```

(
```
read myId from memory at &nextId
myId = myId + 1
write myId to memory at &nextId
```
)

Semaphore

- A semaphore is similar to a mutex
  - They are resources that can be locked and unlocked
  - They allow to make thread to wait for events
- However, there are differences
  - They can be locked several times
  - They can by unlocked by any thread, not only the one that locked it

Principle of a Semaphore

- A semaphore contains a positive integer value
- The value is decreased upon lock
- The value is increased upon unlock
- When a thread tries to unlock a semaphore at 0, it waits until some other thread unlocks it once.
- It possible to initialize it at any value, for example:
  - **0**. the semaphore needs to be unlocked
  - **1**. the semaphore can be locked once
  - **n**. the semaphore can be locked n times consecutively

Mutex in Pthread

- Mutex variable (stores state)

```
pthread_mutex_t mutex;
```

- Lock a mutex (or wait if already locked)

```
pthread_mutex_lock(&mutex);
```

- Unlock a mutex

```
pthread_mutex_unlock(&mutex);
```

Quizz

- How would you implement a barrier with a **mutex** ?

Quizz (Solution)

Put in a critical section !

Count the number of threads that hit the barrier

Wait until all the threads have hit the barrier

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;

void* do_thread(void* arg) {
  […]

  pthread_mutex_lock(&mutex);
  barrier_count++;
  pthread_mutex_unlock(&mutex);

  while(barrier_count!=nb_threads);
  […]
}
```

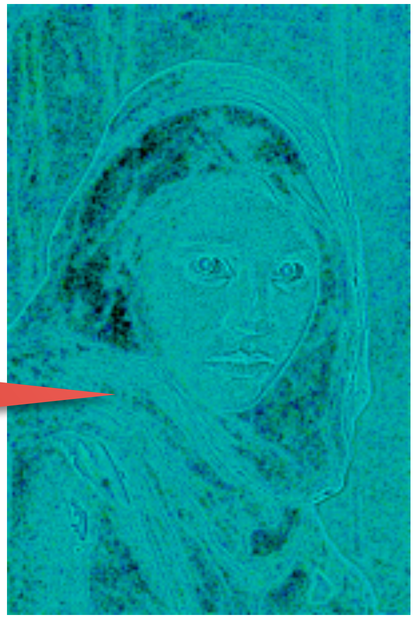
This is called **busy wait**

Exercise

- Adds barriers to **img_kernel.step3.c** to make it work. Call it **img_kernel.step4.c**

You need two barriers

What you should get



Problem with Busy Wait

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;

void* do_thread(void* arg) {
  […]

  pthread_mutex_lock(&mutex);
  barrier_count++;
  pthread_mutex_unlock(&mutex);

  while(barrier_count!=nb_threads);
  […]
}
```

- The while loop keeps the processor busy
- This approach **can be very inefficient** if:
  - The thread has to wait for a long time
  - Two threads are running on the same processor

Semaphore in C

Semaphore are not part of Pthread.
On need to include “semaphore.h”

- Semaphore variable (stores state)

```
sem_t sem;
```

- Initialize a semaphore to n

```
sem_init(&sem, 0, n);
```

- Lock a semaphore (or wait if already locked)

```
sem_wait(&sem);
```

- Waits for the semaphore to be ≠0
- Decrements it

- Unlock a semaphore

```
sem_post(&sem);
```

- Decrements a semaphore

Quizz

- How would you implement a barrier with a **semaphore** ?

Quizz (Solution)

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;
sem_t semaphore;

sem_init(semaphore, 0, 0)

void* do_thread(void* arg) {
  […]

  pthread_mutex_lock(&mutex);
  if(barrier1_count==nb_threads-1) {
    pthread_mutex_unlock(&mutex);
    for(j=0;j<nb_threads-1; j++) { sem_post(&semaphore); }
  } else {
    barrier1_count++;
    pthread_mutex_unlock(&mutex);
    sem_wait(&semaphore);
  }

  […]
}
```

Quizz (Solution)

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;
sem_t semaphore;

sem_init(semaphore, 0, 0)

void* do_thread(void* arg) {
  [...]

  pthread_mutex_lock(&mutex);
  if(barrier1_count==nb_threads-1) {
    pthread_mutex_unlock(&mutex);
    for(j=0;j<nb_threads-1; j++) { sem_post(&semaphore); }
  } else {
    barrier1_count++;
    pthread_mutex_unlock(&mutex);
    sem_wait(&semaphore);
  }

  [...]
}
```

① Variables:
- **barrier_count**: the number of threads that hit the barrier
- **mutex**: the mutex to protect read/write of barrier_count
- **semaphore**: the barrier

② Initializes the semaphore at 0 (locked)

Quizz (Solution)

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;
sem_t semaphore;

sem_init(semaphore, 0, 0)

void* do_thread(void* arg) {
  [...]

  pthread_mutex_lock(&mutex)
  if(barrier1_count==nb_threads-1) {
    pthread_mutex_unlock(&mutex);
    for(j=0;j<nb_threads-1; j++) { sem_post(&semaphore); }
  } else {
    barrier1_count++;
    pthread_mutex_unlock(&mutex);
    sem_wait(&semaphore);
  }

  [...]
}
```

③ Checks and update the barrier counter:
- protect with a mutex
- all threads have hit the barrier if barrier_count = nb_threads-1 (in this case, do not update the counter)

Quizz (Solution)

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;
sem_t semaphore;

sem_init(semaphore, 0, 0)

void* do_thread(void* arg) {
  […]

  pthread_mutex_lock(&mutex);
  if(barrier1_count==nb_threads-1) {
    pthread_mutex_unlock(&mutex);
    for(j=0;j<nb_threads-1; j++) { sem_post(&semaphore); }
  } else {
    barrier1_count++;
    pthread_mutex_unlock(&mutex);
    sem_wait(&semaphore);
  }

  […]
}
```

④ If not all threads have hit the barrier, wait for it to be unlocked with a sem_post.
Note: sem_wait blocks because the semaphore has been initialized to 0

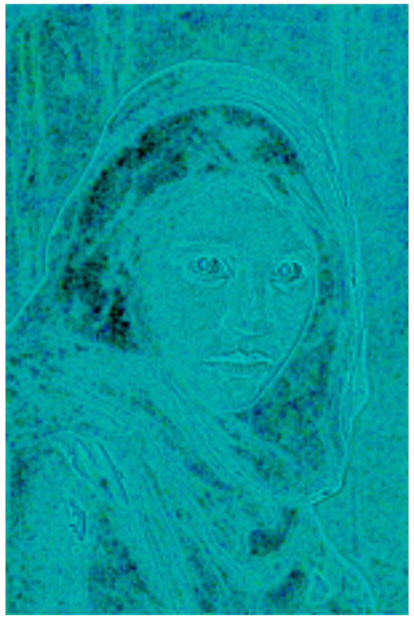
Quizz (Solution)

```
int nb_threads = 2;
int barrier_count = 0;
pthread_mutex_t mutex;
sem_t semaphore;

sem_init(semaphore, 0, 0)

void* do_thread(void* arg) {
  […]

  pthread_mutex_lock(&mutex);
  if(barrier1_count==nb_threads-1) {
    pthread_mutex_unlock(&mutex);
    for(j=0;j<nb_threads-1; j++) { sem_post(&semaphore); }
  } else {
    barrier1_count++;
    pthread_mutex_unlock(&mutex);
    sem_wait(&semaphore);
  }

  […]
}
```

⑤ Releases the barrier. Need to release it once for each waiting thread.

Exercise

- Modify **img_kernel.step4.c** to use a semaphore instead of a busy wait. Call the program **img_kernel.step5.c**.



Problems of Mutex / Semaphores

Deadlock (Example)

- Let us consider this program

	Thread 0	Thread 1
t=0	`pthread_mutex_lock(&mutex1);`	`pthread_mutex_lock(&mutex2);`
t=1	`pthread_mutex_lock(&mutex2);`	`pthread_mutex_lock(&mutex1);`

What happens ?

Deadlock (Example)

- Let us consider this program

	Thread 0	Thread 1
t=0	`pthread_mutex_lock(&mutex1);`	`pthread_mutex_lock(&mutex2);`
t=1	`pthread_mutex_lock(&mutex2);`	`pthread_mutex_lock(&mutex1);`

Thread 0 waits for thread 1

Thread 1 waits for thread 0

Problem 1: Deadlock

- A dead lock occurs when two processes wait for each other
- It results in both threads to **wait forever**
- Deadlocks are often very hard to detect in programs

Exercise

- Implement a simple Pthread program that creates a deadlock with two threads and two mutex

Problem 2: Serialization

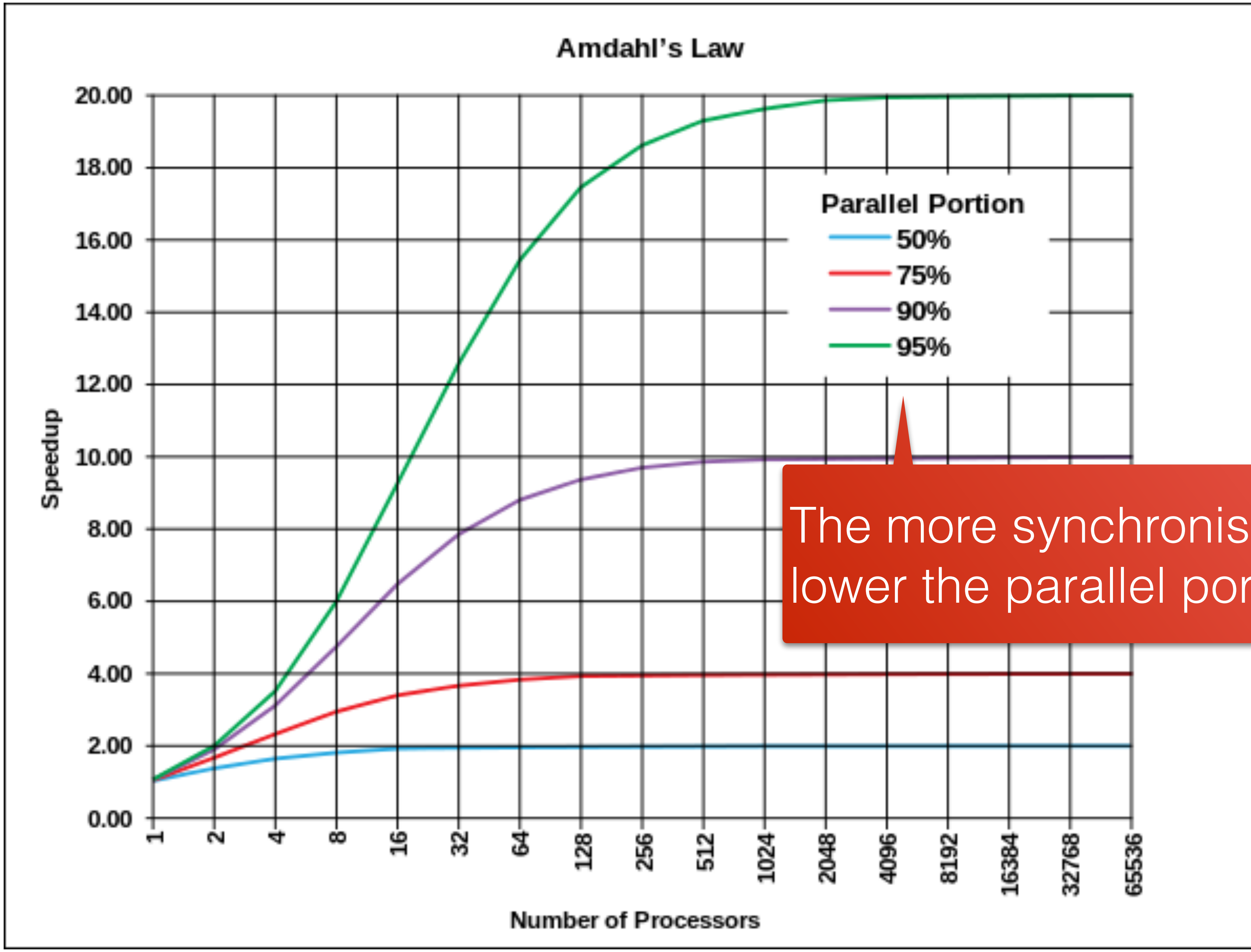
- Code protected by critical sections and barriers cannot be executed in parallel anymore: they are serial
- This reduces the “amount of parallelism” of a program, therefore, the performance

Amdahl’s Law: the speedup of parallel program is limited by its serial components. The speedup of a parallel program can be calculated as:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

N: number of threads

P amount of the program that can execute in parallel



The more synchronisation, the lower the parallel portion gets

http://ja.wikipedia.org/wiki/アムダールの法則

Rule of the Thumb

- Avoid as most as possible any synchronization in programs because:
  - It reduces performance
  - It raise the probability of bug
- Most of the time, it boils down to **avoiding to write in global variable**
- A common mean to achieve this is to **use local variables**
  - Copy the content of global variables into local ones
  - Modify the local ones only
- It is however not possible every time

Exercise

- Remove the need for any synchronization from **img_kernel.step5.c** by using local variables. Call it **img_kernel.step6.c**.

You will need to modify the functions **remove_red** and **apply_matrix33**

Quizz

- Is it possible to remove synchronization in the code below ?

```
int nextId = 0;
pthread_mutex_t mutex;

void* do_thread(void* arg) {
  pthread_mutex_lock(&mutex);
  int myId = nextId++;
  pthread_mutex_unlock(&mutex);

  […]
}
```

Simple Threading with OpenMP

What is OpenMP

- Pthread is hard to use
  - It require a lot of extra code (compared to a sequential program)
  - One need to implement by hand even common threading patterns
- OpenMP aims at reducing the amount of extra code, especially for simple threading patterns such as:
  - Data parallelism
  - Barriers
  - Critical section
- It consists of
  - Compiler directives
  - A library
  - Some environment variables

You need a specific compiler !

Data Parallelism in OpenMP

Sequential Program

```
#include <stdio.h>

int main() {
 printf("Hello");

  return 0;
}
```

OpenMP Program

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

int main() {
  int ID;

  #pragma omp parallel num_threads(10)
  {
    ID=omp_get_thread_num();
    printf("Hello %d\n", ID);
  }

  return 0;
}
```

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

int main() {
  int i, ID;

  #pragma omp parallel num_threads(10)
  {
    ID=omp_get_thread_num();
    printf("Hello %d\n", ID);
  }

  return 0;
}
```

Defines the functions of OpenMP

OpenMP compiler directives

Ask to execute the block after in parallel with 10 threads

Example of OpenMP function

Gets the rank of the current thread (from 0 to 9 in here)

No need to join: there is an implicit barrier at the end of the block

Parallelize a `for` Loop

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

int main() {
  int i, ID;

  #pragma omp parallel for num_threads(10)
  for(i=0; i<20; i++) {
    ID=omp_get_thread_num();
    printf("Hello %d -> %d\n", ID, i);
  }

  return 0;
}
```

Distribute the 20 iterations of the loop in 10 threads

Exercise

- Write, compile and execute the programs of the two previous slides

- Note to compile, use the flag "-fopenmp" in the command line

gcc understands OpenMP

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

int main() {
  int ID;

  #pragma omp parallel num_threads(10)
  {
    ID=omp_get_thread_num();
    printf("Hello %d\n", ID);
  }

  return 0;
}
```

```
student@ip-ac1f162f:~/examples/openmp$ ./openmp1.out
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Hello 6
Hello 7
Hello 8
Hello 0
Hello 9
```

The execution order is not predictable !

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

int main() {
  int i, ID;

  #pragma omp parallel for num_threads(10)
  for(i=0; i<20; i++) {
    ID=omp_get_thread_num();
    printf("Hello %d -> %d\n", ID, i);
  }

  return 0;
}
```

```
student@ip-ac1f162f:~/examples/openmp$ ./openmp2.out
Hello 2 -> 4
Hello 2 -> 5
Hello 1 -> 2
Hello 1 -> 3
Hello 4 -> 8
Hello 4 -> 9
Hello 6 -> 12
Hello 6 -> 13
Hello 8 -> 16
Hello 8 -> 17
Hello 0 -> 0
Hello 0 -> 1
Hello 3 -> 6
Hello 3 -> 7
Hello 7 -> 14
Hello 7 -> 15
Hello 9 -> 18
Hello 5 -> 10
Hello 5 -> 11
Hello 9 -> 19
```

Synchronization with OpenMP

- It is possible to express most synchronization techniques in OpenMP

Critical Sections

```
#pragma omp parallel for num_threads(10)
  for(i=0; i<20; i++) {
    ID=omp_get_thread_num();
    printf("Hello %d -> %d\n", ID, i);
    #pragma omp critical
    {
      printf("Critical section\n");
    }
    printf("End\n");
  }
```

Barriers

```
#pragma omp parallel num_threads(10)
  {
    ID=omp_get_thread_num();
    printf("Hello %d\n", ID);
    #pragma omp barrier
    printf("Good bye %d\n", ID);
  }
```

Variable Sharing with OpenMP

- It is possible to define variables as shared and private explicitly
    - private(x,y): make x and y private to threads
    - shared (x,y): make x and y shared between threads
- By default, variables are shared

Exercise

You can get it from http://trouve.sakura.ne.jp/aca/img_kernel.step2

- Parallelize **img_kernel.step2.c** with OpenMP using barriers. Call it **img_kernel.step7.c**

Major Design Patterns in Thread Programming

When do you **spawn** threads ?

How do you **divide** the work between threads ?

How do you **structure** your program ?

Thread Creation

- **Static** threads
  - The program creates a finite number of threads at startup
  - The programs give tasks to available threads
  - If no thread is available, waits for one to be ready
  - The thread sleeps when the task ends
- **Dynamic** threads
  - The programs spawns a thread for each task
  - The thread dies when the task ends

Divide the Work among Threads

Data and Task Parallelism

The one we used

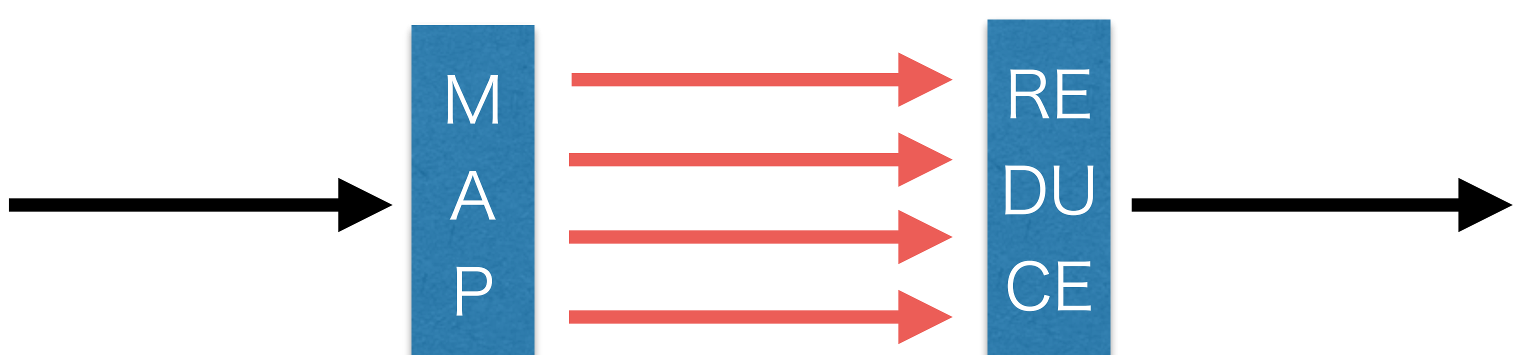
	Data Parallelism	Task Parallelism
Do the thread execute the same code ?	Same code	Different code
Do the thread take same inputs ?	Different inputs	Same inputs (or the output of another thread)

Program Structures

- Libraries like pThread allows to implement any parallel structure
  - But complex parallel programs tend to be hard to maintain
  - Deadlock and performance problems are often hidden behind complexity
- Therefore, people tend to stick to simple structures, the major two being
  - **Map reduce**
  - **Producer / consumer**
- The formalism of these simple techniques has contributed to the popularity of parallel programming (e.g. Hadoop)

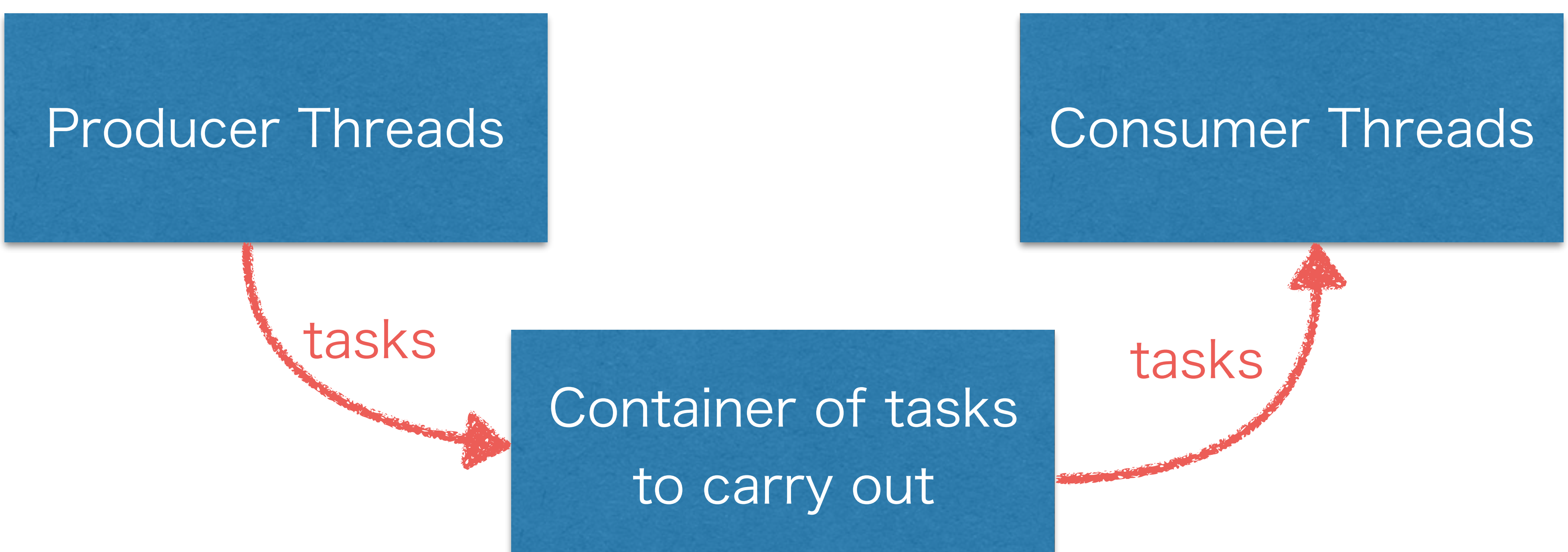
Map Reduce

- Map reduce is a very simple design pattern, in two steps
- **Map** step
  - Many threads are spawn at the same time using data parallelism
  - The input data are mapped to the threads (using more-or-less complex patterns)
- **Reduce** step
  - Upon termination, the output of each thread is gathered
  - All the workers are often synchronized with a barrier



Producer / Consumer

- Producer / Consumer is a simple design patterns that allows to distribute work among threads
- It is articulated around two types of threads (non-synchronous)
  - The **producer** ones: create work to do
  - The **consumer**: pick up work to do
- The “work to do” is stored in a data structure in shared memory
  - Protected by a mutex or a semaphore



Homework
(if you feel like it)

- Implement the image kernel application with a consumer / producer design pattern

Word of the end

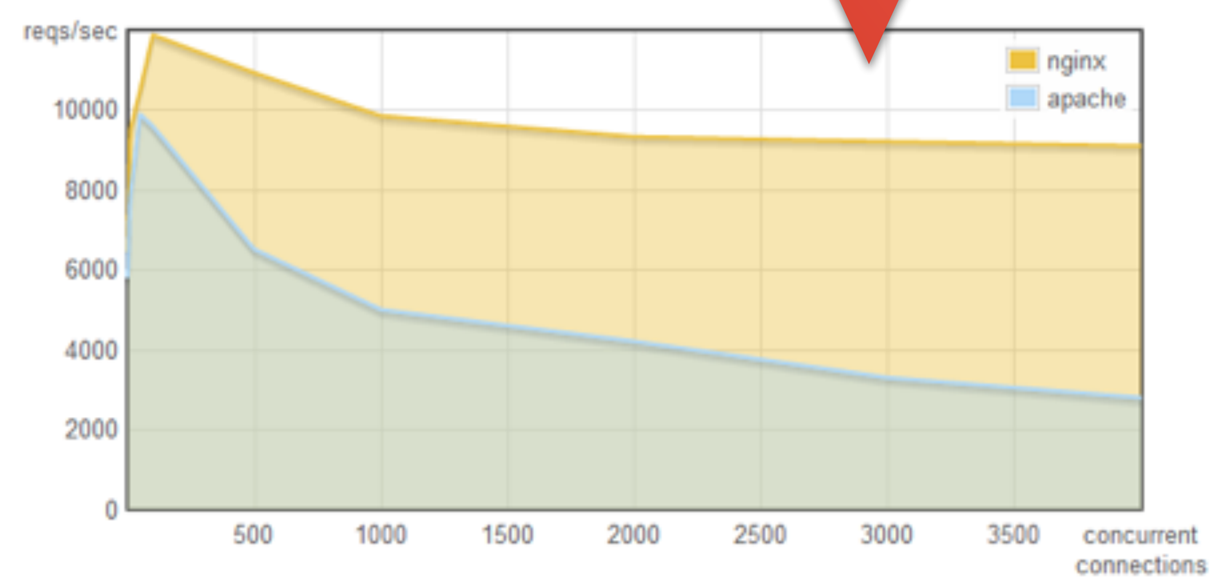
Parallelism should be use **wisely**

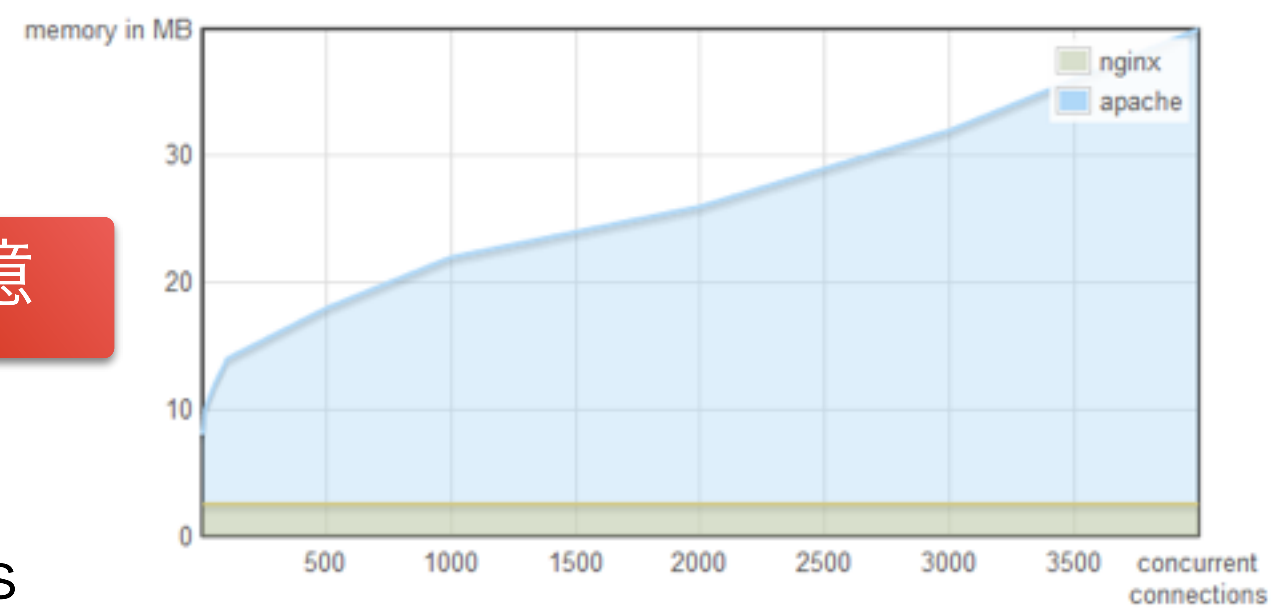
Thread vs. Asynchronous

- Threads are expensive for the system
  - Dedicated stack and structure in the kernels
  - Thread switch is time consuming (system calls, I/D cache misses)
- This cost sometimes overrides the benefits for threading
- In particular, **I/O intensive programs** may gain a lot from asynchronous, non-parallel programming
- Example of web server
  - I/O intensive (1 network access: about 100ms, that is, 1G clock cycles)
  - Apache: use one dynamic threads per user request
  - Nginx: use single-thread, asynchronous programming

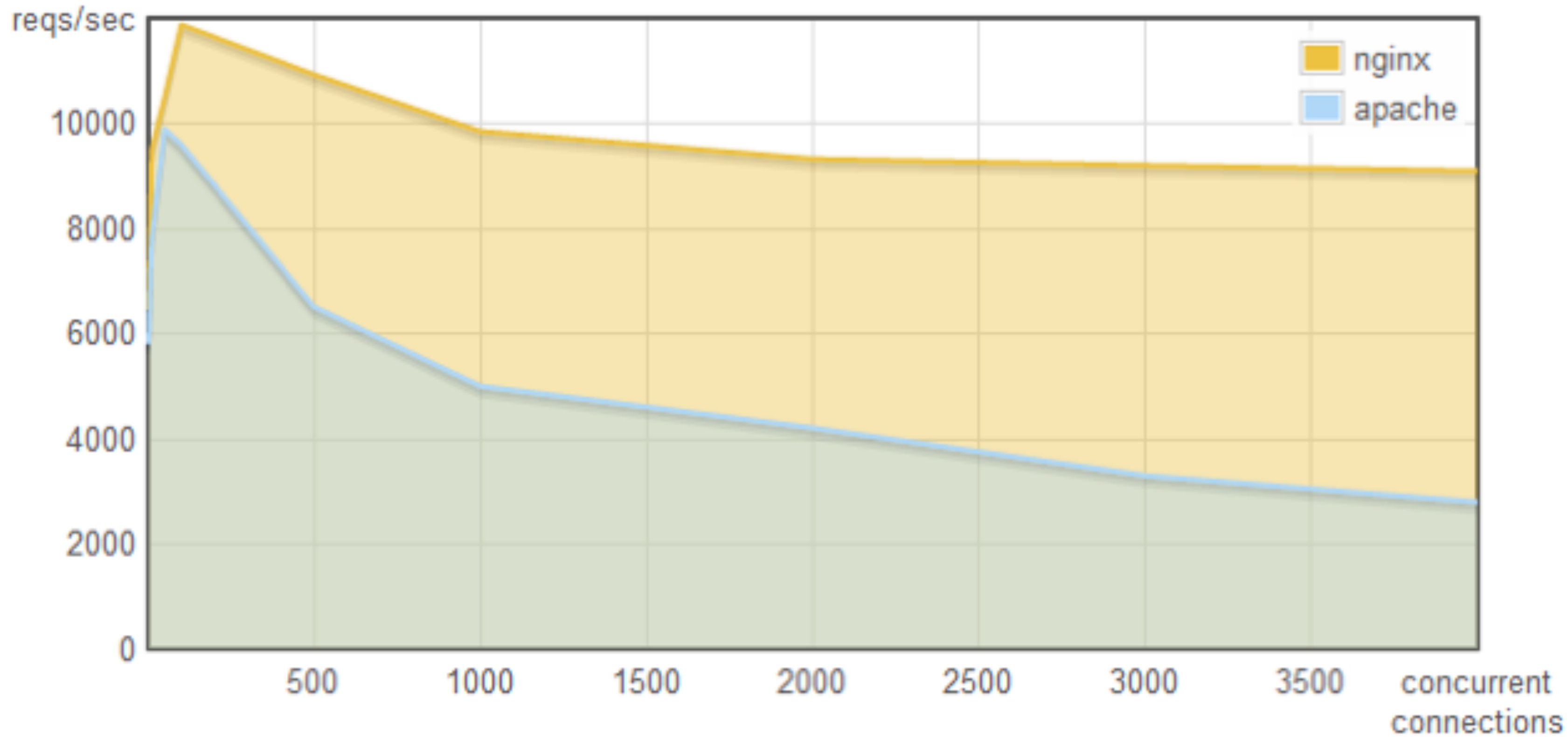
1千億

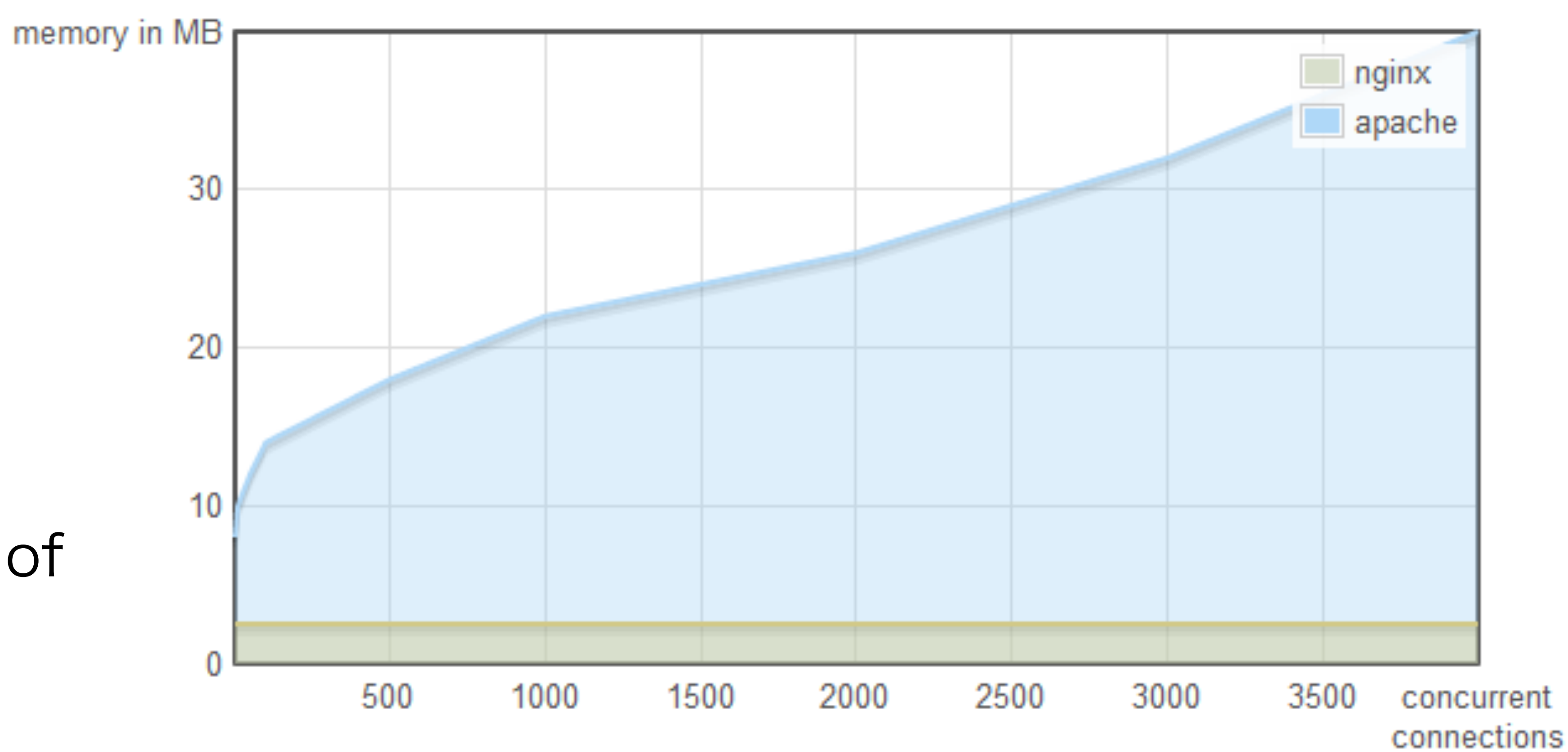
Nginx is 3 times faster, uses far less memory !





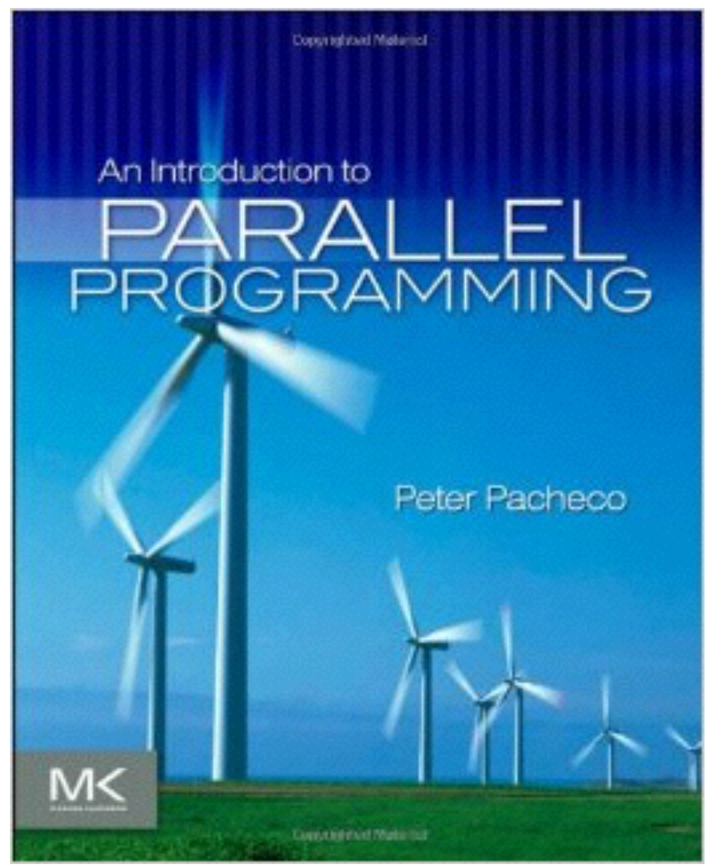
http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/





Note: the benchmarks consists of serving a lot of small static files

A Good Book


Copyrighted Material
An Introduction to
PARALLEL
PROGRAMMING
Peter Pacheco
MK
Copyrighted Material

お疲れさまでした