Multicore Programming (ACSAI)

Gianmaria Romano

A.Y. 2025/2026

Disclaimer

These notes were taken during the "Multicore Programming" course given by Professor De Sensi during the first semester of the Academic Year 2025/2026 at the course "Informatica" of Sapienza Università di Roma.
Keep in mind, however, that these notes do **not** replace the course material, so it is suggested to check the professor's resources as well.

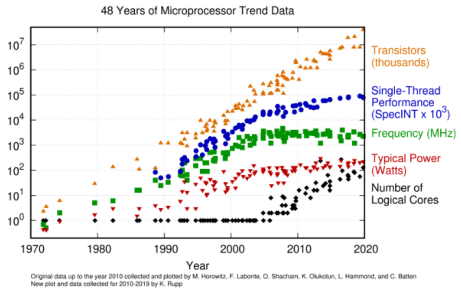
Contents

Chapter 1

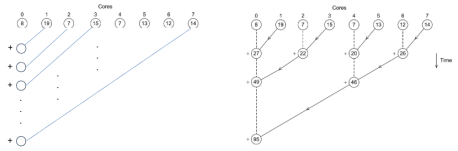
Parallel computing

1.1 The rise of parallel systems

From 1986 to 2023, microprocessor performance has increased by 50% per year, although, due to physical limitations, this increase has slowed down since 2003, leading to the "abandonment" of powerful monolithic systems in favour of placing multiple processors on a single circuit.



However, since sequential programs cannot benefit from multicore systems, it can come in handy to convert these codes into parallel constructs, but, since this approach tends to be inconvenient, it is more convenient to devise brand new algorithms that can effectively exploit parallelism.
Example: Suppose you want to compute the sum of n integers: by employing $p << n$ cores, an easy way of parallelizing this task consists in having each core compute the sum of $\frac{n}{p}$ numbers and, when all the partial sums are ready, a "master" core will deal with providing the global sum of the numbers.
However, since this implementation leaves most of the work to the master core, it is also possible to opt for a tree reduction that, in order to increase activity, computes pairwise partial sums.



1.2 Common parallelism strategies

In order to improve coordination and synchronization, parallel programs typically exploit task parallelism, which consists in partitioning tasks among cores, or data parallelism, which consists in partitioning data among cores so that each processor can work on its chunk of data.
Example: Suppose a professor with three assistants has to grade 300 exams with 15 questions each: applying data parallelism would have each assistant grade 100 exams, whereas task parallelism would have each assistant grade just some questions for each exam.

1.3 Writing parallel programs

Generally speaking, it is possible to write parallel programs by using some dedicated APIs provided by the C programming language.

	Shared Memory	Distributed Memory
SIMD	CUDA	
MIMD	Pthreads/ OpenMP/ CUDA	MPI

1.3.1 Implementing memory structures

Regarding memory structures, a parallel system can implement a shared memory structure, where cores can access a common memory space, or a distributed memory structure, which provides a private memory for each core and thus coordinates activity by message passing over a dedicated network.

1.3.2 Implementing control units

Regarding control units, a parallel system can opt for a multiple-instruction multiple-data approach, where each core has its own control unit and can work independently of the other cores, or a single-instruction multiple-data approach, where cores share the same control unit and can either execute the same instructions on different data or stay idle.

1.4 Classifying multicore systems

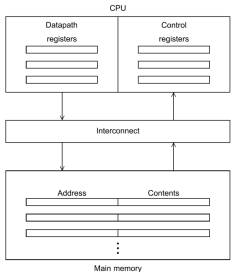
Generally speaking, multicore systems tend to experience performance benefits compared to single-core system.
In particular, it is possible to classify a multicore system in one of the following groups:

- **Concurrent systems:** the system allows multiple tasks to run at any time, although said tasks can still be independent.
- **Parallel systems:** the tasks cooperate closely, typically by means of a shared memory or a communication network, in order to solve a common problem.
- **Distributed systems:** the tasks can cooperate to solve a common problem, but, compared to parallel systems, they work together less frequently.

1.5 The von Neumann architecture

When working on a (parallel) system, it can come in handy to know its high-level architecture in order to simplify the optimization procedure.
Nowadays, most systems are based on the von Neumann architecture, which contains the following components:

- **Main memory:** this structure collects data and instructions, which can be accessed by referring to their address within the memory.
- **Processor:** this structure acts as a control unit as it executes a program with the help of registers, which usually store variables or information about the status of the program, and a program counter, which contains the memory address of the next instruction to execute.
- **Interconnect:** this structure, typically implemented as a bus, allows to connect the main memory with the processor, meaning that it will also need to deal with the "von Neumann bottleneck", which represents a limitation concerning the communication capability of the system.



Chapter 2

Message passing interfaces

2.1 The main aspects of message passing interfaces

Implemented for multiple-instruction-multiple-data structures with a distributed memory, a message passing interface is a library that is commonly described as a "single-program-multiple-data" computational model because it compiles one program that gets executed by multiple processes, whose individual functionalities are specified using selective branches with respect to their ranks.
Example: The following code, which makes use of the "mpi.h" library, prints "Hello, World!" using a message passing interface:

```c
#include <stdio.h>
#include <mpi.h> // This library is needed to implement message passing interfaces.

int main(void) {
    MPI_Init(NULL, NULL); // MPI_Init() sets up the program.
    printf("Hello, World!\n");
    MPI_Finalize(); // MPI_Finalize() terminates the program and provides its exit code.
    return 0;
}
```

2.2 Grouping processes in communicators

In the context of message passing interfaces, the term "communicator" refers to a collection of processes that can send messages to each other.
Generally speaking, a message passing interface always creates a "communicator world" that will contain all the processes that are created when the corresponding program is started.
Most particularly, if more communicators are created, each process will have different IDs relative to each communicator.
Example: The following code makes use of a communicator that allows four processes to print "Hello, World!":

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

int main(void) {
    int comm_sz, my_rank;
    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // MPI_Comm_size() returns the number of processes in the communicator.
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // MPI_Comm_rank() returns the rank of the caller process.
    printf("Hello, World - From process %d out of %d \n", my_rank, comm_sz); //
    MPI_Finalize();
    return 0;
}
```

Observe that, since process scheduling will be non-deterministic, it might be the case that these processes do not print their messages in order.
However, it is possible to handle this issue by calling some functions that deal with message passing, as shown in the following code:

```c
#include <stdio.h>
#include <string.h>
#include <mpi.h>

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz;
    int my_rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Here, process 0 will receive and print all the greetings.
    // On the other hand, the other processes will send a greeting to process 0.
    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d out of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        // Start by printing process 0 as well.
        // This is better than placing process 0 in the loop as it will block instead.
        sprintf(greeting, "Greetings from process %d out of %d!", my_rank, comm_sz);
        printf("%s\n", greeting);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;
}
```

2.3 Handling communications by messages

When implementing a message passing interface, processes can communicate via message passing by using dedicated functions that coordinate both the sender and the receiver.
In particular, message passing interfaces require messages to be non-overtaking, which means that, if process q sends two messages to another process r, then the first message that was sent first by q should be available to r before the second one.
N.B.: The constraint on messages being non-overtaking does not necessarily apply whenever the messages are sent from different processes

2.4 Message matching

When two processes communicate over a message passing interface, a message will be successfully received when all of the following conditions are satisfied:

- The source and destination ranks match between sender and receiver.
  If the sender is unknown, the receiver can refer to a wildcard that allows

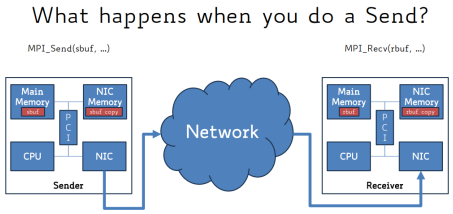
it to receive a message without setting a constraint on the sender's rank: whenever this happens, the receiver can recover the rank of the sender from the "status" structure of the receiving function.

- The sender and the receiver refer to the same communication tag.
  If the tag is unknown, the receiver can refer to a wildcard that allows it to receive a message without setting a constraint on the sender's rank: whenever this happens, the receiver can recover the tag of the sender from the "status" structure of the receiving function.
- The sender and the receiver agree on the data type of the message.
- The receiving buffer is larger than the sending buffer.

Generally speaking, however, the message matching procedure may behave differently depending on the implementation of the message passing interface.
For this reason, when working with a message passing interface, it is more convenient to coordinate message matching according to the default standard to preserve portability.

2.4.1 Hanging processes

When dealing with message passing, a process is said to "hang" if it stops functioning and does not respond to user inputs.
Generally speaking, a sending process will hang if its call blocks and there is no matching receiver: if, under this condition, the call is then buffered, the message will be lost.
On the other hand, a receiving process will hang if it tries to receive a message but there is no matching sender.



For this reason, when implementing a message passing interface, it is a good practice to define as few senders as possible.

2.5 Communication modes

When sending a message, the sender typically uses the standard communication mode, which lets the process choose whether to block globally, meaning that the

sender will block until the receiver collects the message, or locally, meaning that the sending call will return before a matching receiver is found.
Alternatively, in the context of blocking communication, the sender can also opt for one of the following communication modes:

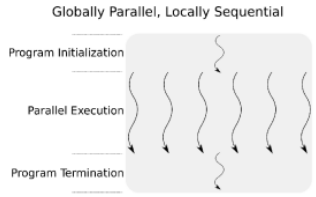
- **Buffered:** the sending call is always locally blocking as it will return before the message is copied in a user-provided buffer.
- **Synchronous:** the sending call is always globally blocking as it will return after the receiver has started retrieving the message.
- **Ready:** the sending call is successful if and only if the receiver has already started retrieving the message.
  Generally speaking, this communication mode comes in handy for reducing the communication overhead.

However, since buffered communication is typically considered to be inefficient, most programs now opt for non-blocking communication modes that maximize concurrency by letting communication and computation overlap: this strategy is faster, but it requires explicit control of operations to let the sender re-use or modify the buffer and to allow the receiver to extract the message.
Example: The following code provides a non-blocking implementation of a ring topology where each process sends and receives something from its left and right neighbours:

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

int main(void) {
    int numtasks, rank, next, prev, buf[2];
    MPI_Request reqs[4]; // Required variable for non-blocking calls.
    MPI_Status stats[4]; // Required variable for Waitall routine.

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Find the ranks to determine the left and right neighbors.

    prev = (rank - 1+ numtasks) % numtasks; // To avoid negative ranks for process 0.
    next = (rank + 1) % numtasks;

    // Post non-blocking receives and sends for the neighbors.
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&rank, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[3]);

    // Wait for all non-blocking send/receive operations to complete before exiting.
    MPI_Waitall(4, reqs, stats);

    MPI_Finalize();
    return 0;
}
```

2.6 Parallel design strategies

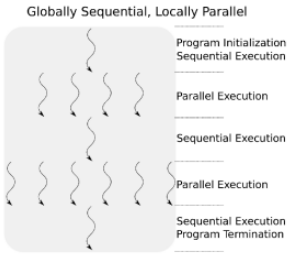
Generally speaking, it is possible to exploit common patterns in an application in order to parallelize the programs it wants to run.
In particular, an application is said to be "globally parallel, locally sequential" if it can perform sequential tasks concurrently.



This type of application is typically implemented according to one of the following patterns:

- **Single program, multiple data:** application logic is kept in a single program that can be run by multiple processes whose specific functionalities can be determined based on their rank.
  While this pattern is relatively easy to implement, it tends to be less flexible as it cannot be implemented the memory requirements for each node are very large or if multiple architectures are involved.
- **Multiple program, multiple data:** this pattern tries to overcome the limitations of the single program approach by allowing processes to run on different programs.
- **Master-worker:** this pattern involves a "master" process, which distributes the workload among processes and collects their partial results, and one, or more, "worker" processes that perform computations.
- **Map-reduce:** this pattern is considered as a variant of the master-worker approach where each node can perform either a map, which consists in working on data to achieve a partial result, or a reduce, which consists in collecting the partial results to obtain the final result.

Alternatively, an application is said to be "globally sequential, locally parallel" if it runs sequentially, but, when needed, it can cooperate with other threads/processes to perform parallel computations.

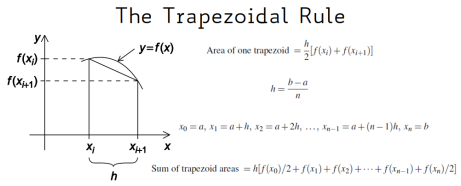


This type of application is typically implemented according to one of the following patterns:

- **Fork/Join:** computations start from a single parent thread that, when needed, can cooperate with other child threads, which can be created dynamically or invoked from a static thread pool.

- **Parallel loops:** this pattern consists in distributing iterative tasks among more processes that can execute them in parallel.
  While parallelizing a loop can be easy, this approach is less flexible because it typically requires the loop to have certain structural properties.

N.B.: While these concepts are presented for message passing interfaces, they can be easily generalized for other frameworks as well.
Example: Suppose that you want to parallelize the trapezoidal rule to approximate the area of a function $f(x)$ over an interval of length h.



Assuming n processes are used, one could ask each process to compute a part of the total area and, at the end, a "master" process can sum these values to obtain the final result.

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

// Declare a function to compute the area of each trapezoid.
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len);

int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    h = (b - a) / n; // All processes use the same height.
    local_n = n / comm_sz; // Define the number of trapezoids.

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    // Here, process 0 will gather and sum the partial integrals computed by the other processes.
    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    } else {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
            total_int += local_int;
        }
    }

    if (my_rank == 0) {
        printf("With n = %d trapezoids, the estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    }

    MPI_Finalize();
    return 0;
}

double Trap(double left_endpt, double right_endpt, int trap_count, double base_len) {
    double estimate, x;
    int i;

    // Assume f is a well-known function defined a priori.
    estimate = (f(left_endpt) + f(right_endpt)) / 2.0;
    for (i = 1; i <= trap_count - 1; i++) {
        x = left_endpt + i * base_len;
        estimate += f(x);
    }
    estimate = estimate * base_len;

    return estimate;
}
```

While this implementation is fairly simple, it tends to be inefficient due to the workload imbalance between processes.

2.7 Reading user inputs in message passing interfaces

When working with message passing interfaces, user inputs are generally handled by the "master" process, which, after reading the input, is able to broadcast it to other processes.
Example: The following code can help to make the parallel implementation of the trapezoidal rule more efficient by allowing users to input the number of trapezoids they want to use:

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

void get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p) {
    int dest;

    // Here, process 0 will get the user input and send it to other processes.
    if (my_rank == 0) {
        printf("Enter a, b and n: \n");
        scanf("%lf %lf %d", a_p, b_p, n_p);

        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

2.8 Collective communication

One of the main issues of parallel communication lies in the fact that, generally speaking, most of the workload is left to the master process.
For this reason, it is possible to make parallel communication more efficient by implementing collective operations that allow more processes to participate to the same routine in order to perform (or even combine) operations like reduction or broadcasting.
Keep in mind, however, that, to avoid hanging or crashing situations, each process in the communicator should always refer to the same collective function:

Example: The following code tries to make the parallel implementation of the trapezoidal rule more efficient through collective communication:

```c
int main(void) {
    int my_rank, comm_sz, n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    get_input(my_rank, comm_sz, &a, &b, &n);

    h = (b - a) / n; // All processes use the same height.
    local_n = n / comm_sz; // Define the number of trapezoids.

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    // Now, use MPI_Reduce() to sum up the partial integrals.
    // Then, process 0 will print the final result.
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("With n = %d trapezoids, the estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    }

    MPI_Finalize();
    return 0;
}
```

In fact, it is possible to observe that, by implementing this version using a "butterfly algorithm", which lets sending and receiving operations take place at the same time, the procedure becomes two times faster.

2.9 Derived datatypes

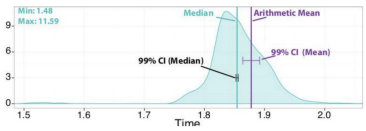
Derived datatypes are used to simulate structs by storing the type and memory offset of each item in the collection, allowing functions that need to send/receive data to collect these items.
N.B.: When the collection is not needed anymore, the memory space that was allocated to it should be freed up.

Chapter 3

Parallel performance

3.1 Evaluating the performance of parallel applications

Generally speaking, evaluating the performance of a parallel system is a complicated task.
In fact, while one could estimate the running time of the application using the slowest process, this estimate tends to be unreliable as the processes might start at different times.
For this reason, when implementing a parallel system, it is a good practice to implement a barrier to makes sure that processes will start at approximately the same time.
Keep in mind, however, that running an application just once is not sufficient to evaluate it as the performance may be affected by interference coming from other applications or from communication networks.
Therefore, it is a good practice to run the application more times and evaluate its performance in terms of the runtime distribution.

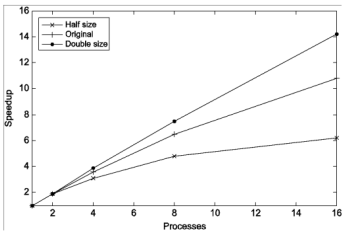


3.1.1 The speed-up factor of a parallel application

Assuming $T_{seq}(n)$ denotes the running time of a sequential application on a problem of size n while $T_{par}(n, p)$ denotes the running time of a parallel application using p processes on that problem, it is possible to define the speed-up factor as the performance improvement obtained by parallelizing the application, which can be measured as follows:

$$S(n,p) = \frac{T_{seq}(n)}{T_{par}(n,p)}$$

Ideally, parallelizing an application should provide a linear speed-up, which is achieved whenever $S(n,p) = p$, although, most of the time, parallelization brings significant benefits only for problems of larger size.



Alternatively, it is possible to evaluate the performance of a parallel application in terms of its efficiency, which measures how well the resources are used through the following formula:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{seq}(n)}{pT_{par}(n,p)}$$

Ideally, a parallel application should achieve $E(n,p) = 1$, although, generally speaking, this requirement is not feasible to meet due to the overhead of increasing processes.

3.1.2 The scalability of a parallel application

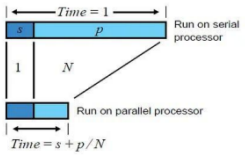
The scalability of a parallel application is defined as the performance improvement that is obtained by passing from one process to p processes, which can be measured as follows:

$$S(n,p) = \frac{T_{par}(n,1)}{T_{par}(n,p)}$$

Most particularly, an application is said to be "strong scaling" if increasing the number of processes but keeping the problem size fixed still allows to improve efficiency.
On the other hand, an application is said to be "weak scaling" if it should increase the number of processes only when the problem size increases as well.

3.2 Estimating the speed-up factor of a parallel application

Generally speaking, each program features a "serial fraction" that cannot be parallelized, resulting in a limit to the speed-up factor that can be achieved by parallelizing the application.



In fact, it is possible to reformulate the running time of a parallel application using p processes through the following weighted average:

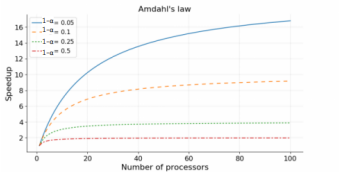
$T_{par}(p) = (1-\alpha)T_{seq} + \alpha\dfrac{T_{seq}}{p}$, where $0 \leq \alpha \leq 1$ is the parallelizable fraction.

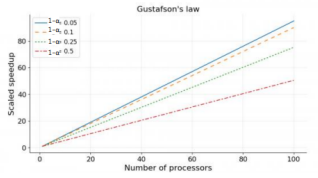
Most particularly, Amdahl's law exploits this formulation in order to provide an asymptotic estimate of the speed-up factor as follows:

$$S(p) = \frac{T_{seq}}{(1-\alpha)T_{seq} + \alpha\frac{T_{seq}}{p}} \text{ and take } \lim_{p\to\infty} S(p) = \frac{1}{1-\alpha}$$

Keep in mind, however, that Amdahl's law can only be used to explain the speed-up factor of strong scaling applications.
In fact, in the context of weak scaling applications, it is possible to observe that the parallel fraction tends to increase with the problem's size.
For this reason, Gustafson's law considers the following scaled speed-up factor:

$S(n,p) = (1-\alpha) + \alpha p$, where $0 \leq \alpha \leq 1$ is the parallelizable fraction.





Example: Suppose you want to parallelize the sum of two vectors $\vec{x}$ and $\vec{y}$.
In the context of message passing interfaces, the master process can use a scatter to read and distribute pieces of the vector to the other working processors, each computing a partial sum.

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

// Try to parallelize the computation of the sum.
void parallel_vector_sum(double local_x[], double local_y[], double local_z[], int local_n) {
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++) {
        local_z[local_i] = local_z[local_i] + local_y[local_i];
    }
}

// Use MPI_Scatter() to read a vector and distribute it among the working processes.
void read_vector(double local_a[], int local_n, int n, char vec_name[], int my_rank, MPI_Comm comm) {
    double* a = NULL;
    int i;

    // Here, process 0 reads the input vector and scatters it among the other processes.
    if (my_rank == 0) {
        a = malloc(n * sizeof(double));
        printf("Enter the vector %s\n", vec_name);
        for (i = 0; i < n; i++) {
            scanf("%lf", &a[i]);
        }
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
        free(a);
    } else {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    }
}
```

Then, the master process can perform a gather in order to collect the partial sums and recover the final result.

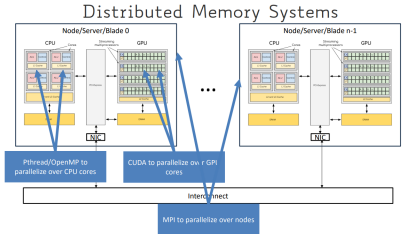
```c
void print_vector(double local_b[], int local_n, int n, char title[], int my_rank, MPI_Comm comm) {
    double* b = NULL;
    int i;

    // Here, process 0 will gather and print all components of the vector.
    if (my_rank == 0) {
        b = malloc(n * sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++) {
            printf("%f ", b[i]);
        }
        printf("\n");
        free(b);
    } else {
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    }
}
```

Chapter 4

Threads

4.1 Defining the main aspects of threads

When dealing with distributed memory systems, it can come in handy to parallelize computations by implementing a message passing interface and, at the same time, each server handles its CPU/GPU activities using threads that are able to communicate through a shared DRAM.



Generally speaking, threads are chosen over processes because, since they represent the smallest independent instance of a (running) program that can be executed by a computer, they tend to be similar to light-weight processes.
Nowadays, most operating systems use threads by referring to the POSIX Threads standard, which, however, is typically implemented using the OpenMP API for better portability.
Overall, a program is said to be "thread-safe" if it can be simultaneously executed by multiple threads without causing problems.

4.2 Race conditions

A race condition is a problematic situation where the result of a program depends on the non-deterministic scheduling of the threads executing the critical

section, resulting in data inconsistencies or unexpected outcomes.
For this reason, it is a good practice to implement a countermeasure, such as a lock, that makes sure that only one process at a time can access the critical section of a program.
A first attempt at thread safety can be obtained by applying busy waiting, which is a countermeasure where a thread can access the critical section only when a certain condition, typically relative to its rank, is met.
While this approach is relatively easy to implement, it is considered to be inefficient as the thread always wastes CPU cycles without doing anything.
In fact, it is typically more convenient to handle threads using a mutex, which is a special generalization of a lock that guarantees mutual exclusion by restricting the access to the critical section to just one thread at a time.

4.3 Deadlock and starvation

Starvation is a problematic situation where, due to unfair scheduling, the execution of a thread is suspended for an indefinite amount of time, although, eventually, the thread will still be able to continue its execution.
On the other hand, a deadlock is a more dangerous situation where no thread can proceed because each entity waits for another entity (or itself) to perform a certain action, resulting in a case of cyclical wait.

4.4 Properly using threads in message passing interfaces

Another main issue of message passing interfaces lies in the fact that communication is not thread-safe.
For this reason, when working with threads, it is important to understand which threading level is supported by the message passing interface and whether this level matches with the required threading level.
Generally speaking, message passing interfaces can support one of the following threading levels:

- **Single:** the message passing interface does not support, meaning that the message passing interface will be used just by one main thread.
- **Funneled:** multiple threads can be used, but only the master thread can interact with the message passing interface.
- **Serialized:** multiple threads can interact with the message passing interface, although, at any given time, only one can communicate with the message passing interface.
- **Multiple:** multiple threads can interact with the message passing interface, even simultaneously, although this approach tends to be less efficient compared to funneled or serialized threading.

Chapter 5

Shared memory systems

5.1 The main aspects of the OpenMP API

OpenMP is a multiprocessing API that is normally used to implement shared-memory systems.
In particular, this API aims to decompose a sequential program into simpler components that can be performed in parallel through some compiler directives. For this reason, OpenMP programs, which are implemented using the Fork/join design pattern, are said to be "globally sequential, locally parallel".

5.2 Pragmas

A pragma is a special preprocessor directive that gives additional information to the compiler in order to instruct it to perform non-basic tasks for parallelizing a sequential code.
N.B.: If the compiler does not know how to read a pragma, it will simply ignore it for portability purposes.
Example: The "parallel" pragma is the simplest directive as it is able to take a sequential code and have it performed in parallel by more threads, which will work on different portions of the data.
For instance, the following code implements a function to print "Hello, World!" from more threads using the OpenMP API:

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

void Hello(void);

int main(int argc, char* argv[]){
    int thread_count = strtol(argv[1], NULL, 10); // Convert the input into a long.

    # pragma omp parallel num_threads (thread_count)
    Hello ();

    return 0;
}

void Hello (void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

5.2.1 Using clauses to specify the number of threads

A clause is a text that allows to modify the behaviour of a compiler directive in order to let the programmer specify how many threads should be used to run an OpenMP program.
Generally speaking, it is possible to specify the number of threads needed to run an OpenMP program using one of the following approaches:

- **Universally:** the number of threads is specified using an environmental variable that indicates the default number of threads for all the parallel regions within the program.
- **Program level:** the number of threads is specified within the program using a dedicated OpenMP function.
- **Pragma level:** the number of threads is specified using a directive that, through a dedicated clause, indicates the number of threads that can be used for a specific parallel region.

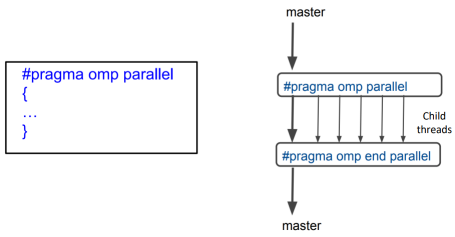
Keep in mind, however, that the actual number of threads that can be used is typically bounded by system and hardware limitations.

5.2.2 Using pragmas to implement mutual exclusion

In order to avoid race conditions, the "critical" pragma allows to implement mutual exclusion to make sure that only one thread at a time can execute the following parallel section.

5.3 Thread terminology in OpenMP APIs

A group of threads that executes a parallel section is known as a "team".
Among the threads of a team, the starting thread, which is known as the "master thread", typically acts as the "parent thread" as, through a directive, it is able to create "child threads" to form the team.



N.B.: When working with variables defined within a parallel construct, each thread works on its own "local" copy, meaning that modifications made by one thread will not be visible to the other threads.

Example: The following code provides an OpenMP implementation for the trapezoidal rule used to approximate integrals:

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;
    double a, b;
    int n;
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b and n: \n");
    scanf("%lf %lf %d", &a, &b, &n);

    #pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With %d trapezoids, the estimate of the integral is %.14e.\n", n, global_result)

    return 0;
}
```

```c
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b - a) / n;
    local_n = n / thread_count;
    local_a = a + (my_rank * local_n * my_rank);
    local_b = local_a + (local_n * h);
    my_result = (f(local_a) + f(local_b)) / 2.0; // Assume f is a well-known function.
    for (i = 1; i <= local_n - 1; i++) {
        x = local_a + (i * h);
        my_result += f(x);
    }
    my_result = my_result * h;

    #pragma omp critical
    *global_result_p += my_result; // Critical section.
}
```

5.4 Variable scope in OpenMP programs

In the context of serial programming, the scope of a variable simply refers to all sections of a program where that variable can be accessed and used.
Regarding OpenMP programs, however, the scope of a variable indicates both its accessibility and whether the variable is public, which means that all threads can access it, or private, which means that each thread accesses an independent instance.

5.4.1 Scope-modifying clauses

The "default" clause allows the programmer to specify the default scopes of the variables within a parallel construct.
In addition, starting from a situation in which no variable is accessible, the following clauses can be used to handle variable scope:

- **Shared:** the variable is shared among all threads.
  Observe that this is the default behaviour for variables defined outside of a parallel block, although one can explicitly indicate it using the "default" clause.

- **Reduction:** the variable participates in a reduction operation where each thread has a private copy of the variable and the final result is reduced into a single variable found after the parallel block.
- **Private:** each thread receives its separate instance of the variable.
  Keep in mind, however, that, since these copies will be uninitialized, they will not maintain the value of the original variable outside the parallel block.

5.5 Reduction operations in OpenMP programs

In the context of OpenMP programs, a reduction is a computation that repeatedly applies a reduction operator, such as addition or multiplication, to a sequence of operands in order to recover a single result that is obtained by grouping the intermediate results into a dedicated reduction variable.
Reductions are widely used in parallel programming because they can be easily implemented using the "reduction" clause, which allows the OpenMP API to handle computations more efficiently compared to a critical section.

5.6 Parallel loops

Using a dedicated pragma, OpenMP programs are able to parallelize sequential loops by distributing iterations among the threads of a team.
However, since the runtime system needs to know the exact number of iterations in advance, this approach only works with for loops as they can provide a clear iteration count.
Most particularly, the control variables should remain fixed, with the index variable changing only with the increment specified in the loop's header.

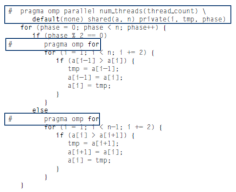
Legal forms for parallelizable for statements

```
                                  index++
                                  ++index
                      index < end     index--
                      index <= end    --index
for  index = start  ; index >= end ;  index += incr
                      index > end     index -= incr
                                  index = index + incr
                                  index = incr + index
                                  index = index - incr
```

Why? It allows the runtime system to determine the number of iterations prior to the execution of the loop

In addition, loops containing a break/return statement cannot be parallelized as the system cannot know when (or if) the condition to exit the loop will be met, unless the exit() call, which terminates the entire program, is used.
Example: The odd-even sorting algorithm sorts an array using an odd phase, which compares and, if needed, swaps adjacent pairs where the first element has an odd index, and an even phase, which compares and, if needed, swaps

adjacent pairs where the first element has an even index.
To achieve better efficiency, it is possible to implement this algorithm using parallel loops, as shown in the following code:



5.6.1 Parallelizing nested loop structures

When a program tries to parallelize a nested loop structure, it is often sufficient to parallelize the outermost loop while keeping the inner loop unchanged.
Sometimes, however, the outer loop runs for too few iterations to fully use the available threads, resulting in a lesser performance improvement: whenever this happens, it is possible to use the "collapse" clause to rewrite the structure as a single loop that is executed in parallel.

5.6.2 Handling loop scheduling

Suppose you want to parallelize a loop by distributing n iterations among t threads.
The default OpenMP standard applies static block distribution, although, for some programs, it might be more convenient to opt for a cyclic distribution.

Thread	Iterations (Default)
0	$0, \ldots, \frac{n}{t} - 1$
1	$\frac{n}{t}, \ldots, \frac{2n}{t} - 1$
$\ldots$	$\ldots$
$t-1$	$\frac{n(t-1)}{t}, \ldots, n-1$

Thread	Iterations (Cyclic)
0	$0, \frac{n}{t}, \frac{2n}{t}, \ldots$
1	$1, \frac{n}{t}+1, \frac{2n}{t}+1, \ldots$
$\ldots$	$\ldots$
$t-1$	$t-1, \frac{n}{t}+t-1, \frac{2n}{t}+t-1, \ldots$

Generally speaking, it is possible to handle loop scheduling using the "schedule" clause, which specifies both the scheduling type and the number of loops that are assigned to each thread.
In particular, the scheduling type can be chosen among the following options:

- **Static:** the iterations are assigned to threads before the loop is executed.
- **Dynamic:** the iterations are assigned to threads as the loop is executed. In particular, each thread executes an iteration chunk on demand and,

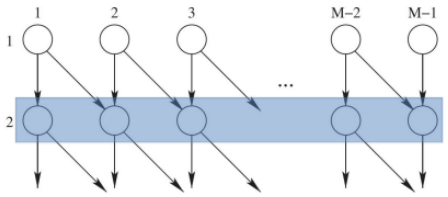
when done, it can request another one from the run-time system, allowing to improve workload balance.

- **Guided:** this scheduling type is a variant of dynamic scheduling in which the chunk size decreases as more iteration chunks are completed.
- **Auto:** the scheduling type is decided by the compiler or by the run-time system.
- **Runtime:** the scheduling type is defined by the programmer at run-time through a dedicated environmental variable.
  Generally speaking, static scheduling is suitable for homogeneous iterations, whereas dynamic scheduling is more convenient for varying execution costs.

5.6.3 Handling data dependencies for parallel loops

One of the main issues of parallel loops lies in the risk of having (loop-carried) data dependencies among iterations, which do not allow to correctly parallelize some parts of the loop.
Generally speaking, a parallel loop can give rise to four main types of dependencies:

- **Read after write:** an iteration needs to read a value that was produced by a previous iteration, setting a limit to loop parallelization.
  This type of dependence can be handled in one of the following ways:
  - **Variable fix:** identify variables whose values predictably change across iterations and transform them into reduction/induction variables to safely parallelize the loop.
  - **Loop skewing:** rearrange the loop statements in order to make sure that they use values in a way that prevents, or at least limits, dependencies.
  - **Partial parallelization:** introduce an iteration space dependency graph, whose nodes indicate loop iterations and whose edges represent dependencies, and use it to find dependence-free loops that can be safely parallelized.



  - **Refactoring:** rewrite the loop(s) in order to spot parallelizable iterations.
  - **Fissioning:** split the loop into a sequential part and a parallel part.
  - **Algorithm change:** if none of the above methods is effective, it is typically more convenient to opt for a different algorithm.

- **Write after read:** an iteration writes to a memory location that a previous iteration still needs to read.
  Generally speaking, it is possible to handle this dependence by making a copy of a value before actually modifying it.

- **Write after write:** two different iterations write to the same memory location, meaning that the final result will depend on the order of execution.
  Generally speaking, it is possible to handle this dependence by refactoring the loop so that different writing operations write into different memory locations, therefore eliminating the risk of conflicts between operations.

- **Read after read:** multiple iterations read the same location without modifying it.
  Compared to other data dependencies, this dependence is not really problematic and it does not prevent parallelization.

5.7 Synchronization constructs

Generally speaking, a synchronization construct is a mechanism that coordinates the execution of multiple threads working in parallel.
In particular, OpenMP provides the following synchronization pragmas:

- **Master/Single:** these directives force the execution of a structured block by a single thread.
  Note that "single" also implies that there is a barrier immediately after the structured block.

- **Barrier:** this directive guarantees that the execution of a code can go on if and only if all threads reach the barrier, providing a useful synchronization point.

- **Section:** this directive divides a program in different sections that will be executed by different threads in parallel.

- **Ordered:** this directive is used inside a parallel for loop to execute some blocks sequentially in their original order.

5.8 Implementing atomic operations in OpenMP

Using a dedicated pragma, it is possible to efficiently perform some atomic operations without the need for a longer critical section.
Alternatively, when dealing with longer critical sections, the OpenMP API allows to assign a name to each critical directive so that the corresponding constructs can be performed in parallel, provided different threads enter different named critical sections.
Keep in mind, however, that the compiler always needs to know the names of the critical sections beforehand before being able to parallelize them.

5.8.1 Choosing the best approach

Generally speaking, the atomic directive represents the fastest method for obtaining mutual exclusion, although, in some cases, it might be more appropriate to enforce mutual exclusion using other directives.
On the other hand, locks should mainly be used for handling critical sections involving data structures rather than on codes.
In particular, it is a good practice to avoid mixing or nesting different mutual exclusion mechanisms as, similarly for the PThreads library, there is no guarantee of fairness in the mutual exclusion constructs.

5.9 Handling interactions between OpenMP and MPI

While message passing interfaces define several levels of thread safety, the funneled implementation is generally considered to be the safest for programs that integrate message passing interfaces with OpenMP.
In fact, the master thread is able to efficiently coordinate more complex operations in order to provide a nice balance between functionality and performance.

5.10 Caching in single-core architectures

A cache is a small and fast memory component that stores copies of data from slower memory resources, allowing these data to be accessed more quickly.
Caches exploit the locality of reference, which comes in two main forms:

- **Spatial locality:** if a memory address is accessed, nearby memory locations are likely to be accessed soon.
- **Temporal locality:** if a memory address is accessed, it is likely to be accessed again in the near future.

To improve performance, data is transferred between memory and the cache in fixed-size blocks or cache lines.

While this approach introduces some overhead, it benefits performance because accessing data in larger contiguous chunks makes better use of spatial locality and reduces the number of slow memory accesses.

5.10.1 Cache levels

Generally speaking, a cache is organized as a hierarchy consisting of multiple levels, typically known as L1, L2 and L3 cache structures.
Whenever the processor needs to access some data, it searches the cache hierarchy starting from the closest and fastest section, which corresponds to the L1 cache, and going down until the resource is found.
However, if the resource is not present in the cache, the processor will need to fetch it from the main memory, which is significantly slower.

5.10.2 Cache consistency

To avoid the risk of inconsistency between cache and memory, caches make use of policies that determine how and when updates are propagated to the main memory.
Generally speaking, it is possible to handle consistency by implementing a write-through cache, which updates data directly in the main memory, or a write-back cache, which marks data in the cache until they are updated.

5.11 Caching for multicore architectures

When dealing with multicore architectures, different cores my hold copies of the same memory addresses in their caches, requiring the system to ensure that these copies stay consistent over time.

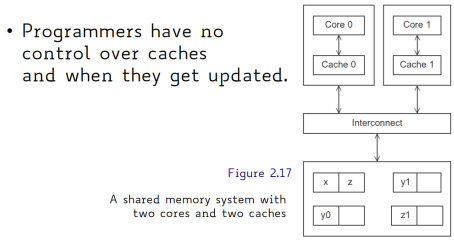


Figure 2.17
A shared memory system with two cores and two caches

Generally speaking, cache coherence in multicore architectures can be handled using one of the following approaches:

- **Snooping:** all processors share a bus and, whenever a processor updates a variable, it will broadcast the information to other processors to allow the to update or invalidate their copies.

- **Directory-based cache coherence:** a directory keeps track of which processors hold each cache line and, whenever a processor updates a cache line, the directory notifies the intended processors to update or invalidate their copies.

However, since invalidating a variable requires invalidating its entire cache line, cache coherence may lead to false sharing, where multiple processors share cache lines without actually accessing data.
A common solution consists in forcing these variables in different cache lines, often by applying additional padding.

- Original

```
double x[N];
#pragma omp parallel for schedule(static , 1)
  for( int i = 0; i < N; i++ )
     x[ i ] = someFunc( x [ i ] );
```

- Padded:

```
double x[N][8];
#pragma omp parallel for schedule(static , 1)
  for( int i = 0; i < N; i++ )
     x[ i ][ 0 ] = someFunc( x [ i ][ 0 ] );
```

Overall, however, applying padding requires using additional memory, often reducing cache efficiency.
For this reason, it might be more convenient to have each processor work on a local copy of the variable and merge their results at the end, allowing to significantly mitigate the issue of false sharing.
Example: The following code provides a PThreads implementation of matrix–vector multiplication:

```
1  void *Pth_mat_vect(void* rank) {
2      long my_rank = (long) rank;
3      int i, j;
4      int local_m = m / thread_count;
5      int my_first_row = my_rank * local_m;
6      int my_last_row = (my_rank + 1) * local_m - 1;
7
8      for (i = my_first_row; i <= my_last_row; i++) {
9          y[i] = 0.0;
10         for (j = 0; j < n; j++) {
11             y[i] += A[i][j] * x[j];
12         }
13     }
14
15     return NULL;
16 }
```

Run the code using a single thread on vectors and matrices of various sizes:

Matrix size denoted as # Rows x # Cols

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000

Notice, however, that, for larger input data, performance is dominated by memory-access costs.
In fact, the input vector might be too large to fit in the cache, requiring frequent transfers from the main memory, which increase the overall execution time.
Now, try running the same code using multiple threads:

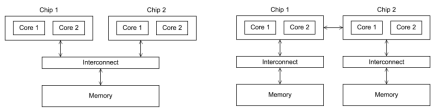
Matrix size denoted as # Rows x # Cols

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

While all the versions experience reduced efficiency, the last configuration shows the largest performance drop as it is the most subjected to false sharing due to having a smaller output vector that could fit within a single cache line.

5.12 Memory organization for multicore architectures

Generally speaking, memory organization for multicore architectures can follow a uniform memory access model, in which all processors share a single memory space with the same access latency, or a non-uniform memory access model, in which processors may access different memory regions, each with their access cost depending on where the data is allocated.

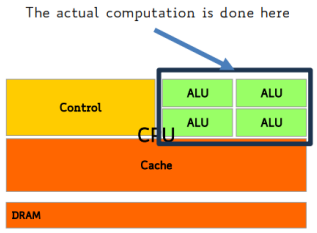


Chapter 6

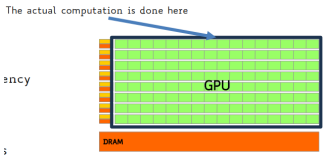
Compute Unified Device Architecture

6.1 The main aspects of the CUDA interface

Due to higher clock frequency and complex control logic, CPUs are designed to be latency-oriented as they implement more complex hardware structures, such as caches and branch predictors, that allow to perform computations in the ALU with minimal delay.



On the other hand, GPUs are designed to be throughput-oriented as they implement simpler control units to make more space for a large number of execution units that can be used in parallel.



For this reasons, CPUs are more suitable for sequential tasks, whereas GPUs

are convenient for parallel tasks.

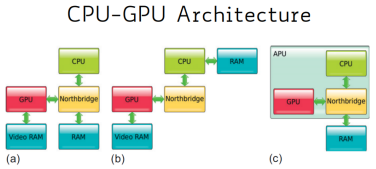
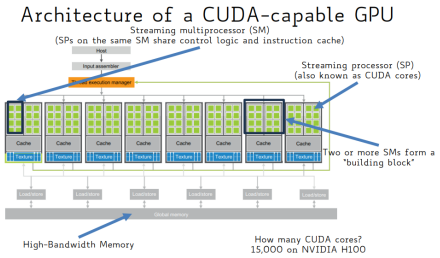


FIGURE 6.1
Existing architectures for CPU-GPU systems: (a) and (b) represent discrete GPU solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to integrated CPU-GPU solutions, such as the AMD's Accelerated Processing Unit (APU) chips.

N.B.: Since GPU and host memories are disjoint, communication will always need some data transfers, although recent versions of CUDA make use of a unified virtual memory that uses common addresses.

6.2 The CUDA architecture

CUDA is a parallel programming platform that provides two API layers for performing parallel computations.
From a high-level perspective, the CUDA architecture consists of several building blocks containing various streaming processors, which are grouped in streaming multiprocessors that share access to a high-bandwidth memory space.

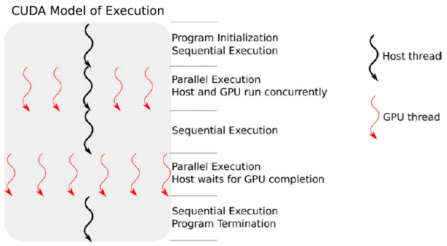


In particular, the CUDA architecture sees the GPU as a co-processor that, using its global memory, can run many threads in parallel to easily perform some tasks.

6.3 The structure of a CUDA program

Generally speaking, the first step of a CUDA program consists in allocating the necessary GPU memory for computations and transferring the required data from the host to the GPU memory.
Then, the program runs a CUDA kernel, which is a function that is executed on

the GPU, and, when done, it copies the output result from the GPU memory to the host memory.

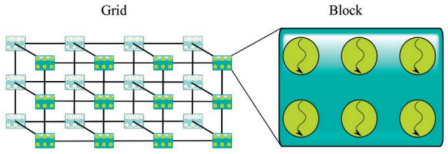


N.B.: I/O operations are typically left to the host.
Example: Writing a CUDA program requires writing a kernel, which gets executed by all the threads, and specifying the grid/block organization for the threads.
For instance, the following code implements a function that prints "Hello, World!" in CUDA:

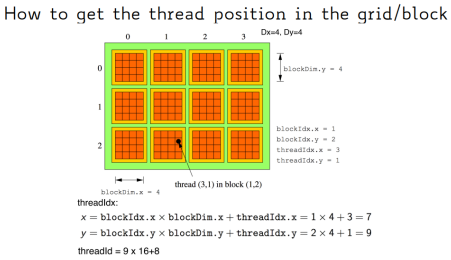
```c
#include <stdio.h>
#include <cuda.h>

// The global decorator indicates that the function must be executed on the GPU.
// In particular, since kernels are declared as void, the computed result should be copied
__global__ void hello() {
  printf("Hello, World!");
}

int main() {
  hello<<<1,10>>>(); // Run hello() using one block containing 10 threads.
  cudaDeviceSynchronize(); // Block until the CUDA kernel terminates.
    return 1;
}
```

6.3.1 Thread organization in a CUDA program

In the context of CUDA programming, threads are organized in blocks, which are arranged in a grid that, depending on the input datatype, can be organized across one, two or three dimensions.



In particular, each thread is able to determine its (global) position within a grid using some intrinsic variables providing size or position information.



6.3.2 Function decorators for CUDA programs

A decorator indicates whether a function should be compiled to be performed on the host or on the GPU.

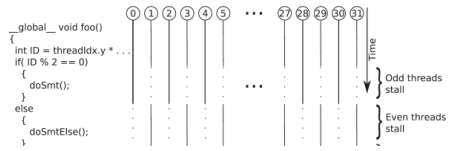
- **Global:** this decorator can be called by the host or by the GPU to specify where the function should be executed.
- **Device:** this decorator indicates that the function runs on the GPU and can only be called from the GPU.
- **Host:** this decorator indicates that the function can only run on the host. Generally speaking, however, this decorator is only used with the device decorator to indicate function that can run on both the host and the GPU.

6.4 Thread scheduling in CUDA programs

When dealing with CUDA programs, each thread runs on a specific streaming processor within a single streaming multiprocessor, which hosts all the threads contained a given block.
In particular, a streaming multiprocessor will (concurrently) run a block on a single instruction and, when done, it will start running a new block.

6.4.1 Scheduling using wraps

Threads within a block are normally grouped in contiguous warps, which typically contain 32 threads.
However, since threads within the same warp must always execute the same assembly task, a warp might experience a divergence when, typically upon a selective branch, some threads within the same warp try to perform different tasks.
In this case, the program must sequentially analyse each possible branch while stalling the threads that do not enter the current path, often leading to a lower usage of the resources.
Example: A warp divergence can happen if the code contains a branch based on whether the current thread's ID is an odd number or an even number.



6.4.2 Context switching for GPUs

A streaming multiprocessor can typically contain more warps that what it is able to execute concurrently, allowing it to efficiently switch between them.
In fact, each thread has a private execution context that is stored directly on the chip, meaning that context switching will not require memory transfers as on CPUs.
In particular, whenever a warp tries to execute an instruction that must wait for a long-latency operation, that warp will not be selected for execution, choosing another ready warp instead.
Therefore, the hardware is able to keep the streaming multiprocessor fully utilized as it can always find a warp that is ready to run.
Example: Suppose a CUDA device supports up to 8 blocks per streaming multiprocessor, with each block containing a maximum of 512 threads, and 1024 threads per streaming multiprocessor.
Using simple 8×8 blocks yields 64 threads per block and, since a streaming multiprocessor can support 8 blocks, it will allow a total of $64 \times 8 = 512$ resident threads, meaning that the device will not fully utilize its resources.
Using larger 16×16 blocks yields 256 threads per block and, since a streaming multiprocessor can support 8 blocks, it will allow a total of $256 \times 8 = 1024$ resident threads, meaning that the device will be able to fully utilize its resources.
Lastly, using 32×32 blocks yields 1024 threads per block, which, however, cannot be supported by the streaming multiprocessor.

6.5 The CUDA memory hierarchy

Remember that, when performing memory allocation operations, data allocated on host memory cannot be seen from the GPU, and viceversa, therefore requiring explicit data transfers during communications.

Memory Allocation and Copy

```
// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr,  // Host pointer address,
                                        // where the address of
                                        // the allocated device
                                        // memory will be stored
                         size_t size )  // Size in bytes of the
                                        // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr );  // Parameter is the host
                                        // pointer address, returned
                                        // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst,     // Destination block address
                         const void* src,  // Source block address
                         size_t count,  // Size in bytes
                         cudaMemcpyKind kind );  // Direction of copy.
```

`cudaError_t` is an enumerated type. If a CUDA function returns anything other than `cudaSuccess` (0), an error has occurred.

Memory Copy Kind

The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost` = 0, Host to Host
- `cudaMemcpyHostToDevice` = 1, Host to Device
- `cudaMemcpyDeviceToHost` = 2, Device to Host
- `cudaMemcpyDeviceToDevice` = 3, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault` = 4, used when Unified Virtual Address space capability is available (see Section 6.7)

From a more general point of view, a CUDA GPU makes use of several memory components, which can be implemented on the chip or off the chip.

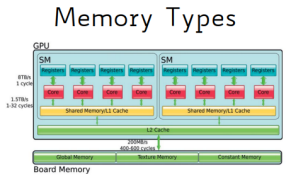


Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
__device__ __shared__ `int SharedVar;`	Shared	Block	Kernel
__device__ `int GlobalVar;`	Global	Grid	Application
__device__ __constant__ `int ConstVar;`	Constant	Grid	Application

6.5.1 The global memory

The global memory is an off-chip memory component that can store large amounts of data but is physically distant from the processors.
For this reason, accessing data from the global memory requires higher latency compared to on-chip memory components.

6.5.2 Registers

Similarly to CPUs, GPU registers are used to store a thread's local variables.
Generally speaking, the available registers on a streaming multiprocessor are evenly distributed among all resident threads, but, whenever the register demand of a kernel exceeds the number of available registers per thread, some local variables will be spilled to the off-chip global memory, which, however, is much slower to access.
Since register usage affects how many threads can be active on a streaming multiprocessor, it will also influence the number of resident warps that can be supported by the streaming multiprocessor.
For this reason, it is possible to define occupancy as the ratio between the number of resident warps and the maximum number of warps that can be supported by the streaming multiprocessor.
Ideally, occupancy should be close to 1 so that the hardware has enough resources to effectively hide latencies.

6.5.3 On-chip shared memory

A shared memory is an on-chip memory component that is shared among threads within a block.
Despite having smaller capacity, the shared memory has a much lower latency compared to other (off-chip) components, making it useful for storing frequently accessed data that, otherwise, would take much longer to fetch.
Example: The following code provides an implementation of a stencil, which is a geometric tool that is often used for solving mathematical problems:

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

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS]; // RADIUS adds a padding on both sides.
    int gindex = threadIdx.x + blockIdx.x * blockDim.x; // Global index in the grid.
    int lindex = threadIdx.x + radius; // Local index in the shared memory.

    // Read input elements into the shared memory.
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads(); // Synchronize all threads within a block to avoid dependencies.

    // Apply the stencil to perform computations.
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++) {
        result += temp[lindex + offset];
    }

    // Store the result.
    out[gindex] = result;
}
```

6.5.4 The constant memory

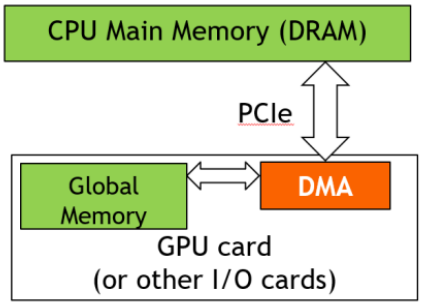
A constant memory is a memory component that is used to store constant values that do not change during execution.
In particular, the constant memory provides performance benefits because it is cached and supports broadcasting when threads within a warp need to access the same memory location.

```cuda
__constant__ type variable_name; // static
cudaMemcpyToSymbol(variable_name, &host_src, sizeof(type), cudaMemcpyHostToDevice);
// warning: cannot be dynamically allocated
```

- data will reside in the constant memory address space
- has static storage duration (persists until the application ends)
- readable from all threads of a kernel

6.6 Host-device data transfers in CUDA

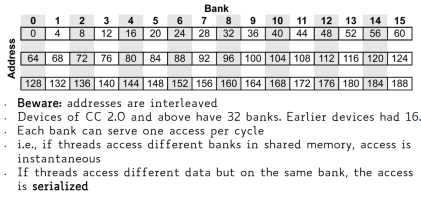
Data transfers in CUDA rely on direct memory access hardware to achieve high efficiency as this approach allows to handle data transfers without involving the CPU, which is able to perform other tasks concurrently.



However, since direct memory access works on physical memory accesses, unmanaged memory access might cause conflicts if different processes try to access overlapping regions in the physical memory.
For this reason, many modern systems make use of virtual memory management, which maps the virtual memory space of a process onto physical memory pages, in order to provide memory isolation and safety during data transfers.
In this context, CUDA enables efficient data transfers through page-locked memory, which consists of virtual memory pages that are guaranteed to stay in the physical memory can cannot be swapped by the system, ensuring stable physical addresses that allow the hardware to directly access the host memory during CPU-GPU data transfers.
Therefore, any source or destination located in the host virtual memory must be allocated as page-locked memory using dedicated allocation functions.
If this is not the case, CUDA can copy the data into a temporary page-locked buffer before performing the actual transfer, providing safety at the cost of some overhead.

6.7 Bank conflicts in the shared memory

Generally speaking, the shared memory of a CUDA device is organized into a set of memory banks.
In particular, memory address are typically interleaved across different blocks so that different threads accessing different banks can perform their accesses simultaneously.
On the other hand, in the event of a bank conflict, which happens when different threads try to access different addresses in the same bank, the accesses must be serialized, leading to reduced performance.

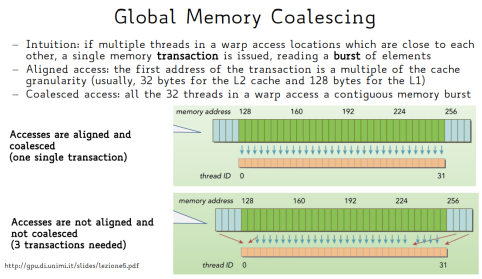


N.B.: If all threads in a warp try to access the same memory address, it is more convenient for the address to perform a "broadcast read" and propagate the value to all threads, therefore avoiding any conflict

6.8 Global memory coalescing

When a thread accesses a global memory location, the hardware actually reads a contiguous block of memory addresses.
For this reason, when all threads in a warp try to execute a load instruction,

the hardware should check whether these threads are accessing contiguous location and, if this is the case, it will coalesce the individual accesses into a single memory access, therefore improving efficiency.



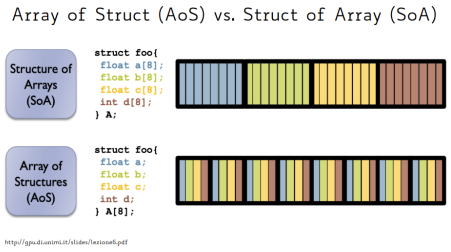
From a broader perspective, global memory accesses in CUDA support two main types of data loads:

- **Cached load:** used by default for devices with a L1 cache, this approach checks whether the desired item is located in the cache before accessing the global memory.
- **Non-cached load:** used for devices without a L2 cache, this approach checks whether the desired item is located in the L2 cache before accessing the global memory.

N.B.: In some cases, it is possible to disable the L1 cache and directly use non-cached loads in order to improve performance.

6.8.1 Data structures for coalesced accesses

When dealing with coalesced memory accesses, data in the memory can be organized either as an array of structs, in which each element is a complete structure containing all data fields, or as a struct of arrays, in which each data field is stored in a separate array.

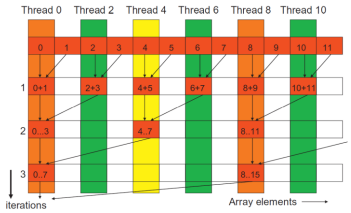


Generally speaking, GPUs tend to benefit more from using a struct of arrays because this layout allows threads in a warp to access contiguous memory addresses for a specific field, therefore improving efficiency for coalesced accesses.

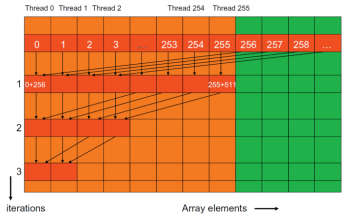
On the other hand, CPUs tend to benefit more from using an array of structs because this layout improves locality when dealing with all fields of a single element.

6.9 Reduction operations in CUDA

While atomic operations can be used to implement reductions in CUDA, they often suffer from poor performance because, at each step, only half of the threads actually performs computations, resulting in a utilization drop.



For this reason, it is convenient to structure reductions so that threads within the same warp will always execute the same task, although only a part of the used threads may actually contribute to computations as the reduction progresses, therefore improving utilization.



Example: The following code provides an efficient implementation of a reduction operation on an array:

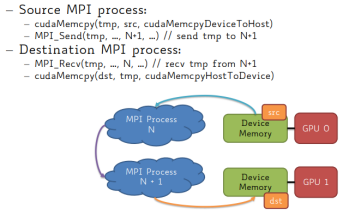
```
#include <stdio.h>
#include <cuda.h>

__shared__ float partialSum[SIZE];
partialSum[threadIdx.x] = X[blockIdx.x * blockDim.x + threadIdx.x];

unsigned int t = threadIdx.x;
for (unsigned int strid = blockDim.x / 2; stride >= 1; stride = stride>>1) {
    // N.B.: The loop divides the stride by 2 at each iteration.
    __syncthreads();
    i (t < stride) {
        partialSum[t] += partialSum[t + stride];
    }
}
```

Note that, if the number of elements in the array is larger than the number of

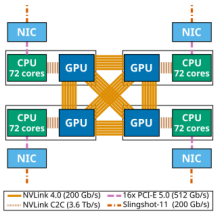
threads within a block, it is possible to introduce some additional kernels and work on partial results that will be merged at the end.

6.10 GPU data transfers

When dealing with distributed GPU applications, data transfers between GPUs are often handled using message passing interface.
In particular, if the message passing interface is not GPU-aware, data must be explicitly copied from the sender device to the host memory before making the desired call.
On the receiving side, data is transferred from the host to the receiver device.



On the other hand, if the message passing interface is GPU aware, it can directly access device memory, allowing to handle communications without having to involve the host memory during data transfers.



6.11 Performance estimation for GPUs

Generally speaking, GPU performance is measured in terms of the number of floating-point operations it can perform in one second.
In practice, however, the performance of a program is limited by the data transfer rate, causing the application to be "memory-bound".
For this reason, it can come in handy to define the compute-to-global-memory-access ratio, which measures the number of floating-point operations that are executed per global memory access within a certain region of the program.

Appendix A

The C programming language

A.1 The basics of the C programming language

Every C program requires a header, which invokes the preprocessor before compiling time and declares input/output functions, and the main() function, which executes the program, eventually calling other functions to perform sub-tasks.
Example: The following code prints "Hello, World!":

```c
#include <stdio.h>

int main(int argc, char** argv) {
    printf("Hello, World! \n");
    return 0; // Deafult return value of the main function.
}
```

A.2 Declaring variables

Before being used, a variable must be declared by specifying a name and a type, although it is a good practice to initialize a value as well.
In addition, it is also possible to assign values in order to modify the contents of a variable.
Example: The following code declares a variable x, which is then set to store the value 5, and initializes another variable $y = 0$:

```c
#include <stdio.h>

int main() {
    int x; // Declaration of x as an integer.
    int y = 0; // Initialization of y = 0.
    x = 5; // Assignment of x = 5.
    return 0;
}
```

A.2.1 Defining macros

Using the #define command, it is possible to create macros, which act as "symbolic constants" whose value will always stay associated to the name of the variable.

A.3 Arithmetic operators

In order to perform computation involving variables, it is possible to use arithmetic operators, which include the standard mathematical operations, along with the modulo division, and self-increments.
Example: The following code declares two integer variables x and y that are used to declare new variables z, which is implemented as the sum between x and y, and v and w, which are implemented through self-increments:

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 2;
    int y = 3;
    int z = x + y; // Declare z = x + y.
    int v = x++; // Let v = 2, x = 3.
    int w = ++y; // Let y = 4, w = 4.
    return 0;
}
```

A.4 Relational operators

Generally speaking, relational operators are used to check conditions among variables, typically for comparison purposes.

A.5 Logical operators

Logical operators deal with combining more conditions, which, through dedicated commands, can be evaluated over entire words or bit-by-bit.

A.6 Input/Output activities

Input/Output activity mainly deals with showing values through the printf() function, which allows to print strings or, through dedicated placeholders, numerical values.
Alternatively, in the case of user-inputted values, it is possible to print values through the scanf() function.

A.7 Selective codes

A selective code branches towards different actions depending on whether a certain condition is true or not.

A.7.1 If/else structures

An if/else structure creates a hierarchical structure that gradually checks whether each condition is true or not, going on until the default case.
Example: The following code takes as input a positive integer x and, if x is even, it prints x^2, otherwise, if x is odd, it prints $x + 3$:

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x;
    printf("Enter an integer value for x: ");
    scanf("%d", &x);

    if (x % 2 == 0) {
        printf("x^2 = %d\n", x * x);
    } else {
        printf("x + 3 = %d\n", x + 3);
    }

    return 0;
}
```

A.7.2 Case structures

A case structure branches according to the value of a control variable, meaning that, to avoid issues, a default case must always be specified.
Example: The following code prints a certain string depending on the received input:

```c
#include <stdio.h>

int main() {
    int x;

    printf("Enter an integer value for x: ");
    scanf("%d", &x);

    switch (x) {
        case 0:
            printf("0\n");
            break;
        case 100:
            printf("1\n");
            break;
        default:
            printf("?\n");
            break;
    }

    return 0;
}

```

A.8 Iterative codes

An iterative code repeats some actions as long as a certain condition is true.

A.8.1 While loops

A while loop builds the iterative condition using an external control variable that gets updated during the iterations.
Example: The following while loop prints the sum of all integers from 0 to 10:

```c
#include <stdio.h>

int main() {
    int sum = 0;
    int i = 0;

    while (i <= 10) {
        sum += i;
        i++;
    }

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

A.8.2 Do/while loops

A do/while loop works similarly to a standard while loop, but, since it first executes the loop and then updates the control variable, it is always guaranteed to run at least once.
Example: The following do/while loop prints the sum of all integers from 0 to 10:

```c
#include <stdio.h>

int main() {
    int sum = 0;
    int i = 0;

    do {
        sum += i;
        i++;
    } while (i <= 10);

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

A.8.3 For loops

A for loop relies on a control variable that can be initialized and updated in a dedicated command.
Example: The following for loop prints the sum of all integers from 0 to 10:

```c
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 0; i <= 10; i++) {
        sum += i;
    }

    printf("Sum of integers from 0 to 10: %d\n", sum);
    return 0;
}
```

A.9 Arrays

An array is a vector of elements of the same type (which are accessed by index slicing) that is declared by specifying its size and type, although, generally speaking, it is a good practice to initialize its values as well.
Example: The following code initializes an array where $arr[i] = i$:

```c
#include <stdio.h>

#define L = 10;

int main() {
    int arr[L];

    int i = 0;

    for (i = 0; i < L; i++) {
        arr[i] = i;
    }

    return 0;
}
```

A.9.1 Generalizing arrays to build matrices

Arrays are often used to implement matrices as two-level arrays, whose elements are accessed using the $[i][j]$ notation, which allows to access the value located at the i^{th} row and j^{th} column.

A.9.2 Using arrays to represent strings

A string is a particular array of characters that is delimited using a dedicated terminator character.
Example: The following code provides some ways to implement a string as a character array:

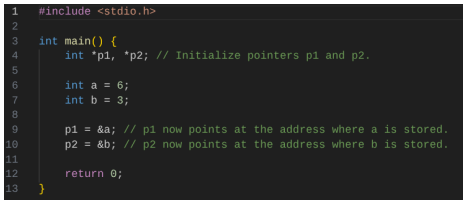
```c
#include <stdio.h>

int main() {
    char stringa[] = {"h", "e", "l", "l", "o"}; // First initialization.
    char stringa[] = "hello"; // Second initialization.

    char s[7] = "hello"; // Extra spaces will be kept empty.
}
```

A.10 Implementing functions

A function is a piece of code that can work independently of other functions and, if needed, it can also declare local variables within its scope.
Generally speaking, a function takes parameters by value, meaning that it will work on a copy of the original parameters, which are instead left untouched, although structures like arrays and pointers take parameters by reference, meaning that it will refer to the actual instance of the input values.

A.11 Pointers

A pointer is a particular variable that contains the memory address to another variable of the same type, eventually allowing to retrieve its value as well through dedicated commands.
Example: The following code, after initializing two pointers and two integers, sets p_1 to point towards a's memory address and p_2 to point towards b's memory address:



```c
#include <stdio.h>

int main() {
    int *p1, *p2; // Initialize pointers p1 and p2.

    int a = 6;
    int b = 3;

    p1 = &a; // p1 now points at the address where a is stored.
    p2 = &b; // p2 now points at the address where b is stored.

    return 0;
}
```

A.11.1 The void pointer

Commonly used for memory allocation, the void pointer is a special pointer that can be used to specify that a function can use any type of pointer, therefore allowing to easily convert pointer types.

A.12 Memory allocation

Memory in C consists of a static memory, which is used by the stack for executing a function, and a dynamic memory, which is used by the heap to handle dynamic variables whose size may not be known at runtime.
In particular, dynamic memory allocation is typically performed using the malloc() function, which reserves a memory buffer and provides a void pointer towards the address of the allocated space, although, in some cases, it might be more convenient to use functions such as realloc(), which tries to modify the size of the allocated buffer, or calloc(), which is used to reserve a memory buffer for a collection, such as an array.
On the other hand, it is possible to free up memory space by calling the free() function on a pointer towards an allocated buffer.

N.B.: Whenever there is not enough space to perform memory allocation, the allocation functions will return a NULL pointer to state that there was an error during the allocation.
Generally speaking, it is possible to see how much memory is needed to store a certain variable using the sizeof() function.
Example: The following code tries to allocate an integer in the memory space:

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

void func1(int *ptr);

int main() {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        printf("It is not possible to allocate space! \n");
        return 0;
    }

    *p = 30;
    printf("p = %d \n", *p); // Print 30.

    func1(p); // Modify the variable indicated by p.

    printf("p = %d \n", *p); // Print 13.

    return 0;
}

void func1(int *ptr) {
    *ptr = 13;
}
```

In particular, if the NULL pointer is returned, meaning that there is not enough space for the allocation, the code will return an error message.
Otherwise, it will change the value of the variable indicated by the pointer and call the func1() function to modify it.

A.13 Structured data

Declared using the "struct" keyword, a structure contains variables of different types that can be accessed using the .field command on a single instance.
Example: The following code defines a structure for articles that contains information about their code and price:

```c
#include <stdio.h>

struct Article {
    int articleID;
    float price;
}; // The semicolon is used because this is a declaration.

int main() {
    struct Article a1 = {467, 12.5}; // Direct initialization.

    struct Article a2; // Simple declaration.
    a2.articleID = 189; // Assignment by access.
    a2.price = 34.99;

    printf("a1 - Code: %d, Price: %0.2f \n", a1.articleID, a1.price);
    printf("a2 - Code: %d, Price: %0.2f \n", a2.articleID, a2.price);

    return 0;
}
```

A.13.1 Pointers to structured data

Similarly for variables, it is possible to assign pointers to a struct and give it a memory address, which can also be used to efficiently access its data.

A.13.2 Changing names using the typedef command

Generally speaking, it is possible to declare structured data more efficiently by exploiting an alias that is created using the "typedef" keyword.
Example: The following code defines a structure for books, which is then invoked at runtime using an alias:

```c
#include <stdio.h>

typedef struct Book {
    int pages;
    char title[50];
    float price;
} book; // This alias introduces a new type "book" representing Book structures.

int main() {
    book book1 = {354, "Fondamenti di Programmazione", 42.50};

    return 0;
}
```

A.13.3 Union structures

The "union" keyword allows to create a structure containing variables of different types that are all located at the same memory address, although, at any given time, only one of these variables can contain a value.
For this reason, a union structure always requires a memory space that is large enough to store a value for its biggest variable.
Example: The following code creates a union containing an integer and a character:

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

union MyData {
    int i;
    char c;
} mydata;

// An int requires 4 bytes of memory space while a char only requires 1 byte.
// MyData therefore requires 4 bytes to contain all its variables.

int main() {
    printf("Allocated memory: %d\n", sizeof(mydata));
    return 0;
}
```

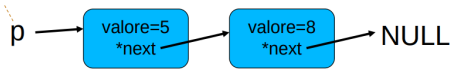
Since integers take up more memory space than characters, the union will need a memory space of the same size as an integer to store its possible value.

A.14 Linked lists

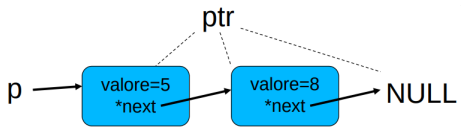
A limitation of arrays lies in the fact that they are static, meaning that each instance has a fixed size and stores its element contiguously in the memory.
On the other hand, a linked list is a dynamic collection, meaning that its size can change over time and, generally speaking, its elements are not stored contiguously in the memory.
For this reason, the elements of a list always feature a pointer towards the next element, with the exception of the last element, which instead uses a NULL pointer to indicate the end of the list.

Most particularly, list pointers come in handy to perform the following operations:

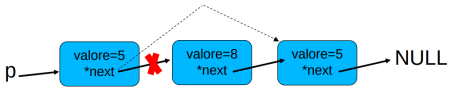
- **Inserting elements:** assuming for simplicity that the value is added at the end of the list, the last node has to point towards the new element, which instead will now feature the NULL pointer to represent the end of the list.



- **Printing elements:** a list can be printed by declaring a new pointer and moving it across the array to print its elements one by one.



- **Deleting elements:** if the first element has to be deleted, it is sufficient to set the first pointer towards the second element.
  From a more general point of view, deleting an element in the list requires moving up until the preceding element and updating its pointer to the element after the node to delete, which ends up having no references in the list.



N.B.: The pointer towards the first element of the list should never be modified as it allows to access the entire list.