Recap
Chapter 1: Parallel computing introduction

Zhiming Zhao

University of Amsterdam

	Shared memory			Distributed memory		
	Single core	Multi core	Many cores…	Cluster	Cloud	Blockchain…
Sequential application						
Concurrent application						
Parallel application						
Distributed application						

What is the difference between concurrency and parallelism?

- Quiz: L1Q1
- DPP2025

<table>
<tr><td></td><td colspan="3">Shared memory</td><td colspan="3">Distributed memory</td></tr>
<tr><td></td><td>Single core</td><td>Multi core</td><td>Many cores…</td><td>Cluster</td><td>Cloud</td><td>Blockchain…</td></tr>
<tr><td>Sequential application</td><td colspan="6">Can be executed on all machines, but may not fully utilizing the resources.</td></tr>
<tr><td>Concurrent application</td><td colspan="6">Can be executed</td></tr>
<tr><td>Parallel application</td><td></td><td colspan="4">Can be executed</td><td></td></tr>
<tr><td>Distributed application</td><td></td><td></td><td></td><td colspan="3">Can be executed</td></tr>
</table>

Chapter 2
Introduction to multithreading and PThread

Zhiming Zhao

Multiscale Networked Systems

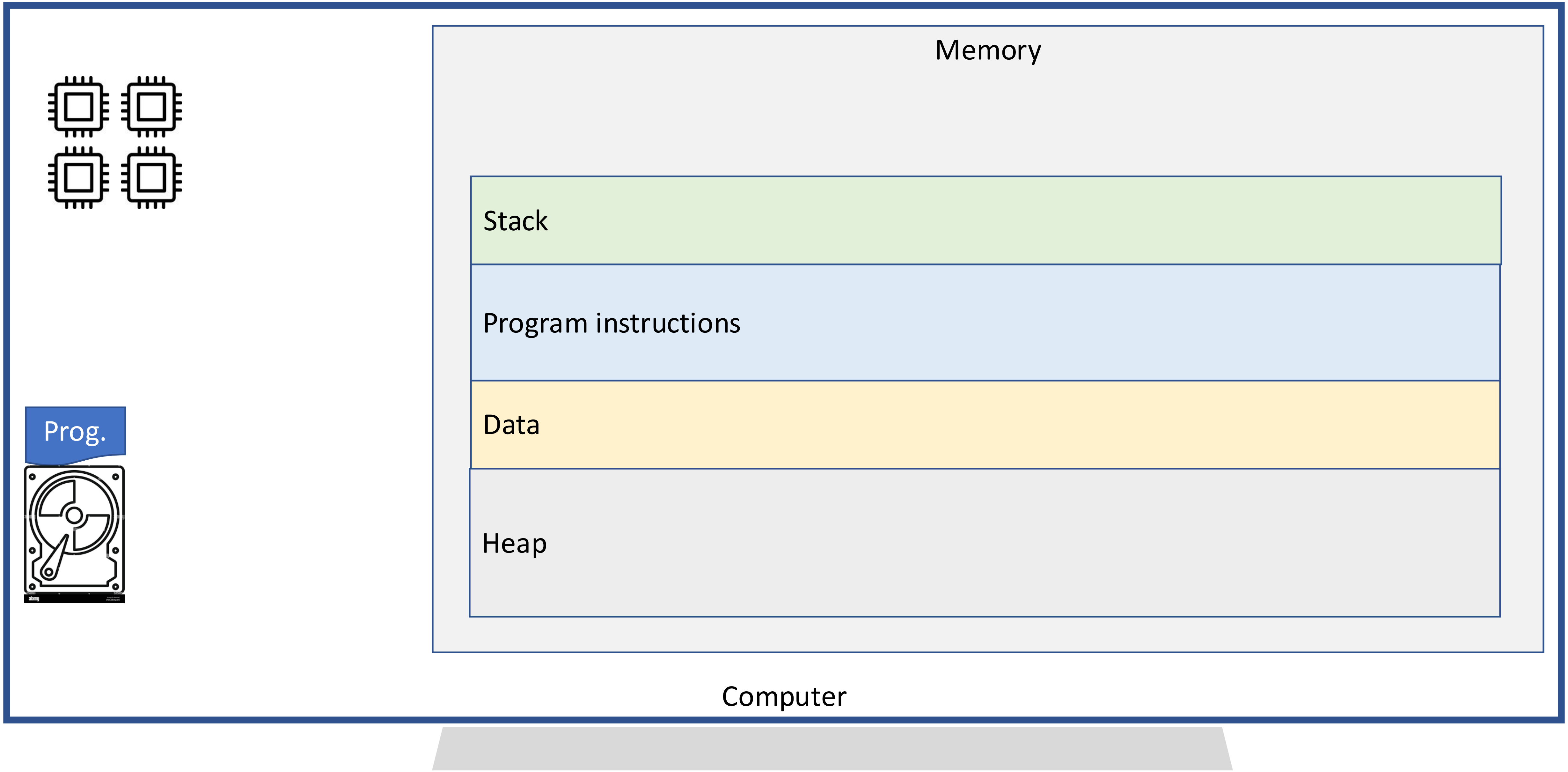
University of Amsterdam

Content

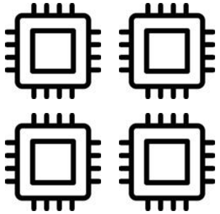
1. Multithreading
2. pThread
3. Race condition
4. Data parallelisation using pThread
5. Speedup and efficiency
6. Task parallelization

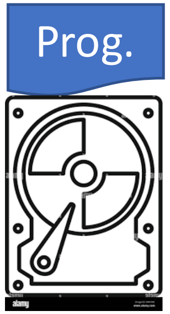
1. Multithreading

Process and thread

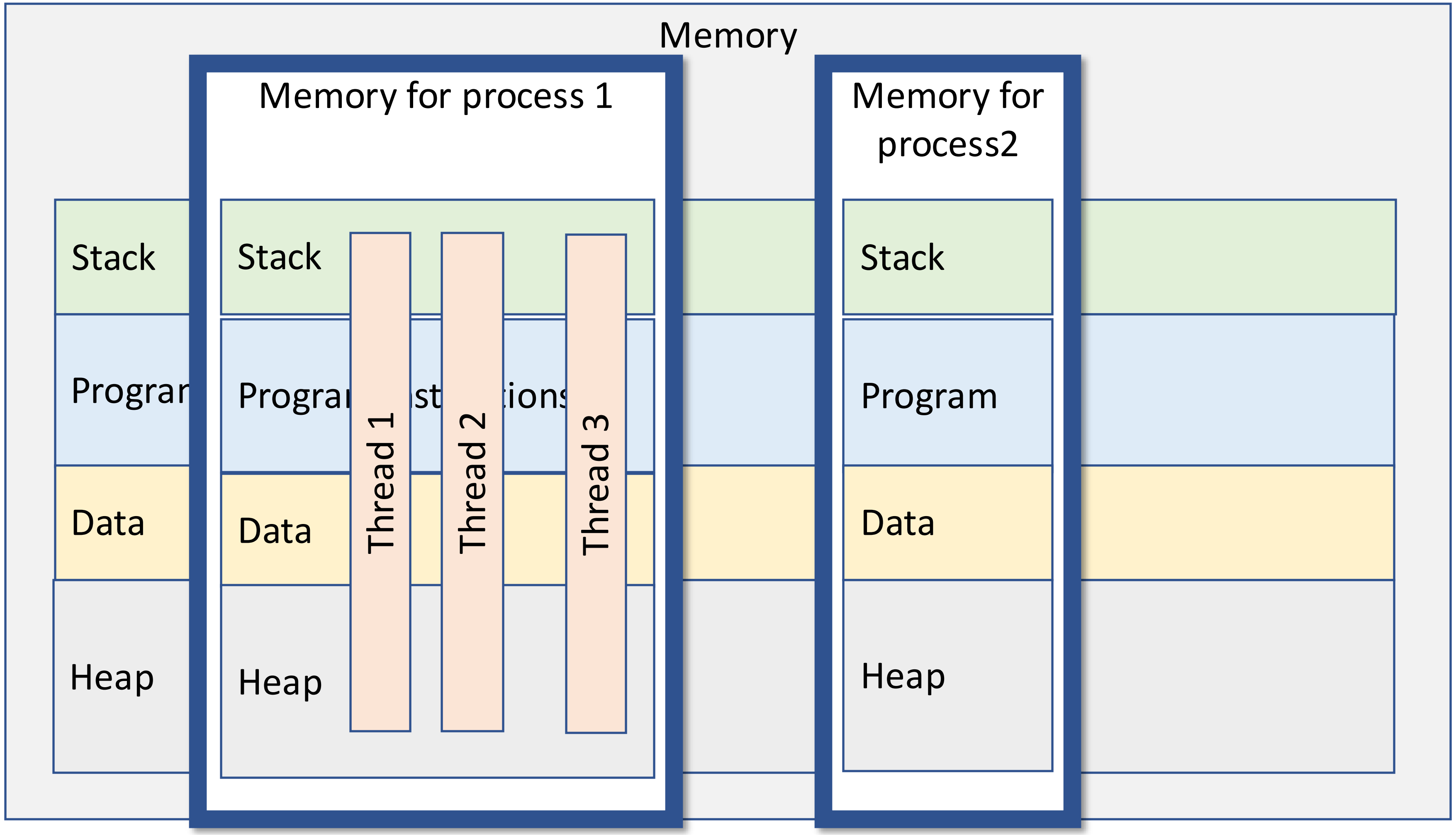


Process and thread





Being executed by the **Operating System**, a **program** will be loaded in the memory as a **process**.



Computer

Threads are “mini processes” inside a process, and share the memory of the process.

Process and Threads

- Processes
  - control their own resources
  - No shared memory
  - Communication through messages or files
  - Heavy-weight (expensive to set up and destroy)

- Threads
  - Control their limited local memory and shared address space
  - Communication through shared memory
  - Light-weight (much cheaper to set up and destroy)

Programming models

- Pthreads + intrinsics L03
- TBB – Thread building blocks
  - Threading library
- OpenCL, CUDA
  - To be discussed …
- OpenMP
  - High-level, pragma-based L04
- … <many more>
- Cilk
  - Simple divide-and-conquer model
- … <many more>

Level of abstraction increases

What will we learn?

- How to create a thread?
- How to perform computations within a thread?
- How do threads communicate with each other?
- How do threads concurrently access shared memory?
- How can you stop or terminate a thread?
- How do you design an appropriate multithreaded program for a specific problem?
- How do we evaluate the performance of threads?

2. Multithreading with POSIX threads

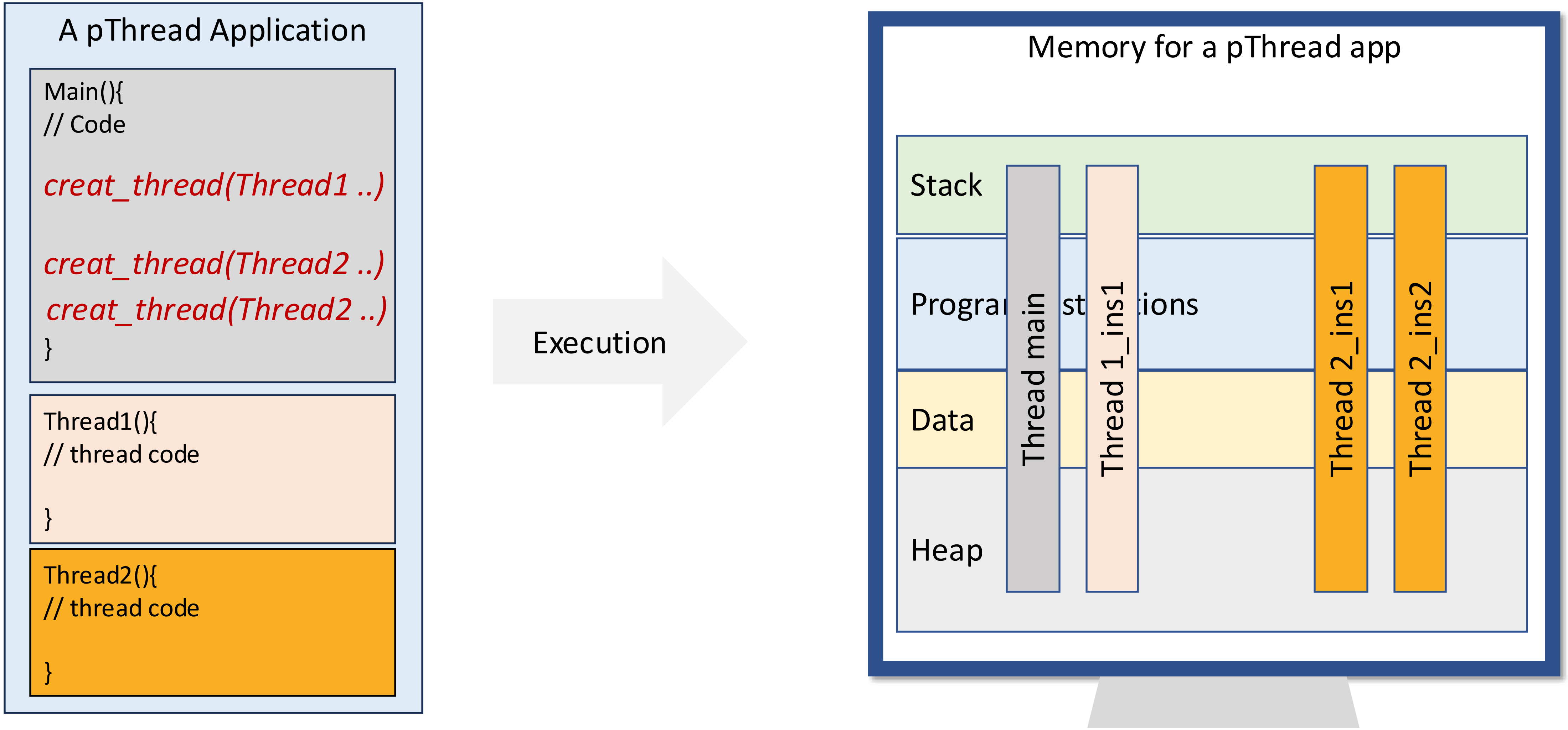
What are “Pthreads”

- Standardized programming interface for threads
  - UNIX systems : IEEE POSIX (Portable Operating Systems Interface) 1003.1c standard (1995)

- All implementations of these standards are known as POSIX threads or threads.

- Pthreads are defined as a set of **C language programming types** and **procedure called**
  - pthread.h header/include file
  - a thread library
    - Or part of libc in some implementations.

A high level view of multi thread program



Create and start a thread

```
int pthread_create (
        pthread_t* thread, // OUT: thread object
        pthread_attr_t* attributes, // IN: attributes
        void* (*start_routine)(void *), // IN: routine to start thread
        void* argument// IN: input attributes to the start routine
        );
```

2.1 Create and start a thread

```
int pthread_create (
    pthread_t* thread, // OUT: thread object
    pthread_attr_t* attributes, // IN: attributes
    void* (*start_routine)(void *), // IN: routine to start thread
    void* argument); // IN: input attributes to the start routine

int pthread_join (
    pthread_t thread_id ,  // IN: thread object
    void **return_value ) // OUT: return value
```

A “*hello world*” example

```c
/*
 * hello.c - Pthreads "hello, world" program
 */

#include <pthread.h>

void *thread(void *vargp);

int main()

{
        pthread_t thd;
        pthread_create(&thd, NULL, thread, NULL);
        pthread_join(thd, NULL);
        exit(0);
}


void *thread(void *vargp) /* thread routine */
{
      printf("Hello, world!\n");
      return NULL;
}
```

1. #include <pthread.h>
2. pthread_create(&thd, NULL, thread, NULL);
3. pthread_join(thd, NULL);

To create a thread

```
/*
* hello.c - Pthreads "hello, world" program
*/

#include <pthread.h>

void *thread(void *vargp);

int main()

{
        pthread_t thd;
        pthread_create(&thd, NULL, thread, NULL);
        pthread_join(thd, NULL);
        exit(0);

}


void *thread(void *vargp) /* thread routine */
{
      printf("Hello, world!\n");
      return NULL;
}
```

1. Thread routine name
2. Thread object

Question: 1) how many threads will be created? 2) what will be the output?

To create multiple threads

```c
/*
* hello.c - Pthreads "hello, world" program
*/

#include <pthread.h>

void *thread(void *vargp);

int main()
{   int number=5;
    pthread_t *thd;
    thd = (pthread_t*) malloc (sizeof(pthread_t)*number);

    for (int i=0;i<number; i++)
       pthread_create( thd[i], NULL, thread, NULL);
    for (int i=0;i<number;i++)
       pthread_join(thd[i], NULL);
    exit(0);
}


void *thread(void *vargp) /* thread routine */
{
      printf("Hello,world!\n");
      return NULL;
}
```

&

1. Each thread needs a dedicated **pthread_t object**
2. Create thread **one by one**, they can have **different thread routine**
3. Put the **join** after creating all threads.

Discussion:

1. Do you have to create threads in the **main** function?
2. Any bug you can find in the code?

To create multiple threads

```c
/*
* hello.c - Pthreads "hello, world" program
*/

#include <pthread.h>

void *thread(void *vargp);

int main()

{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);

        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        exit(0);

}


void *thread(void *vargp) /* thread routine */
{
      printf("Hello,world!\n");
      return NULL;
}
```

1. Each thread needs a dedicated **pthread_t object**
2. Create thread **one by one,** they can have **different thread routine**
3. Put the **join** after creating all threads.
4. **Pthread_t *** in the create function!!!

Question: How does a thread know its position among all threads?

Another example

```
#include <pthread.h>

void *thread(void *vargp);

int count=0;

int main()

{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);


        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n",                count);
        exit(0);

}

void *thread(void *vargp) /* thread routine */
{
      for (int i=0;i< 100;i++)
           count++;
      return NULL;
```

Discussion

1. What will the program print?
   1. 500
   2. >=500
   3. > 500
   4. <=500
   5. < 500
   6. 0

2. How to solve it?

count ++ will be executed as i) read count, ii) add count with 1, iii) write count

Variables and memory access

```
#include <pthread.h>

void *thread(void *vargp);

int count=0;

int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);

        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n",               count);
        exit(0);
}


void *thread(void *vargp) /* thread routine */
{
  static int total = 100;
      for (int i=0;i< total;i++)
          count++;
      return NULL;
}
```

- Global variables
  - Def: Variable declared outside of any function
  -
- Local variables
  - Def: Variables declared inside functions, without the ***static*** attribute
  -
- Local static variables
  - Def: Variable declared inside functions, with the ***static*** attribute
  -

Question: What are the global, local, and local static variables?

Variables and memory access

```
#include <pthread.h>

void *thread(void *vargp);

int count=0;

int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);

        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n",              count);
        exit(0);
}


void *thread(void *vargp) /* thread routine */
{
   static int total = 100;
      for (int i=0;i< total;i++)
           count++;
      return NULL;
}
```

- Global variables
  - Def: Variable declared outside of any function
  - Exactly **one instance** exists for each global variable (shared across all functions and threads)
- Local variables
  - Def: Variables declared inside functions, without the ***static*** attribute
  - **Each thread stack contains its own instance of each local variable**
- Local static variables
  - Def: Variable declared inside functions, with the ***static*** attribute
  - Exactly **one instance** exists for each local static variable, shared across all function calls and threads.

Question: What are the global, local, and local static variables?

2.2 Race Condition

- A race condition occurs when two or more threads in one application:
  - try to access the same memory location(**variables**) **concurrently**, and
  - more than one of the accesses are for **writing** and
  - the threads **do not use** exclusive mechanisms to **control their access** to that memory.

3. Race condition

Race Condition

A race condition occurs when two or more threads in a single process:

- try to access and change the same variable at the same time
- the threads do not use exclusive mechanisms to control their access

Most solutions are based on

1. Replication
2. Atomic operation
3. Locking

Choice 1: move global variable to local

```
#include <pthread.h>
void *thread(void *vargp);
int countT[5];
int count=0;
int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, &countT[i]);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        for (int i=0;i<number;i++)
           count+=countT[i];
        printf("Result:%d\n",               count);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
  int *localCount=(int*)vargp;
  *localCount=0;
      for (int i=0;i< 100;i++)
         (*localCount) ++;
      return NULL;
}
```

Replicate variables

1. Pass variables to each thread: countT[i]
2. Each thread adds its own copy of the variable: localCount
3. The main thread calculates the final result: count

Question: What output will you get?

- 500

Note:

- How do you pass value or address to the thread?
- Don't forget to initialise the variable.

Choice 2: Using atomic operation

```
#include <pthread.h>
#include <stdatomic.h>
void *thread(void *vargp);
atomic_int acount=ATOMIC_VAR_INIT(0);
int count=0;

int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);

        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", acount);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
      for (int i=0;i< 100;i++)
      atomic_fetch_add_explicit( &acount, 1, memory_order_relaxed);
      return NULL;
}
```

Solution:

1. Define shared variable as atomic variable: **atomic_int acount**
2. Use atomic operation:

atomic_fetch_add_explicit

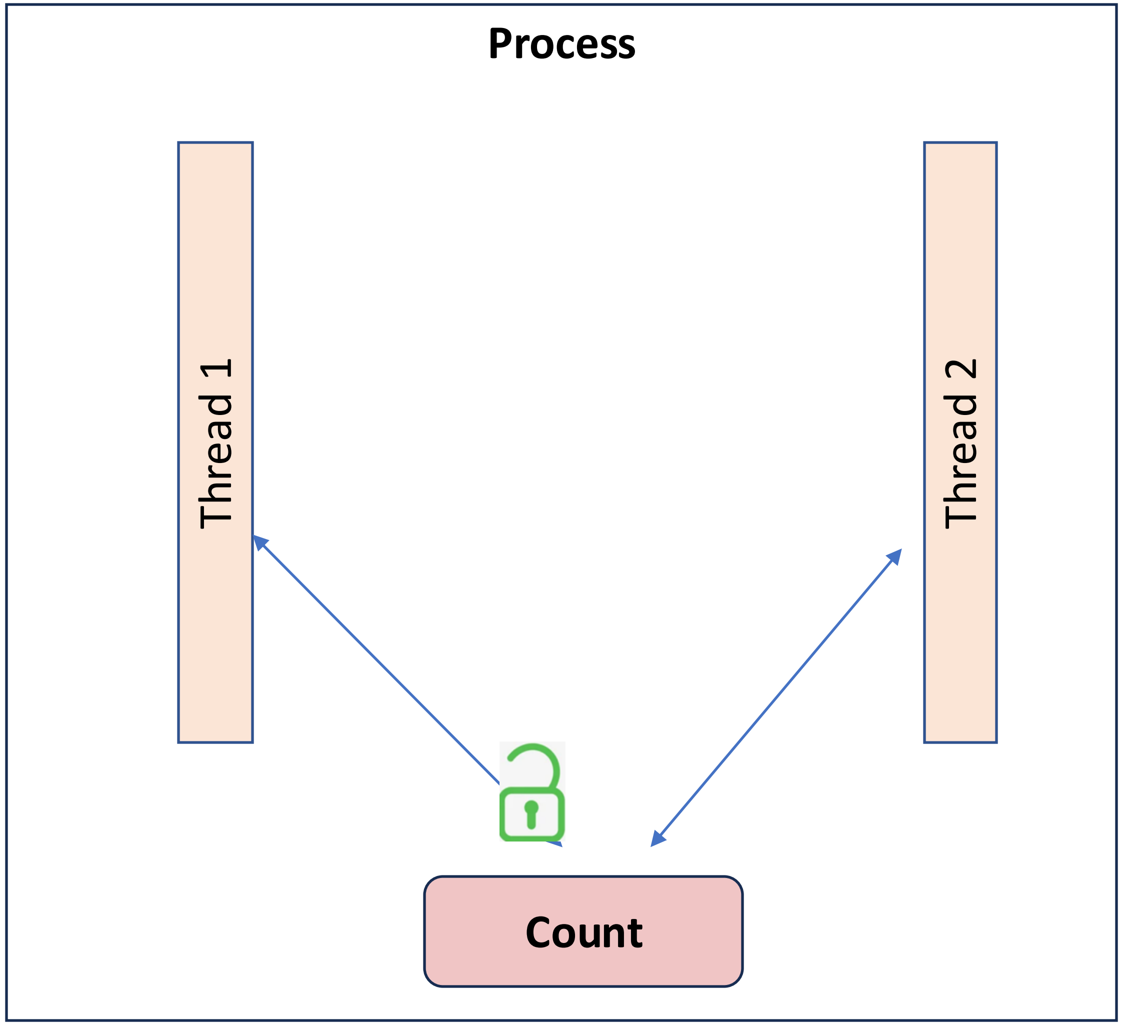
It will also guarantee an output as 500.

Note:

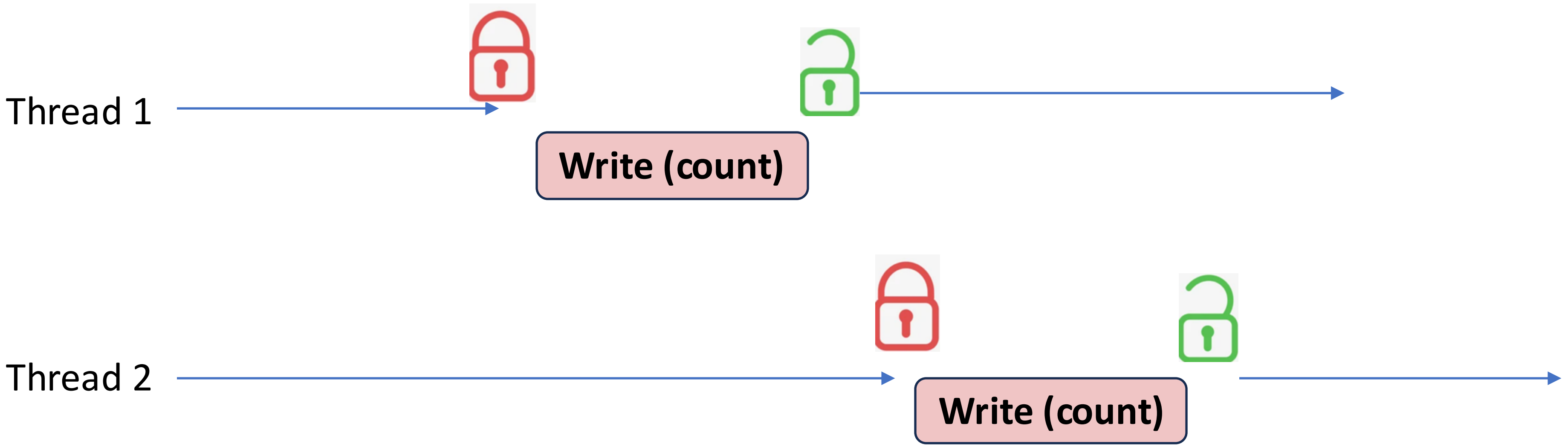
1. Not always possible for complex operations

Choice 3: exclusive access to critical region

- Threads are NOT allowed to access the same region simultaneously
- Only one thread is allowed to use the region at one time



How does it work?



1. A thread has to **lock** a critical region if it wants to work on it
2. A thread can only **lock** a critical region when its status is **unlocked** (available)
3. A thread has to **wait** for a locked region to become **unlocked** if it wants to work on it
4. A thread can only enter the critical region when it has **locked** it.
5. A thread has to **unlock** a critical region after it finishes the work on it

Mutex locks: essential functions

Global/main/static:

```
pthread_mutex_t lock;
```

Main:

```
pthread_mutex_init(&lock, attributes);
…
pthread_mutex_destroy(&lock);
```

Per-thread:

```
pthread_mutex_lock ( &lock );
 // critical region here
 pthread_mutex_unlock ( &lock );
```

Code example

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
pthread_mutex_t mutex;
int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        pthread_mutex_init(& mutex, NULL);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        pthread_mutex_destroy(& mutex);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
      for (int i=0;i< 100;i++)
      {  pthread_mutex_lock( &mutex );
         count ++;
         pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

Use a lock

1. Critical region: **count**
2. pthread_mutex_t
3. pthread_mutex_init
4. pthread_mutex_destroy
5. **pthread_mutex_lock**
6. **pthread_mutex_unlock**
7. ***pthread_mutex_trylock***

Question: What output will you get?

- 500

Note:

- You can define multi mutex objects for lock
- Carefully check which one is using

More mutex functions

- Static initialization with default attributes:
- **pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;**

- Dynamic initialization with attributes:
- **int pthread_mutex_init (pthread_mutex_t *lock,**
  pthread_mutexattr_t * attributes);
  - Attributes* include ERRORCHECK, RECURSIVE, DEFAULT

- Dynamic de-allocation:
- **int pthread_mutex_destroy (pthread_mutex_t * lock);**

- Optimistic locking:
- **int pthread_mutex_trylock (pthread_mutex_t * lock);**
    - `// returns 0 upon success`
    - `// returns EBUSY upon failure`

- (counting) Semaphores: multiple resources => multiple threads
- **sem_init(), sem_wait(), sem_post(), …**

*Check, for example: **pthread_mutexattr_gettype**

Choice 3: Enforce the critical region

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        pthread_mutex_init(& mutex, NULL);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        pthread_mutex_destroy(& mutex);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
      for (int i=0;i< 100;i++)
      {  pthread_mutex_lock( &mutex );
         count ++;
         pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

Where should we put the lock?

```
void *thread(void *vargp) /* thread routine */
{
  pthread_mutex_lock( &mutex );
      for (int i=0;i< 100;i++)
         count ++;
      pthread_mutex_unlock( &mutex);
      return NULL;
}
```

Mutex locks

- Mutex locks are abstract data objects.
- Only one thread at a time may hold a mutex lock.
- Threads block upon locking if lock is unavailable.
- Locking / unlocking are guaranteed to be atomic.
- No fairness on waiting threads upon unlocking.
- Only the owner of the lock can unlock it.
- Re-locking by the owner causes deadlock.

Question: what is the output?

```c
#include <pthread.h>
void *thread(void *vargp);
int count=0;
int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

      for (int i=0;i< 100;i++)
      {   pthread_mutex_lock( &mutex );
          count ++;
          pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500

Question: what is the output?

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
int main()
{  int number=5;
      pthread_t *thd;
      thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
      for (int i=0;i<number; i++)
         pthread_create(&thd[i], NULL, thread, NULL);
      for (int i=0;i<number;i++)
         pthread_join(thd[i], NULL);
      printf("Result:%d\n", count);
      exit(0);
}
void *thread(void *vargp) /* thread routine */
{
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
     for (int i=0;i< 100;i++)
     {  pthread_mutex_lock( &mutex );
        count ++;
        pthread_mutex_unlock( &mutex);
     }
     return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500

Be very careful when using static variables in the threads.

Question: what is the output?

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
int main()
{   int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
static pthread_mutex_t mutex;
  pthread_mutex_init(& mutex, NULL);
      for (int i=0;i< 100;i++)
      {   pthread_mutex_lock( &mutex );
          count ++;
          pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500

Be very careful when you call a function inside a thread. Make sure it is thread **safe**.

Locks & Deadlocks

- **Locking** = a thread locks a resource for single use.

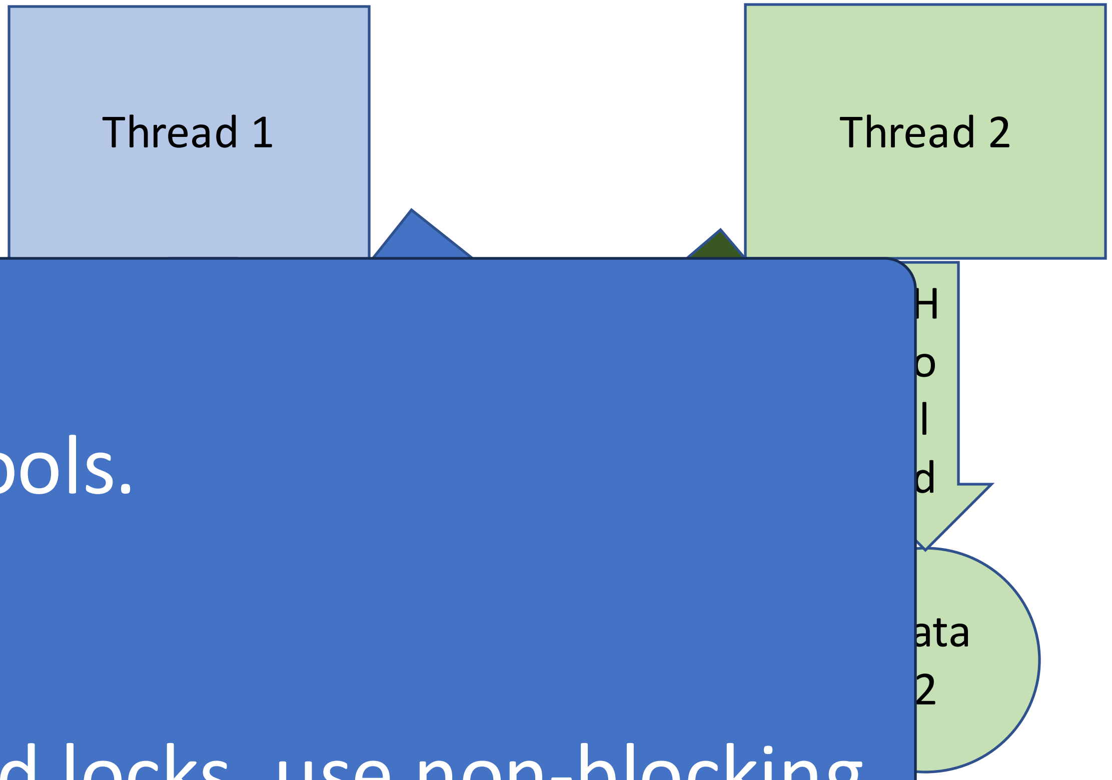
- **Deadlock** = threads are blocked, waiting for each other and no progress is being made.
  - Examples?
    - 4-way stop
    - Dining philosophers problem

- **Livelock** = threads "oscillate" between states with regard to one another and no progress is being made.
  - Examples?
    - People meeting "on the corridor"
    - MAC protocol collisions

Deadlock

Deadlock occurs when

1.Hold and Wait



Detecting Deadlock:

•Cycle detection,   or debugging tools.

Fixing Deadlock:

•Timeouts with retries, avoid nested locks, use non-blocking lock attempts, or use higher-level concurrency utilities.

Question: How to fix it?

Summary: race condition solutions

- Solution 1
  - 1 counter per thread
  - Advantage?
    - No data races
  - Disadvantage?
    - More memory
    - Reduction phase

- Solution 2
  - Busy waiting for an atomic operation
  - Advantage?
    - Do not suspend the thread execution.
  - Disadvantage?
    - Not suitable for complex coding blocks

- Solution 3
  - Enforce a critical region
    - Guarantees all operations will execute without interleaving
  - Advantage?
    - Same memory
    - No reduction
  - Disadvantage?
    - Serialization!

Which one is better?

4. Data parallelization using pThread

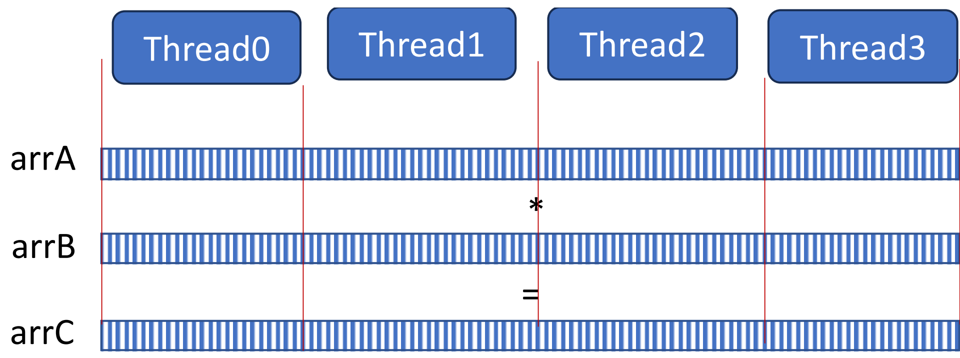
Discussion: How to do data parallelism?

```c
#include <pthread.h>
#include <stdatomic.h>
void *thread(void *vargp);
int *arrA, *arrB, *arrC;
int main()
{   int number=4;
    int length=100000;
        pthread_t *thd;
        arrA=(int*)malloc(length*sizeof(int));
        arrB=(int*)malloc(length* sizeof(int));
        arrC=(int*)malloc(length* sizeof(int));

        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
          pthread_create(&thd[i], NULL, thread, NULL);

        for (int i=0;i<number;i++)
          pthread_join(thd[i], NULL);
        printf("Result:%d\n", acount);
        exit(0);
}
```

```c
void *thread(void *vargp)
{
    for (int i=?;i< ?;i++)
      // compute arrC[x]=arrA[x]*arrB[x]
    Return NULL;
}
```



Question: How to inform the threads about their tasks?

How to inform the thread about their task?

```
#include <pthread.h>
#include <stdatomic.h>
void *thread(void *vargp);
int *arrA, *arrB, *arrC;
typedef struct {//data to send to each thread
   int start, end;
   int id;
} t_data;

int main()
{  int number=5;
         pthread_t *thd;
         t_data *myData;
         // skip the code for initializing thd, myData etc.
         // compute the start/end for each thread
         for (int i=0;i<number; i++)
            pthread_create(&thd[i], NULL, thread, (void*)&myData[i]);
         ...
}

void *thread(void *vargp) /* thread routine */
{
   int start= *((t_data*) vargp).start;
   int end= *((t_data*) vargp).end;
   ...
       for (int i=start ;i< end;i++)
            arrC[x]=arrA[x]*arrB[x]
}
```

```
//create the threads in the main ()
    for (i=0; i<number; i++) {
        myData[i].id=i;
        myData[i].start=i*(length/number);
        myData[i].end=(i+1)*(length/number)-1;
        pthread_create(&thd[i], NULL, thread,(void*)&myData[i]);
    }
```

- Send it to each thread

How to synchronize the progress of different threads?

```
int cnt[10000];
int average[NUM_ITER];
void* thread_func(void* args) {
 // obtain start, end from args
 for(int i = 0; i<NUM_ITER; i++)
   {
     for(int j=start;j<end;j++)
          cnt[j]++;
      //compute_the_average_of_cnt
   }
}

int main() {
   int *tid=malloc(sizeof(int));
   // .. Some code for configuring the start/end for each thread

   for (int i = 0; i < num_threads; i++)
      pthread_create(&threads[i], NULL, thread_func, (void*)&myData);
   for (i = 0; i < num_threads; i++) {
      pthread_join(threads[i], NULL);
   }
   return 0;
}
```

- Threads compute different data chunks often need to be **synchronized**, e.g., from one iteration to the next one
- How do they synchronise the progress?

Other synchronization primitives

- Barriers: all threads wait at barrier before they all can continue.
  pthread_barrier_*(…)**
  e.g. **pthread_barrier_init(&barr, attr, count)** **pthread_barrier_wait(&barr)**

- Join: wait for “child” threads to finish and collects results
  pthread_join(…)

How to synchronize the progress of different threads?

```
int cnt[10000];
int NUM_ITER=100;
int num_threads;
pthread_barrier_t barrier;
void* thread_func(void* args) {
 // obtain start, end from args
 for(int i = 0; i<NUM_ITER; i++)
   {
     for(int j=start;j<end;j++)
          cnt[j]++;
     pthread_barrier_wait(&barrier);
     //compute_the_average_of_cnt
   }
}

int main() {
   int *tid=malloc(sizeof(int));
   // .. Some code for configuring the start/end for each thread

   pthread_barrier_init(&barrier, NULL, num_threads);
   for (int i = 0; i < num_threads; i++)
      pthread_create(&threads[i], NULL, thread_func, (void*)&myData);
   for (i = 0; i < num_threads; i++) {
      pthread_join(threads[i], NULL);
   }
   return 0;
}
```

- pthread_barrier
- pathread_barrier_wait
- pthread_join

Question: what is the output?

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
int main()
{  int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
static pthread_mutex_t mutex;
  pthread_mutex_init(& mutex, NULL);
      for (int i=0;i< 100;i++)
      {  pthread_mutex_lock( &mutex );
          count ++;
          pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500

Be very careful when you call a function inside a thread. Make sure it is thread **safe**.

Question: what is the output?

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
pthread_barrier_t barrier;
int main()
{  int number=5;
        pthread_t *thd;
        thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
         pthread_barrier_init(&barrier, NULL, num_threads);
        for (int i=0;i<number; i++)
           pthread_create(&thd[i], NULL, thread, NULL);
        for (int i=0;i<number;i++)
           pthread_join(thd[i], NULL);
        printf("Result:%d\n", count);
        exit(0);
}
void *thread(void *vargp) /* thread routine */
{
  Static pthread_mutex_t mutex;
  pthread_mutex_init(& mutex, NULL);
  pthread_barrier_wait(&barrier);
     for (int i=0;i< 100;i++)
     {   pthread_mutex_lock( &mutex );
         count ++;
         pthread_mutex_unlock( &mutex);
     }
     return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500

Question: what will be the output?

```
#include <pthread.h>
void *thread(void *vargp);
int count=0;
pthread_barrier_t barrier;
int main()
{   int number=5;
            pthread_t *thd;
            thd = (pthread_t*) malloc (sizeof(pthread_t)*number);
             pthread_barrier_init(&barrier, NULL, num_threads);
            for (int i=0;i<number; i++)
               pthread_create(&thd[i], NULL, thread, NULL);
            for (int i=0;i<number;i++)
               pthread_join(thd[i], NULL);
            printf("Result:%d\n", count);
            exit(0);
}
void *thread(void *vargp) /* thread routine */
{
   Static pthread_mutex_t mutex;
   pthread_mutex_init(& mutex, NULL);
   pthread_barrier_wait(&barrier);
      for (int i=0;i< 100;i++)
      {   pthread_mutex_lock( &mutex );
          count ++;
          pthread_barrier_wait(&barrier);
          pthread_mutex_unlock( &mutex);
      }
      return NULL;
}
```

What is the output?

1. 500
2. 0
3. <=500
4. Deadlock

Pthread parallelism

- Data parallelism, e.g., Processing large array
  - Allocate data ranges for each thread
  - Identify potential race conditions
  - Apply lock and synchronization mechanisms

5. Speedup and efficiency for data parallelisation

Speed-up given n processors(cores)

- Highly dependent on Time(1)
- *Best performed sequential execution as the baseline*

Sequential execution

$$Speedup(n) = \frac{Time_{best}(1)}{Time(n)}$$

Parallel execution

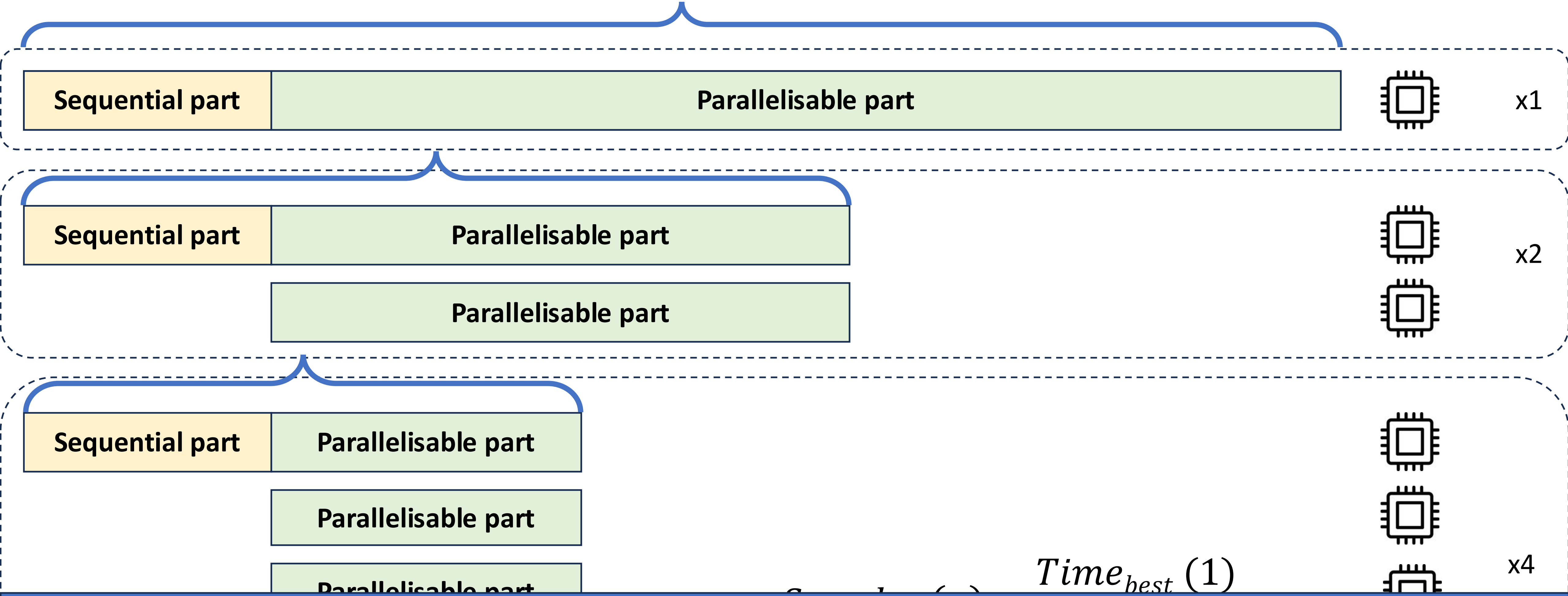
How do threads speed up a sequential application?

- Can we speed up a data parallelisable sequential application using multithreads on a single core?
  - No! We need more cores to make concurrent threads in parallel.
- Can we speed up a task parallelisable sequential application using multithreads on a single core?
  - Yes! Avoid busy waiting in the sequential execution

How do multi cores speed up a multithread application?

- For **a given problem size**, how will the program be **speed up** when **adding more processors(cores)**?
- For **a given multithread program**, how will it handle **bigger problem size** when **adding more processors(cores)**?

How do multi cores speed a multithread application up?



Discussion: How will race conditions affect the speed-up?

Amdahl's Law

- The speedup of an application when using multiple processors is limited by the app's sequential parts

- Thus:

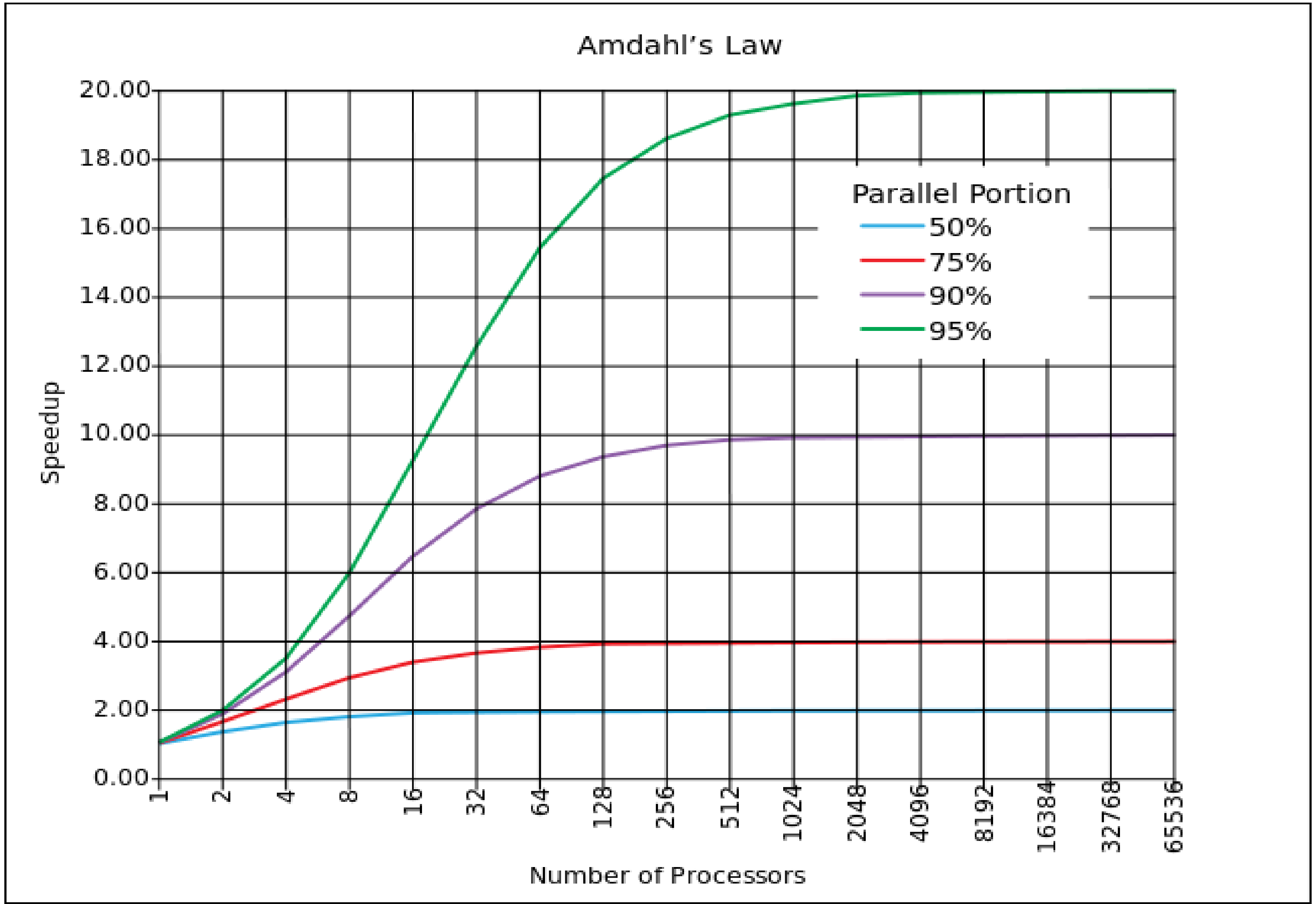
$$Time_{new} = Time_{sequential\_part} + \frac{Time_{parallelizable\_part}}{Number\ of\ processors}$$

- Or:

$$Speedup = \frac{1}{(1 - Fractionpa_{rallelizable\_part}) + \frac{Fraction_{parallelizable\_part}}{Number\ of\ processors}}$$

Amdahl's Law in pictures

$$Speedup = \frac{1}{(1 - Fraction_{parallelizable}) + \frac{Fraction_{parallelizable}}{Number\ of\ processors}}$$



$$Speedup_{up\_limit} = \frac{1}{(Fraction_{sequential})}$$

Efficiency

How well the application utilises the computational resources to achieve the speedup.

$$Efficiency(n) = \frac{Speedup(n)}{n} = \frac{Time_{best}(1)}{n * Time(n)}$$

Efficiency

Efficiency can also be seen as the ratio between the “ideal performance” and the “actual performance”.

$$Efficiency(n) = \frac{Speedup(n)}{n} = \frac{Time_{best}(1)}{n * Time(n)}$$

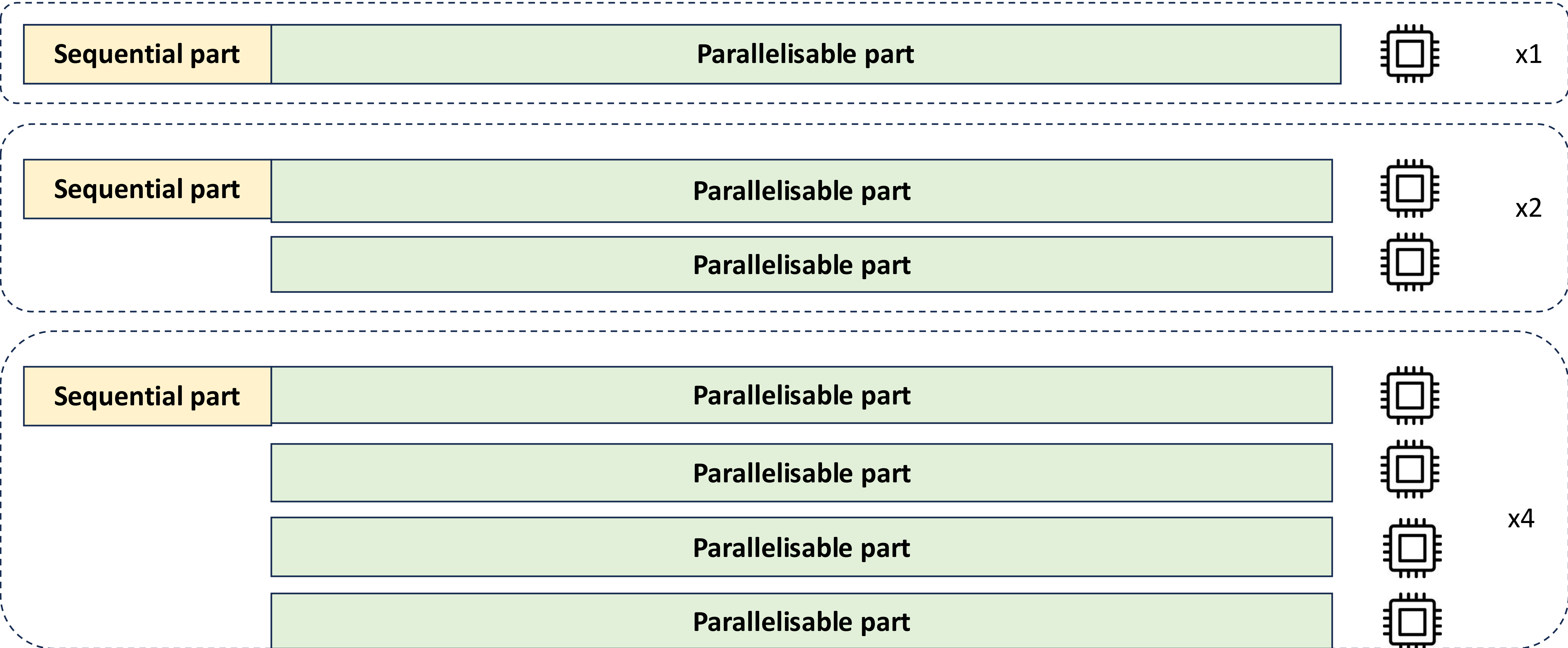
$$= \frac{Time_{ideal}(n)}{Time(n)}$$

$$Time_{ideal}(n) = \frac{Time_{best}(1)}{n}$$

How do multi cores speed up a multithread application?

- For **a given problem size**, how will the program be **speed up** when **adding more processors(cores)**?
- For **a given multithread program**, how will it handle **bigger problem size** when **adding more processors(cores)**?

How problem size changes with more cores?



Gustafson's Law in pictures

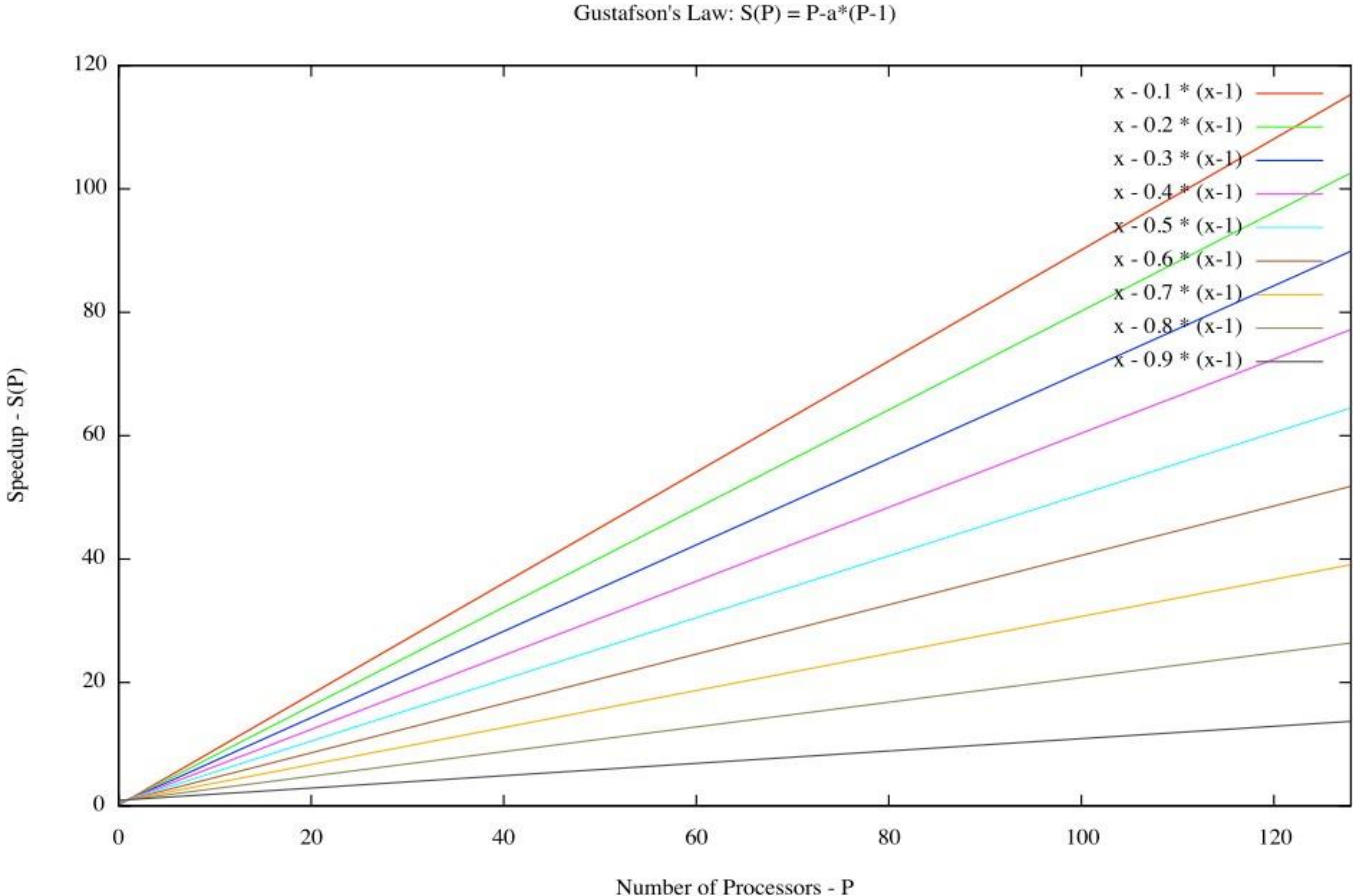
- Allows for scaling the problem size
- Gustafson's law considers the execution time of a parallel program using *n* processors with a sequential fraction *s* (with s in [0,1])

- *Translates into:*

$$Speedup = \frac{Fraction_{Sequantial\_part} + n * (1 - Fraction_{Sequantial\_part})}{Fraction_{Sequantial\_part} + (1 - FractionSeq_{uantial\_part})}$$

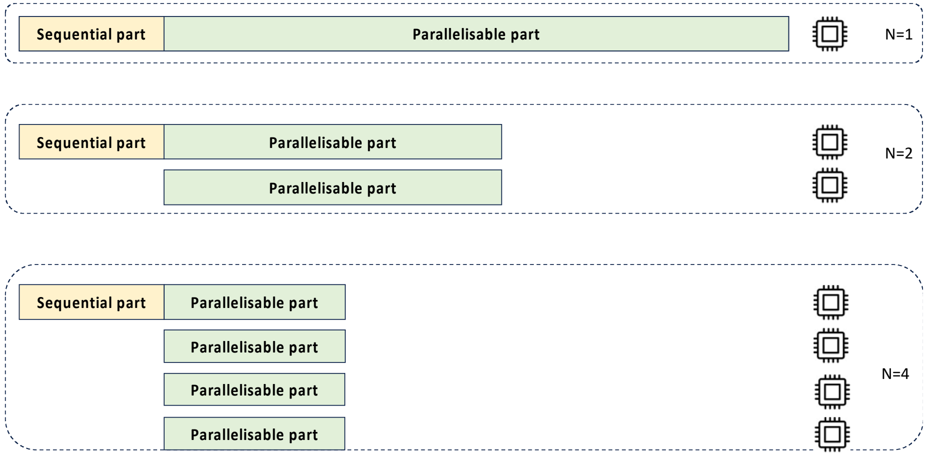
$$\begin{aligned} &Speedup \\ &= FractionSequantial_{\_part} + n * (1 - Fraction_{Sequantial\_part}) \\ &= n - Fraction_{Sequantial\_part} * (n - 1) \end{aligned}$$

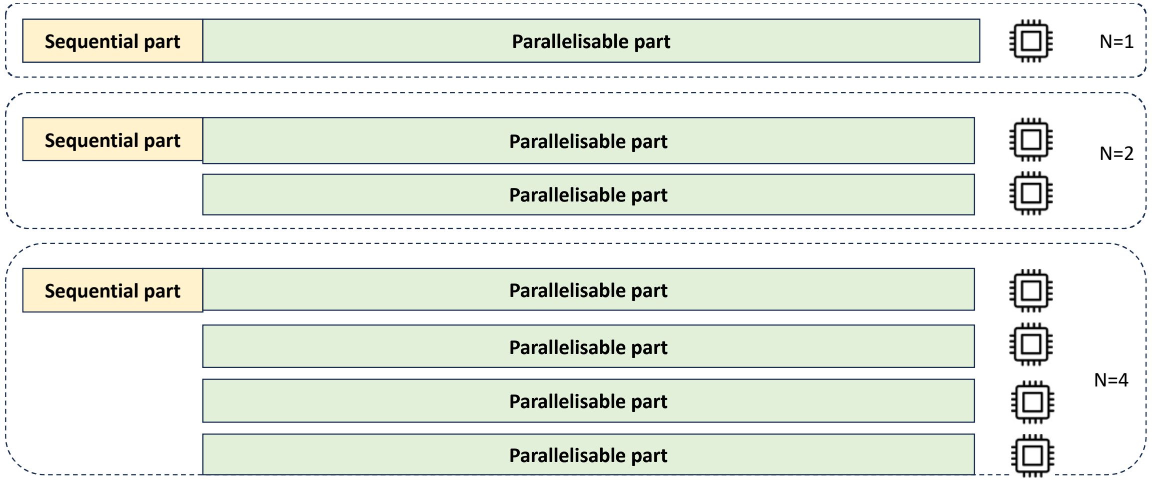
Gustafson’s Law in pictures



What is the difference between Amdahl's and Gustafson's law?

- Amdahl's law
- Gustafson's law





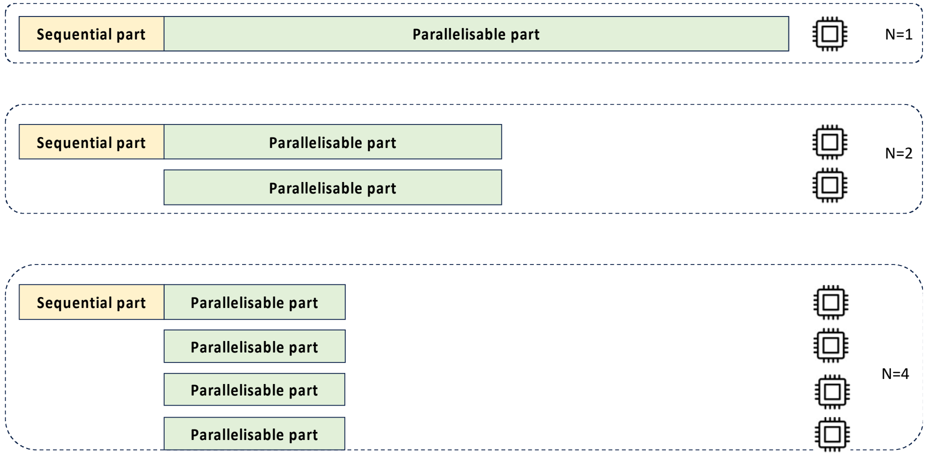
What is the difference between Amdahl’s and Gustafson’s law?

- Amdahl’s law

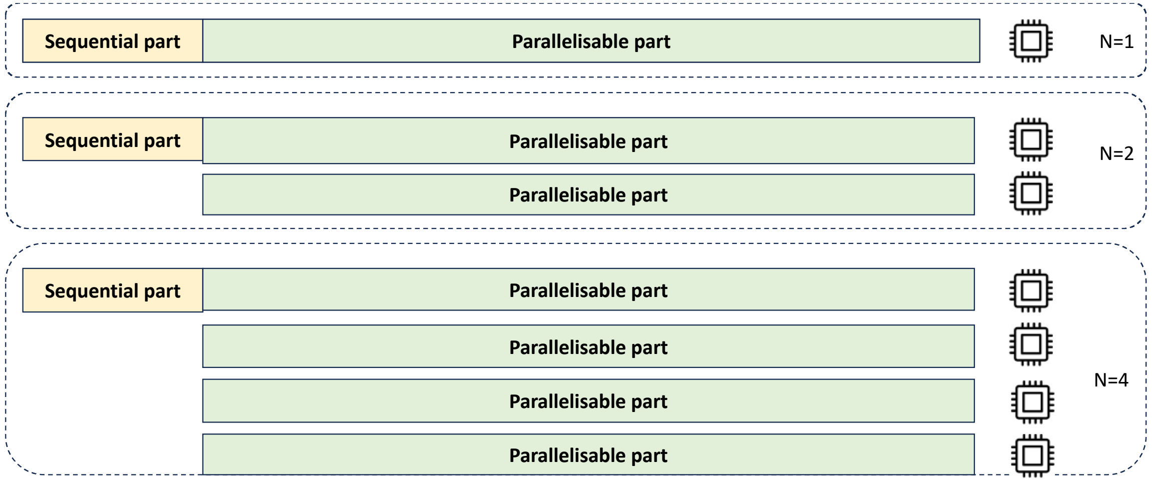
  How does the ***speedup*** of a **fixed-size problem *scale*** when it is executed on **N cores**?

- Gustafson’s law

  How does the ***speedup*** of a **changing size problem *scale*** when it is executed on **N cores**, but with **fixed size on each core**?



Strong scaling



Weak scaling