Università degli Studi di Trieste

Dipartimento di Matematica e Informatica

Appunti di

Programmazione Avanzata e Parallela

Prof. Luca Manzoni

A.A. 2024-25

Autore

Ludovico Urbani

SM3201372

Aggiornato il

8 gennaio 2025

Corso di laurea in

Artificial Intelligence and Data Analytics

Indice

Parte I

C

Capitolo 1

Compilazione separata

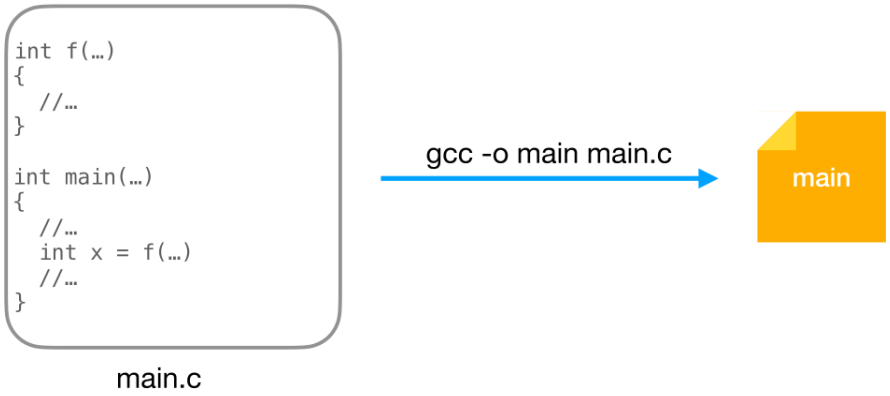


Figura 1.1: Compilazione semplice

In progetti più grandi, è comune dividere il codice in più file sorgente per migliorare la manutenibilità e l'organizzazione del progetto. Ad esempio, potresti avere un file che gestisce la logica principale (main.c), un altro che contiene funzioni di utilità, un altro per le variabili globali, ecc...

1.1 extern

Quando suddividi il codice in più file, il compilatore deve essere in grado di gestire questa separazione. Tuttavia, il compilatore analizza i file sorgente uno per uno, senza avere una visione globale di tutte le funzioni e variabili presenti nel progetto. Qui entra in gioco `extern`.

Definizione 1.1 (extern)
Il compilatore ha bisogno di sapere se una variabile o una funzione è definita da qualche parte, anche se non la trova immediatamente nel file corrente che sta compilando. La parola chiave `extern` dice al compilatore: "Questa variabile o funzione esiste, ma non è definita in questo file. La troverai altrove". Processo di compilazione in più fasi:

- Compilazione (fase per file): Ogni file sorgente (.c) viene compilato indipendentemente dagli altri. In questa fase, il compilatore si preoccupa solo di controllare la sintassi e verificare che tutte le variabili e funzioni siano almeno dichiarate. Se trova una dichiarazione con `extern`, sa che la definizione verrà trovata in un altro file e quindi non genera errori.
- Linking (fase finale): Dopo che ogni file è stato compilato separatamente, il linker prende tutti i file oggetto (.o) e li unisce. Il linker risolve le dichiarazioni `extern` trovando le effettive definizioni delle variabili o funzioni in altri file.

◊

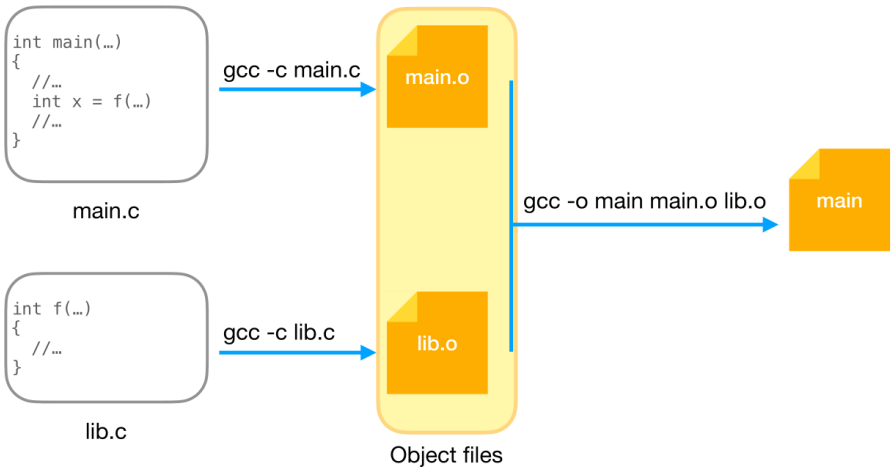


Figura 1.2: Compilazione separata con extern

Listing 1.1: main.c

```c
#include <stdio.h>

extern int var;

extern void print(int x);

void main() {
    print(var);
}
```

Listing 1.2: lib.c

```c
#include <stdio.h>

int var = 1;

void print(int x) {
    printf("%d\n", x);
}
```

Listing 1.3: bash

```bash
# Compilazione dei singoli file
gcc -c main.c
gcc -c lib.c

# Linking
gcc main.o lib.o -o program
```

`extern` permette di dichiarare la variabile globale e la funzione in un file e utilizzarle in altri file, mantenendo separata la definizione e facilitando la modularità del codice.
In fase di compilazione, il compilatore accetta di vedere solo la dichiarazione con `extern` e poi in fase di linking risolve la posizione della variabile e della funzione definiti in un altro file.

1.2 File header

La suddivisione del codice in file header (.h) e file sorgente (.c) è una pratica comune nella programmazione in C, soprattutto per organizzare e mantenere più facilmente progetti di grandi dimensioni. Questa separazione aiuta a migliorare la modularità, la riusabilità del codice e la gestione delle dipendenze tra i vari componenti di un progetto.

Definizione 1.2 (Struttura Header-Source)
- File Header (.h): contiene le dichiarazioni di funzioni, variabili, costanti, e tipi di dati che devono essere accessibili da più file sorgente. Gli header

non contengono codice eseguibile (come la definizione di funzioni), ma solo dichiarazioni e direttive che informano il compilatore su come utilizzare ciò che è definito in altri file. È nuona norma che gli header vengono inclusi nei file sorgente attraverso la direttiva `#include`.

- File Sorgente (.c): Contiene le definizioni effettive di funzioni e variabili. Questo è il file in cui il codice eseguibile è effettivamente scritto e dove viene implementata la logica del programma. I file .c possono includere i file header per accedere alle dichiarazioni e usarle nel proprio codice. ◊

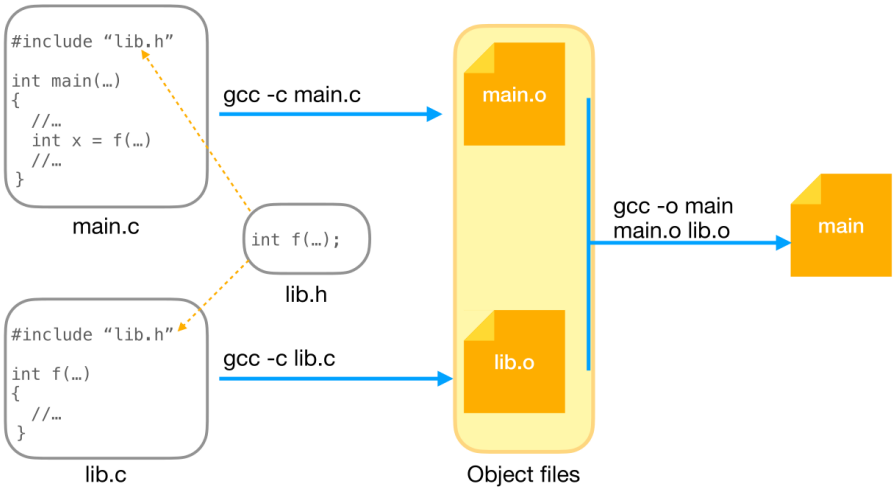


Figura 1.3: Compilazione separata con struttura Header/Source

Listing 1.4: main.c

```c
#include <stdio.h>
#include "lib.h"

void main() {
    printf("%d\n", f());
}
```

Listing 1.5: lib.h

```c
#include <stdio.h>

int f();
```

Listing 1.6: lib.c

```c
#include "lib.h"

int f() {
    return 1;
}
```

Vengono compilati solo i file sorgente, non gli headers.

1.2.1 Direttive del preprocessore

Definizione 1.3 (Direttive del preprocessore)
Le direttive del preprocessore in C sono comandi che vengono elaborati dal preprocessore prima che il vero processo di compilazione inizi. Queste direttive iniziano tutte con il simbolo # e forniscono istruzioni su come il codice sorgente dovrebbe essere modificato o manipolato dal preprocessore prima di essere passato al compilatore. ◊

Le direttive del preprocessore sono:

`#include`	Include il contenuto di un file esterno, come file header o librerie.
`#define`	Definisce macro o costanti simboliche che vengono sostituite testualmente nel codice.
`#undef`	Annulla una macro precedentemente definita con `#define`.
`#iflstinline`	Permette di includere codice in base a una condizione.
`#ifdef`	Include il codice solo se una macro è definita.
`#ifndef`	Include il codice solo se una macro non è definita.
`#else`	Specifica il blocco alternativo da includere se la condizione `#if` o `#ifdef` non è soddisfatta.
`#elif`	Alternativa a `#else` e `#if`, simile a "else if" nei linguaggi di programmazione.
`#endif`	Chiude un blocco condizionale iniziato con `#if`, `#ifdef` o `#ifndef`.
`#warning`	Emette un messaggio di avvertimento durante la compilazione, ma non interrompe il processo.
`#error`	Genera un errore durante la compilazione e interrompe il processo.

Quando un file header viene incluso più volte nello stesso progetto (direttamente o indirettamente) nello stesso file o in diversi files, il compilatore potrebbe tentare di processare le stesse dichiarazioni o definizioni più volte; Questo porta all'errore di compilazione *redefinition errors*.
Le direttive del preprocessore, in particolare quelle relative alle inclusioni condizionali, sono fondamentali per evitare la doppia inclusione dei file header nella struttura header-source di un programma C.

Listing 1.7: main.c

```c
#include <stdio.h>
#include "lib.h"

void main() {
    printf("%d\n", f());
}
```

Listing 1.8: lib.h

```c
#ifndef _LIB_H
#define _LIB_H

#include <stdio.h>

int f();

#endif
```

Listing 1.9: lib.c

```c
#include "lib.h"

int f() {
    return 1;
}
```

Capitolo 2

Opzioni di gcc

- `-On` con $n \in \{0, 1, 2, 3\}$:
  - `-O0` disattiva le ottimizzazioni
  - `-O3` è solitamente il massimo livello di ottimizzazione
- `-march=native` indica al compilatore di utilizzare tutte le estensioni e ottimizzazioni che hanno senso sulla macchina su cui si sta eseguendo la compilazione
- `-march=[nome]` indica al compilatore di utilizzare tutte le estensioni e ottimizzazioni che hanno senso sulla macchina indicata
- `-g` compila con i simboli di debug: rene più semplice l'uso dei debugger
- `-profile-generate` abilita la generazione di dati di profilazione durante l'esecuzione del programma compilato. Viene utilizzata in un processo chiamato Profiling-Guided Optimization (PGO) o Profile-Guided Optimization, che consiste in due fasi principali: la generazione dei dati di esecuzione e la ri-compilazione ottimizzata basata su questi dati.
- `-Wall` abilita tutti i warnings ragionevoli
- `-pedantic` abilita tutti i warnings relativi al non rispetto dello standard
- `-Wextra` abilita più warnings di `-Wall`
- `-std=` indica lo standard di C da utilizzare (es. c89, c99, c11, c17, c2x)
- `-fno-if-conversion` disabilita la conversione degli if in branchless
- `-fno-unroll-loops` disabilita lo svolgimento dei loops

Capitolo 3

Librerie

3.1 Librerie statiche

Definizione 3.1 (Librerie statiche)
Le librerie statiche in C sono collezioni di file oggetto che vengono collegate staticamente al programma durante la fase di linking. In questo contesto, "statico" significa che il codice delle librerie viene copiato direttamente nel file eseguibile durante il processo di compilazione. Una volta compilato, il programma non ha bisogno della libreria separata per poter essere eseguito, perché tutto il codice richiesto dalla libreria è stato incorporato nell'eseguibile. ◊

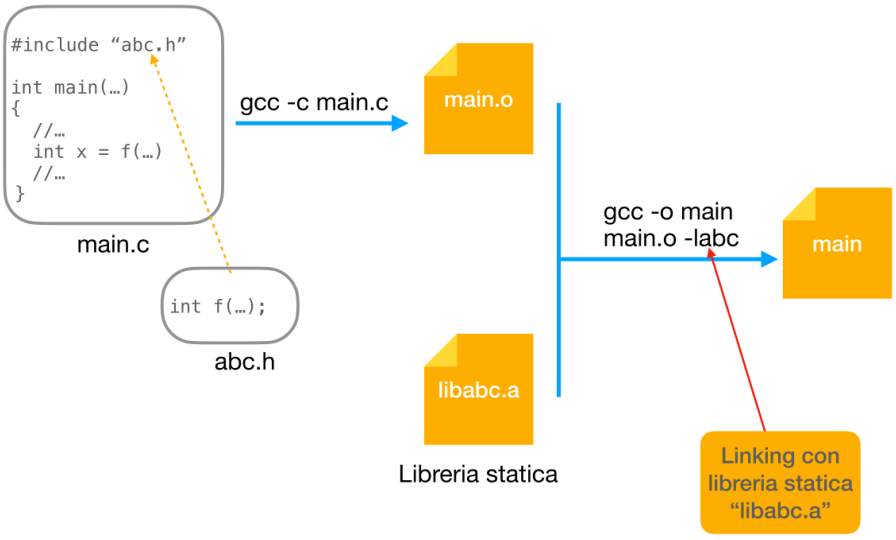


Figura 3.1: Compilazione con una libreria statica

Caratteristiche delle librerie statiche:

- Estensione: le librerie statiche di C hanno solitamente l'estensione .a (su Unix/Linux) o .lib (su Windows)
- Incorporazione nel binario: durante il linking, il codice necessario dalla libreria viene copiato direttamente nel file eseguibile. Questo aumenta la dimensione del file eseguibile, ma rende il programma indipendente dalla libreria durante l'esecuzione
- Prestazioni: poiché tutto il codice richiesto dalla libreria è incluso nel file eseguibile, non c'è bisogno di caricare dinamicamente alcun codice durante l'esecuzione, il che può migliorare le prestazioni in alcuni casi

- Indipendenza: una volta creato l'eseguibile, non c'è bisogno di distribuire separatamente la libreria, perché tutto il codice necessario è già incluso nel file binario
- Riuso del codice: le librerie statiche permettono di organizzare e riutilizzare il codice senza dover riscrivere le stesse funzioni in più file. La libreria può essere compilata una sola volta e riutilizzata da diversi programmi

Listing 3.1: libfile1.c

```c
#include <stdio.h>

void hello() {
    printf("Hello from libfile1!\n");
}
```

Listing 3.2: libfile2.c

```c
#include <stdio.h>

void hello() {
    printf("Hello from libfile2!\n");
}
```

Listing 3.3: Compilazione della libreria

```
gcc -c libfile1.c
gcc -c libfile2.c

# Creazione della libreria
ar r lib.a libfile1.o libfile2.o

# Creazione degli indici
ranlib lib.a
```

La funzione `ranlib` crea gli indici dei simboli della libreria, in pratica spiega cosa la libreria fornisce. Le funzioni `ar` e `ranlib` possono essere condensate nella funzione `ar rs lib.a libfile1.o libfile2.o`. Per poter usare la libreria statica, bisogna dire al compilatore dove trovare la libreria con l'opzione `-L` e di fare il linking con l'opzione `-l`:

Listing 3.4: Compilazione nella directory locale

```
gcc -L. -o main main.c -llib
```

3.2 Librerie dinamiche

Definizione 3.2 (Librerie dinamiche)
Le librerie dinamiche (o shared libraries) in C sono collezioni di file oggetto che vengono caricate dinamicamente dal sistema operativo durante l'esecuzione di un programma. A differenza delle librerie statiche, non vengono incorporate nell'eseguibile durante il linking, ma restano esterne e vengono utilizzate in fase di runtime. ◊

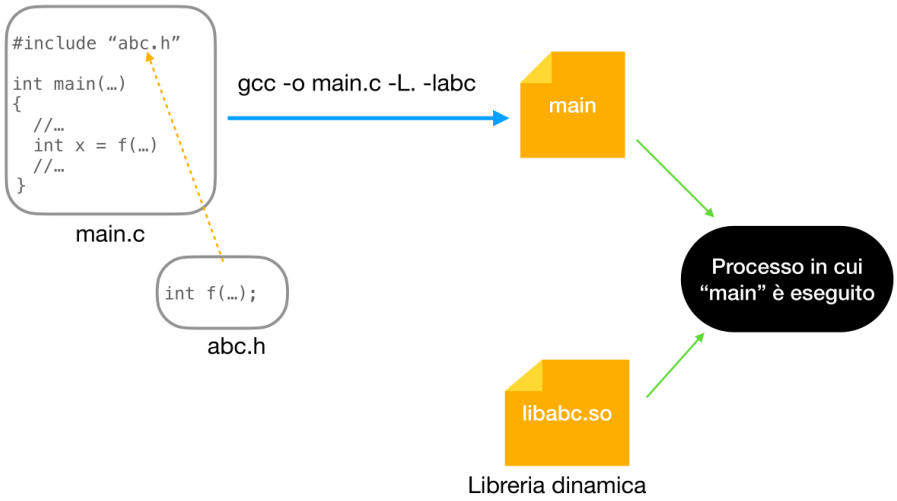


Figura 3.2: Compilazione con una libreria dinamica

Caratteristiche delle librerie dinamiche:

- Estensione: su sistemi Unix/Linux, le librerie dinamiche hanno l'estensione .so (shared object), mentre su Windows l'estensione è .dll (dynamic-link library)
- Separazione dal binario: durante il linking, il programma include solo i riferimenti alla libreria, ma il codice non viene copiato nell'eseguibile. Quando il programma viene eseguito, il sistema operativo carica la libreria dinamica in memoria
- Dimensioni dell'eseguibile ridotte: poiché il codice della libreria non è incluso nell'eseguibile, quest'ultimo rimane più piccolo rispetto a quello prodotto con le librerie statiche
- Condivisione della memoria: più programmi possono condividere la stessa libreria dinamica in memoria, riducendo l'uso complessivo della memoria. Questo è utile in ambienti multi-processo o multi-utente
- Flessibilità: le librerie dinamiche permettono di aggiornare e correggere bug senza dover ricompilare tutto il programma. È sufficiente aggiornare la libreria, e i programmi che la utilizzano beneficeranno automaticamente delle modifiche

Listing 3.5: file.c

```c
#include <stdio.h>

void hello() {
    printf("Hello from lib!\n");
}
```

Listing 3.6: Compilazione della libreria

```
gcc -fPIC -c file.c

# Creazione della libreria
gcc -o lib.so -shared file.o
```

L'opzione `-fPIC` serve per generare codice "position independent".
Per poter usare la libreria dinamica, bisogna dire al compilatore dove trovare la libreria con l'opzione `-L` e di fare il linking con l'opzione `-l`:

Listing 3.7: Compilazione nella directory locale

```
gcc -L. -o main main.c -llib
```

Per vedere le librerie dinamiche necessarie per far girare un programma:

Listing 3.8: ldd

```
# Linux
ldd main

# MacOS
otool -L main
```

3.2.1 Locazione nel filesysem

La posizione delle librerie dinamiche in un sistema operativo dipende dalle configurazioni del sistema stesso e dalle variabili d'ambiente. Ogni sistema operativo (Linux, macOS, Windows) ha il proprio metodo per gestire e individuare le librerie dinamiche, oltre a variabili d'ambiente specifiche che possono influenzare questa ricerca.

Linux Su Linux le librerie dinamiche hanno estensione .so (shared object) e si trovano comunemente nelle seguenti directory di sistema:

- `/lib` e `/usr/lib`: directory principali dove si trovano le librerie di sistema condivise
- `/usr/local/lib`: solitamente utilizzato per librerie installate dall'utente
- `/opt/lib`: utilizzata per applicazioni o librerie installate in modo opzionale

La variabile d'ambiente è `LD_LIBRARY_PATH`[1].

MacOS Su MacOS le librerie dinamiche hanno estensione .dylib o .bundle e si trovano tipicamente in:

- `/usr/lib`: directory di sistema dove si trovano le librerie condivise del sistema `/usr/local/lib`: solitamente utilizzata per librerie installate dall'utente o da software di terze parti `/opt/local/lib`: spesso utilizzata da gestori di pacchetti come MacPorts o Homebrew per installare librerie aggiuntive

La variabile d'ambiente è `DYLD_LIBRARY_PATH`

[1] Per leggere: `echo $LD_LIBRARY_PATH`
Per aggiungere path (es. `./lib`): `export LD_LIBRARY_PATH=./lib:$LD_LIBRARY_PATH`

Capitolo 4

Make

Se abbiamo decine o molte migliaia di file .c e .h sarebbe utile gestire bene la compilazione separata. Cioè se modifico il file "test.c" devo ricompilare "test.o", però non devo per forza ricompilare tutti gli altri files, ma solo quelli necessari.
Make è uno strumento sviluppato nel 1976 che ci permette di specificare: obiettivi, prerequisiti e comandi da eseguire.
Di default basta creare un file chiamato "Makefile" e invocando `make` posso fare tutti (e soli) i passi necessari a costruire la versione aggiornata del mio programma. L'obbiettivo viene ricostruito solo se i prerequisiti sono stati modificati, cioè la data di ultima modifica è stata cambiata. La verifica dei prerequisiti richiede un'applicazione ricorsiva:

- Se il prerequisito non esiste, cerca una regola per crearlo
- Se il prerequisito esiste, verifica se deve essere ricostruito e nel caso riapplica la procedura

4.1 Struttura

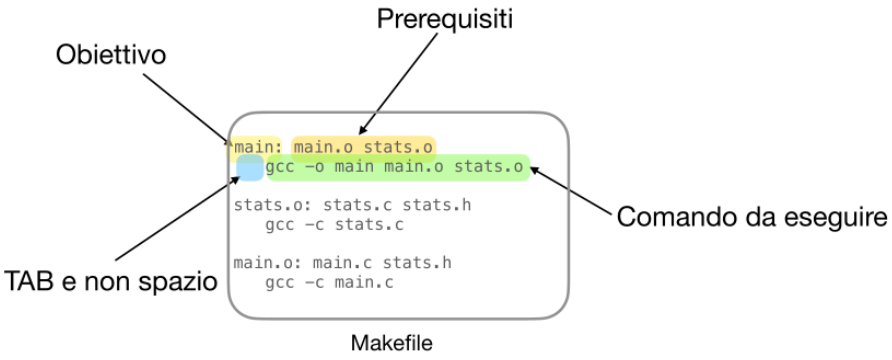


Figura 4.1: Makefile

È possibile eseguire un target a prescindere se sia stato modificato o no, basta aggiungere la riga di codice `.PHONY: target`
Le variabili sono indicate come `NOME = stringa` e possono essere usate per sostituzione `\$\{NOME\}`

Listing 4.1: Makefile

```
CFLAGS = -O3 -Wall

all: main.c
    gcc \${CFLAGS} main.c
```

le variabili possono essere modificate direttamente dalla chiamata a `CFLAGS=-O2`.
Vengono definite in automatico le variabili `\$@` il target, `\$^` la lista dei prerequisiti, `\$<` il primo prerequisito e `\$*` matching della parte template per poterla usare nei comandi da eseguire.

In molti casi ci sono regole simili che cambiano solo per il nome del file considerato. Per fare questo ci vengono incontro delle regole template in cui possiamo usare "%" per indicare qualsiasi sequenza di caratteri" nel target e poi usare "%" per indicare la stessa sequenza nei prerequisiti.

Listing 4.2: Pattern

```
%.o: %.c
    gcc -c \$<
```

Mentre il `\${wildcard *.c}` produrrà una lista di tutti i file .c presenti nella directory corrente.

Listing 4.3: Esecuzione condizionale

```
ifeq "stringa1" "stringa2"
    # code
else
    # code
endif
```

L'opzione `-p` stampa che comandi verranno eseguiti. Mentre l'opzione `@` davanti ad un comando, ne previene la stampa.

4.2 Esecuzione

- `make main` chiede la costruzione del target "main"
- `make -f nome_makefile main` specifica un makefile particolare
- `make` esegue il target "all"

Capitolo 5

Pipeline

Definizione 5.1 (Latency)
Definiamo la latenza come il tempo che intercorre da quando una operazione inizia e quando viene completata.
La latenza può essere misurata in termini di tempo (quanti nanosecondi) ma più normalmente in numero di cicli perché una istruzione completi. ◊

Definizione 5.2 (Throughput)
Definiamo il throughput come il numero di operazioni completate per unità di tempo.
Il throughput può essere misurato in termini di IPC (instructions per cycle), cioè quante istruzioni sono completate a ogni ciclo di clock. ◊

5.1 Fetch-decode-execute

Schema "base" di come viene eseguita una istruzione:

- Fetch: l'istruzione viene caricata dalla memoria
- Decode: l'istruzione viene decodificata
- Execute: l'istruzione viene eseguita, facendo magari uso della ALU o della FPU
- Memory: accesso alla memoria
- Write: scrittura dei risultati nei registri

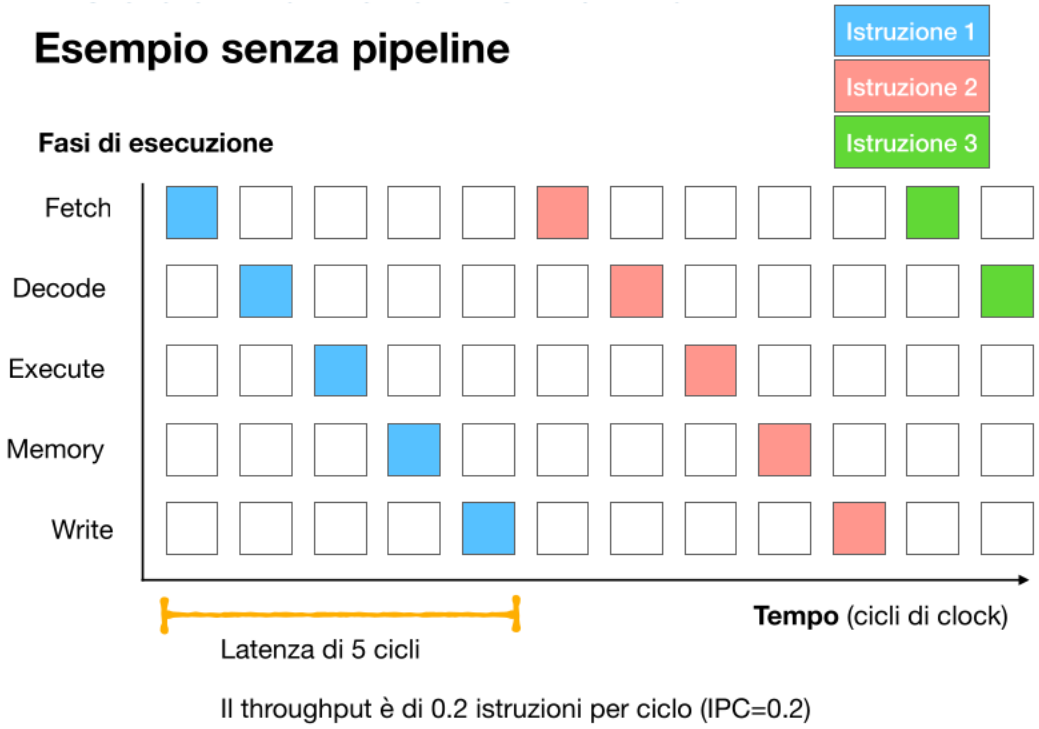


Figura 5.1: Fetch-decode-execute senza pipeline

5.2 Pipeline

Notiamo che le fasi di fetch-decode-execute possono essere compiute da diverse componenti della CPU. Se sappiamo quale è la prossima istruzione da eseguire allora possiamo iniziare l'esecuzione prima che quella precedente abbia completato.

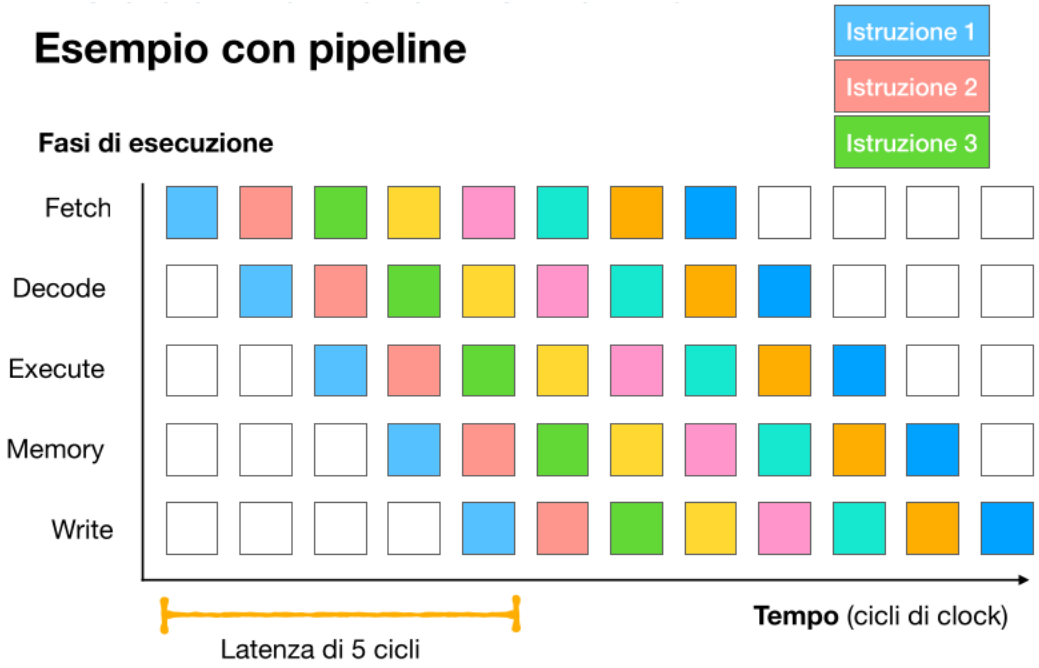


Figura 5.2: Fetch-decode-execute con pipeline

All'inizio le istruzioni iniziano a eseguire una alla volta fino a che tutte le diverse componenti non sono impegnate a eseguire qualcosa. Questo è il setup cost. Se abbiamo molte istruzioni possiamo ottenere un valore di IPC molto vicino a 1.
Se abbiamo due istruzioni consecutive che operano su dati differenti potenzialmente possono essere eseguite in parallelo. Richiede avere molte più componenti duplicate assieme a componenti in grado di decidere se due istruzioni possono essere eseguite in parallelo.
Per le CPU moderne il tempo di esecuzione di ogni istruzione può variare (e a volte, per la stessa istruzione, varia a seconda degli operandi). Una misura per questo è il Reverse Throughput (RThroughput), che con un valore < 1 significa che più istruzioni possono essere eseguite nello stesso momento.
Nella tabella successiva viene riportato per ogni operazione e tipo degli operandi, nelle ultime due colonne, la latenza e il RThroughput:

Arithmetic instructions				
ADD, SUB	r,r	1	1	0.25
ADD, SUB	r,i	1	1	0.25
ADD, SUB	r,m	1		0.33
ADD, SUB	m,r8/16	2	7-8	1
ADD, SUB	m,r32/64	2	1	1
ADC, SBB	r,r	1	1	1
ADC, SBB	r,i	1	1	1
ADC, SBB	r,m	1	1	1
ADC, SBB	m,r8/16	2	8	1
ADC, SBB	m,r32/64	2	1	1
ADCX ADOX	r,r	1	1	1
CMP	r,r	1	1	0.25
CMP	r,i	1	1	0.25
CMP	r,m	1		0.33
CMP	m,i	1		0.33
INC, DEC, NEG	r	1	1	0.25

Figura 5.3: RThroughput per Zen 4

5.2.1 Esecuzione out-of-order

Anche se possiamo eseguire no a n istruzioni in parallelo duplicando l'hardware non è detto che n istruzioni consecutive siano sempre eseguibili in parallelo. Possiamo però cambiare l'ordine delle istruzioni: le istruzioni possono iniziare a eseguire nel momento in cui hanno gli operandi sono disponibili. Dobbiamo però scrivere i risultati nell'ordine giusto.

- Fetch: l'istruzione viene caricata dalla memoria
- Decode: l'istruzione viene decodificata
- Dispatch: l'istruzione viene inserita in una coda di istruzioni da eseguire.
- Quando tutti gli operandi per eseguire l'istruzione sono presenti l'istruzione può iniziare l'esecuzione (anche prima di istruzioni precedenti)
- I risultati sono inseriti in coda
- Retire: quando tutte le istruzioni precedenti hanno scritto i risultati, questi ultimi vengono scritti

5.2.2 Pipeline hazards

Possiamo identificare tre casi in cui la pipeline deve aspettare:

1. Structural hazard: quando due o più istruzioni necessitano della stessa parte della CPU. Dobbiamo attendere che l'unità si liberi (solitamente 1-2 cicli), queste penalità dipendono dall'hardware a disposizione.

2. Data hazard: quando una istruzione necessita di un operando che deve essere ancora computato. Dobbiamo attendere che il risultato sia disponibile (la latenza del "critical path"), possiamo ristrutturare la computazione per limitare questa penalità.

3. Control hazard: quando non è possibile stabilire la successiva istruzione da eseguire. In questo caso non possiamo eseguire nessuna altra istruzione fino a quando non sappiamo la prossima istruzione da eseguire, solitamente perdendo 15-20 cicli.

In tutti questi casi vengono inserire delle "bolle" (bubbles) nella pipeline in cui non viene eseguito nulla.

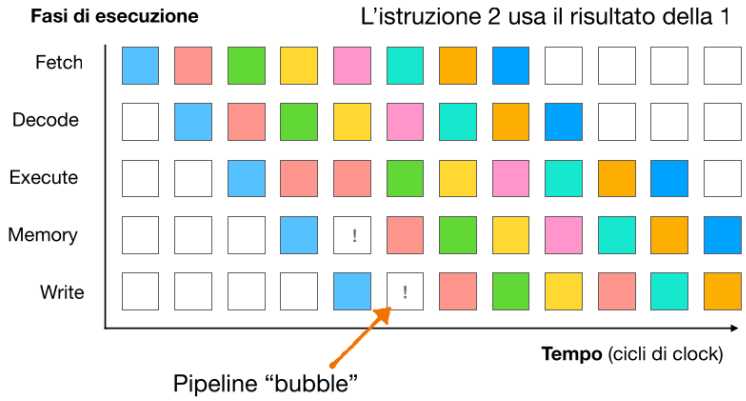


Figura 5.4: Bubbles

Dato che i control hazard derivati dai branch sono costosi è possibile pensare a una esecuzione speculativa: in un branch possiamo avere due (o più) target in cui la computazione prosegue. Questo ci permette di non pagare sempre la penalità per un control hazard.
Una volta che sappiamo dove l'esecuzione deve proseguire abbiamo due possibilità:

- La scelta era corretta. In questo caso possiamo effettivamente scrivere i risultati senza pagare la penalità.

- La scelta era sbagliata. Dobbiamo scartare i risultati e proseguire l'esecuzione dall'istruzione corretta. In questo caso non abbiamo risparmiato nulla

5.2.2.1 Beanch predictor

Il modo in cui facciamo la scelta di dove proseguire è dettato dal sistema di branch prediction che abbiamo.

- Branch predictor statico: non tiene statistiche, sceglie sempre preso o non preso.

- Branch predictor con contatore a saturazione: usando un contatore di n bit aggiungendo +1 se il branch è preso e -1 se non preso (no a un minimo e un massimo), si decide la predizione in base ad una soglia. Nel caso di 1 bit corrisponde a fare quello che è successo l'ultima volta che si è incontrato il branch.

- Branch predictor a due livelli: si utilizza lo storico di n bit (es. 010 per non-preso, preso, non-preso) per indicizzare in una tabella il predittore apposito per quello storico (es. uno con contatore a saturazione).

- Branch predictor locale: predittore separato per i diversi branch (es. in base all'indirizzo del branch), magari uno a due livelli.

- Branch predictor globale: predittore globale che usalo storico di tutti i branch (utile per vedere correlazioni tra i branch)
- Possibilità di combinare più branch predictors in un sistema ibrido che sceglie tra più predittori
- Predatori speci ci per cicli, ritorni da funzione e branch indiretti

In alcuni casi è possibile evitare i branch usando delle tecniche per programmazione branchless.

5.3 Ottimizzazioni

5.3.1 Prefetch

Possiamo iniziare a dire al sistema che alcuni indirizzi di memoria è probabile che vengano richiesti successivamente. Questo viene fatto tramite la funzione builtin del compilatore `__builtin_prefetch (void *)`. Questa direttiva informa il sistema di memoria che gli indirizzi passati come argomento potrebbero essere necessari a breve.

Listing 5.1: Prefetch

```
int array[100];

for (int i = 0; i < 100; i++) {
    array[i] = i;
}

int sum = 0;

for (int i = 0; i < 100; i++) {
    if (i + 1 < 100) {
        __builtin_prefetch(&array[i + 1]);
    }

    sum += array[i];
}
```

5.3.2 Loop unrolling

Per cicli il cui numero di iterazioni è piccolo (e noto a priori) è possibile rimpiazzare il branch per tornare all'inizio del ciclo ripetendo il corpo del ciclo tante volte quanto il numero di iterazioni. Alternativamente è comunque possibile srotolare parzialmente un ciclo, eseguento ad esempio 100 operazioni nello stesso ciclo.
Il comando `#pragma GCC unroll 16` srotola il ciclo 16 volte, cioè il ciclo viene raggruppato a step di 16.

5.3.3 Function inlining

Il "function inlining" consiste nel sostituire una chiamata a funzione col corpo della funzione.
Questo evita di dover fare una chiamata a funzione (con salvataggio di registri, modifica dello stack pointer, etc...), ma ha lo svantaggio che se ho più punti in cui la funzione è chiamata il codice è duplicato.
Il compilatore può decidere di fare inlining, ma possiamo dare degli hint con la keyword `inline` prima della definizione della funzione.

5.3.4 Likeliness di un branch

Se per qualche motivo sappiamo che un branch è più probabile che sia preso (o non preso) possiamo indicarlo al compilatore.
Gcc fornisce `__builtin_expect()`, che prende due argomenti:

- Il primo è una espressione da valutare (es. i < n)
- Il secondo è il valore atteso dell'espressione (es. 1 per vero e 0 per falso)

Listing 5.2: Prefetch

```
int count1 = 0;
int count2 = 0

for (int i = 0; i < 100; i++) {
    if (__builtin_expect(i % 5 == 0, 0)) {
        count1++;
    } else {
        count2++;
    }
}
```

Capitolo 6

Cache

L'accesso alla memoria rallenta di molto il codice. Si potrebbe tenere tutti i dati nei registri, ma questi sono limitati. Per questo bisogna aggiungere della memoria più piccola, ma più rapida in cui mantenere i dati più recenti: la cache.

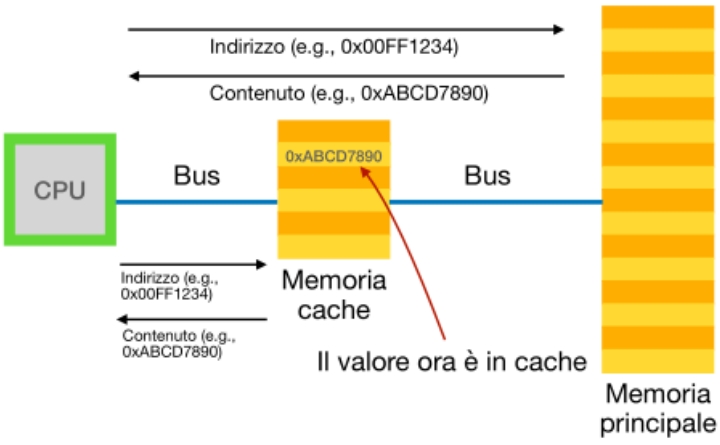


Figura 6.1: Cache

Ci aspettiamo che le cache funzionino se sono soddisfatti due principi:

- Località temporale: un indirizzo a cui abbiamo avuto accesso di recente verrà acceduto nuovamente a breve
- Località spaziale se abbiamo acceduto a un indirizzo x di recente ci aspettiamo di accedere anche a $x+1$, $x-1$, etc...

Anche leggere l'istruzione successiva richiede anche quello un accesso alla memoria. Per questo viene aggiunta anche un cache apposita per le istruzioni (es. loops).
Solitamente più è grande la cache e più è lenta, per questo solitamente vi sono più livelli di cache con diverse tradeoffs tra dimensioni e prestazioni:

- L1dati e L1istruzioni: due cache separate, una per i dati e una per le istruzioni (dimensione in KB)
- L2: non distingue tra dati e istruzioni, specifica per core (dimensione in MB)
- L3: più grande e più lenta, spesso condivisa tra più core

La cache non carica mai un solo byte alla volta, ma un'intera cache line (solitamente di dimensione 64bytes).

6.1 Associatività

Una cache deve poter associare un indirizzo a una specifica linea di cache.
Nel caso un indirizzo possa essere in un solo "slot" la cache è detta "direct-mapped" o "1-way associative". Se si leggono due indirizzi che condividono lo stesso "slot" la vecchia linea di cache viene rimossa e inserita quella nuova (rapida da implementare ma poco efficiente in termini di "hit rate").
Opposto al caso "direct mapped" vi è quello "fully associative". In questo caso ogni linea di cache può essere inserita in ogni "slot", questo aumenta l'"hit rate" ma rende la cache più difficile da costruire e rendere veloce. Inoltre serve selezionare una politica di rimpiazzo (replacement policy), cioè quale rimuovere se tutte le posizioni sono occupate (una politica comune è least recently used LRU).
Un compromesso è una cache k-way associative, in cui ogni indirizzo può essere associato a k slot.

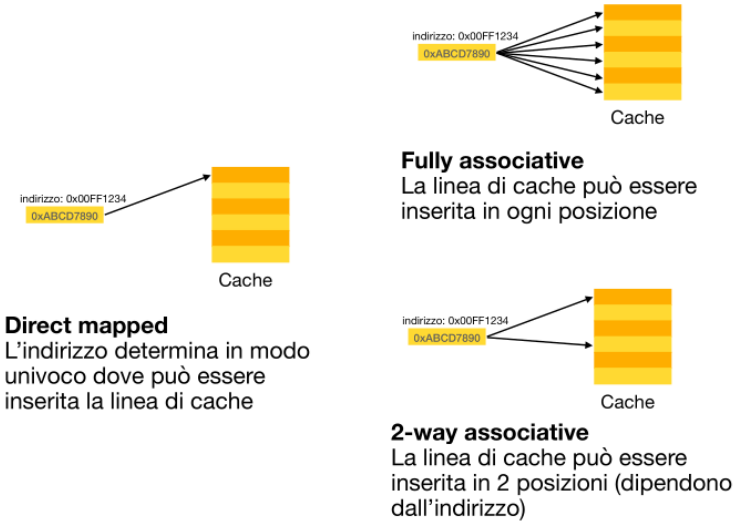


Figura 6.2: Associatività

6.2 Ottimizzazioni

6.2.1 Liste concatenate

ù

Listing 6.1: Lista concatenata

```
struct node {
    int key;
    struct node * next
}

int sum = 0;
struct node * current = head;
while (current != NULL)
{
    sum += current->key;
    current = current->next;
}
```

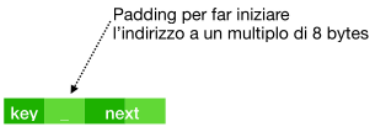


Figura 6.3: Linked list

Con una struttura dati a linked list classica otterrò una cache line in cui il prossimo nodo non è detto sia nella stessa cache line.
Invece con una lista concatenata "srotolata" avrò più valori nello stesso nodo, che finiranno nella stessa cache line.

Listing 6.2: Lista concatenata

```
struct node {
    int key[N];
    bool valid[N];
    struct node * next;
}

int sum = 0;
struct node * current = head;
while (current != NULL)
{
    for (int i = 0; i < N; I++) {
        if (current.valid[i])
        sum += current.key[i];
    }
    current = current.next;
}
```

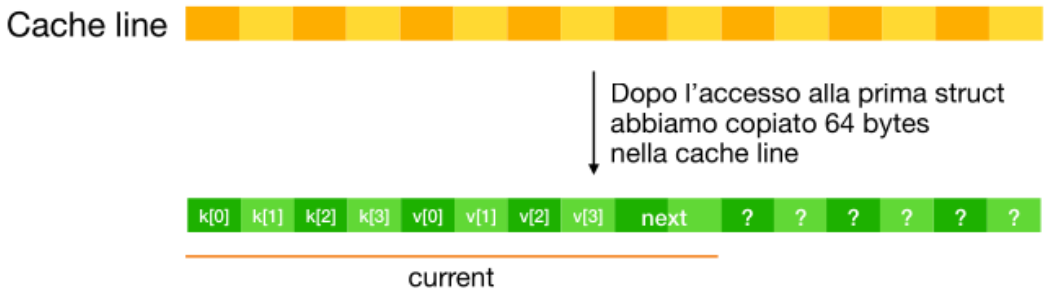


Figura 6.4: Unrolled linked list

6.2.2 Array di strutture

Listing 6.3: Array di struct

```
struct foo {
    int a;
    int b;
    int c;
    int d;
}

int sum = 0;
for (int i = 0; i < N; i++)
{
    sum += v[i].a;
}
```

La cache sarà costretta a caricare l'intero struct per leggere un solo attributo.

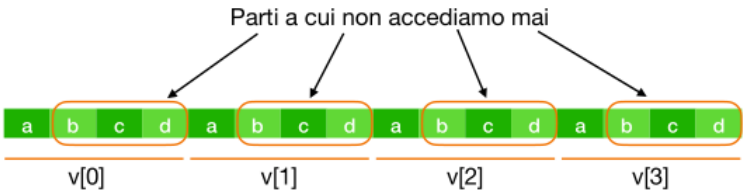


Figura 6.5: Array of struct

Invece con uno struct di array, la cache caricherà solo i dati utilizzati.
Se iteriamo spesso su tutta la collezione accedendo a un singolo membro della struttura allora struct-of-array è più adatto[1], se invece accediamo a più membri di una sola struttura alla volta allora array-of-struct è meglio, perché tutti i membri saranno spazialmente vicini in memoria.

[1] Un altro vantaggio di SoA è che iterando su un vettore di int/float (e non strutture) è più facile generare istruzioni vettoriali che operano su più elementi alla volta

Capitolo 7

Allineamento

In molte architetture una struttura può richiedere più bytes della somma dei byte occupati dai suoi membri. Ad esempio possono avere requisiti di allineamento per l'accesso alla memoria: `LOAD` può accedere solo a indirizzi multipli di 4 o 8.
Allora il compilatore allineerà i membri delle strutture in modo che sia possibile accederci in modo allineato. Per standard il compilatore deve rispettare l'ordine dei membri delle strutture, quindi `struct foo {int a; int * b; int c};` è diverso da `struct bar {int * b; int c; int a};`

7.1 Padding

Durante un'allocazione il compilatore deve aggiungere padding a fine struttura affinché in un array ogni struttura inizi a un indirizzo allineato.
Ad esempio il codice:

Listing 7.1: Padding

```
struct foo {
    int8_t a;
    int64_t b;
    int8_t c;
    int64_t d;
    int16_t e;
}
```

produrrà una struttura con il seguente padding:

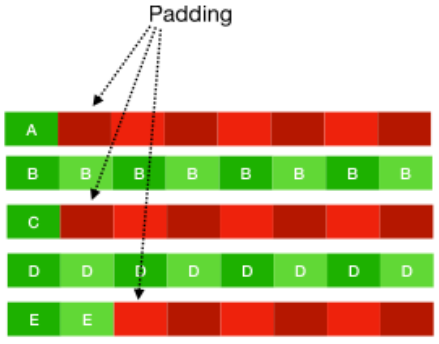


Figura 7.1: Padding

dove lo spazio utile è $1+8+1+8+2=20$ bytes, mentre lo spazio effettivamente utilizzato è 40 bytes. Questo avrà effetto sulla cache, infati in una cache line sarà possibile salvare solo due strutture.

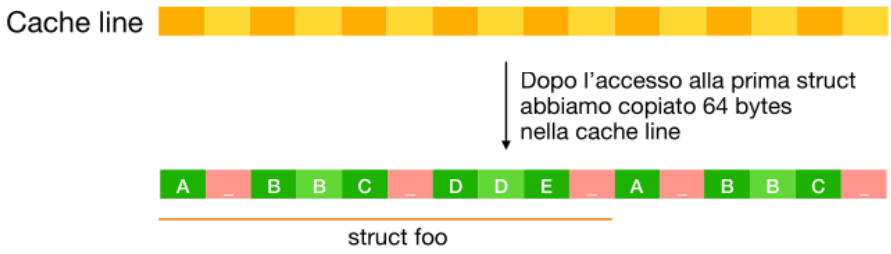


Figura 7.2: Cache with padding

Riordinando i membri della struttura:

Listing 7.2: Padding

```
struct foo {
    int8_t a;
    int8_t c;
    int16_t e;
    int64_t b;
    int64_t d;
}
```

produrrà una struttura con il seguente padding:

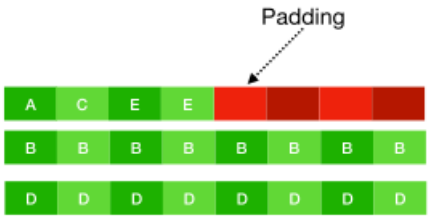


Figura 7.3: Padding

dove lo spazio utile è $1+8+1+8+2 = 20$ bytes, mentre lo spazio effettivamente utilizzato è 24 bytes, quindi solo 4 bytes di padding invece di 20. Ora la cache line potrà contenere oltre due strutture.
È possibile eliminare del tutto il padding usando l'attributo `__attribute__ ((packed))`, però questo ridurrà l'efficienza di accesso ai membri.

Listing 7.3: Padding

```
struct __attribute__ ((packed)) foo {
    int8_t a;
    int64_t b;
    int8_t c;
    int64_t d;
    int16_t e;
}
```

ora lo spazio occupato sarà esattamente 20 bytes:

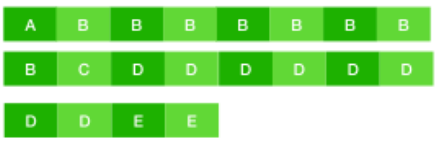


Figura 7.4: Padding

Capitolo 8

Accesso ai files

Quando vogliamo leggere un file il suo contenuto non viene direttamente caricato tutto in memoria, ma chiediamo al sistema operativo di fornirci una struttura che useremo per l'accesso. Questo ci permette di accedere al contenuto del file come fosse un flusso (o stream) di caratteri/byte.
Le operazioni di lettura e scrittura si riferiscono alla posizione corrente nel file.

Apertura Per aprire il file usiamo `fopen` della libreria <stdio.h> e per chiuderlo `fclode`:

Listing 8.1: Apertura

```
1 FILE * fopen(char * path, char * mode);
2 fclose(FILE * fp);
```

dove l'argomento mode stabilisce la modalità di accesso al file:

- `"r"`: lettura
- `"w"`: scrittura
- `"a"`: append (aggiunge in coda al file)
- `"r+"` o
- `"w+"`: lettura e scrittura
- `"rb"`,
- `"rb"`,
- `"ab"`,
- `"r+b"` o
- `"w+b"`: modalità binarie

8.1 Lettura e scrittura

Le funzioni usuali di lettura e scrittura (`printf` e `scanf`) hanno una versione che prende come argomento aggiuntivo che indica da dove leggere e scrivere:

Listing 8.2: Lettura e scrittura

```
1 fprintf(FILE * fp, char * format, ...);
2 int fscanf(FILE * fp, char * format);
```

`fprintf` funzione come printf e, infatti, possiamo passare stdout come primo parametro a fprintf per ottenere lo stesso comportamento.
`fscanf` funziona come scanf, ritorna il numero di elementi letti oppure EOF se è terminato il contenuto del file.
Più a basso livello abbiamo `getc()` e `putc()` che lavorano a livello di un singolo carattere. `getc` ha come argomento il file stream da cui leggere e ritorna un intero che è un cast a intero di un `unsigned char` oppure EOF. `putc` ha come argomenti un intero (il carattere da scrivere) e il file stream su cui scrivere, ritorna EOF in caso di errore di scrittura.
Per lavorare su blocchi di dati possiamo usare `fread` e `fwrite` in cui passiamo un vettore e quanti caratteri scrivere (del vettore) o leggere (salvandoli nel vettore):

Listing 8.3: Lettura e scrittura

```
1 int fread(void * ptr, size_t size, size_t nmemb, FILE * fp);
2 int fwrite(void * ptr, size_t size, size_t nmemb, FILE * fp);
```

dove il ritorno è il numero di elementi letti/scritti, `ptr` è il puntatore su cui leggere/scrivere i dati, `size` la dimensione del singolo elemento e `nmemb` il numero di elementi.

Effetti sul OS Solitamente il sistema operativo non scrive immediatamente sulla memoria di massa, ma tiene in un buffer quello che c'è da scrivere, questo viene svuotato (scritto su disco) quando si riempie oppure quando è passato abbastanza tempo.
Per le letture il sistema solitamente non legge un byte alla volta ma un blocco alla volta, con accessi successivi allo stesso blocco che accedono direttamente alla memoria principale (questo caching è gestito dal sistema operativo).
In Linux (e in diversi altri sistemi) questa componente del sistema operativo si chiama buffer cache.

Muoversi nel file La funzione `ftell` ci fornisce la posizione corrente all'interno del file, mentre `fseek` permette di riposizionarci:

Listing 8.4: fseek

```
1 int fseek(FILE * fd, long offset, int whence);
```

dove `offset` indica di quanto spostarsi in bytes e `whence` è una costante che indica rispetto a cosa sia l'offset: `SEEK_SET` l'inizio del file, `SEEK_CUR` la posizione corrente e `SEEK_END` la fine del file. La funzione ritorna 0 se lo spostamento è avvenuto correttamente.

8.2 MMAP

Sarebbe comodo accedere al contenuto dei file direttamente come se fossero presenti in memoria, quindi avere un accesso simile all'accesso a un vettore, caricando in memoria solo le parti del file a cui si accede a salvare ogni modifica sul file. Per far questo si utilizza la presenza di una memoria virtuale tramite MMAP.
Lo spazio degli indirizzi è solitamente molto più grande della memoria fisica (es. almeno 48 bit di indirizzi su alcune architetture moderne). Quindi possiamo fare finta che l'intero contenuto del file sia disponibile in memoria ad alcuni indirizzi virtuali, ma caricando il contenuto solo quando serve. Questo può essere fatto in modo trasparente dalla gestione della memoria del sistema operativo.
La funzione che ci permette di mappare il contenuto di un file in memoria è data dallo standard POSIX: mmap (disponibile nella libreria <sys/mman.h>).

Listing 8.5: mmap

```
1 void * mmap(void * addr, size_t len, int prot, int flags, int fd, off_t offset);
```

dove:

- `addr` è l'indirizzo in cui mappare. `(void *)0` lascia la scelta al sistema operativo
- `len` è la dimensione del mapping. Approssimata per essere un multiplo della dimensione delle pagine
- `prot` è il permesso di accesso alla memoria: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` (possono essere concatenati con OR)
- `flags` è la proprietà della memoria mappata: `MAP_PRIVATE` (le modifiche non sono visibili dall'esterno), `MAP_SHARED` (le modifiche sono visibili e applicate al file)
- `fs` è il file descriptor
- `offset` indica da che punto del file iniziare la mappatura (deve essere multiplo della dimensione delle pagine)

la funzione ritorna un puntatore alla memoria in cui è stato mappato il file.
La chiusura di un le non rimuove i mapping che lo coinvolgono, quindi anche la chiusura è in più passi: chiamare `munmap(void * addr, size_t size)` con l'indirizzo restituito da `mmap` e la dimensione passata quando si è chiamato `mmap`, successivamente chiudere il file con `fclose`.
Quando viene e ettuato `munmap` le modifiche saranno scritte non necessariamente immediatamente, ma è possibile forzare la scrittura del file con `msync`.
È ottenerne la dimensione del file chiamando `stat(char *, struct stat *)` con argomenti il path e un puntatore a una struttura di tipo `struct stat` che conterrà una serie di informazioni sul file, tra cui anche la dimensione nel campo `st_size`.

Listing 8.6: mmap

```
FILE * fd = fopen("file", "r");

int descriptor = fileno(fd);

stat("mmap_demo", &sbuf);

data = mmap((void *)0, sbuf.st_size, PROT_READ, MAP_SHARED, fdnum, 0);

munmap(data, sbuf.st_size);
fclose(fd);
```

Per cambiare la dimensione di un le (es. crediamo un file nuovo e vogliamo incrementarne la dimensione a 10MB) possiamo usare `ftruncate(int fd, off_t length)`, della libreria <unistd.h>, per cambiare la dimensione del file alla lunghezza indicata.
`madvise` può essere utilizzato per indicare come intendiamo accedere a un range di indirizzi (es. sequenziale o casuale) o se certi indirizzi non saranno più usati (o necessiteremo di usarli a breve).

Vantaggi Possiamo accedere al file come se fosse della normale memoria (niente necessità di avere un punto preciso dove fare letture e scritture).
Invece di utilizzare "read", "write" e "lseek" come chiamate possiamo a darci al sistema di gestione della memoria (potenzialmente più efficiente).
Facilita l'accesso quando si hanno più processi o thread che devono lavorare sullo stesso file (attenzione che servono dei lock).

Svantaggi Per alcune tipologie di accesso (es. sequenziale) potrebbe non essere necessariamente più efficiente e mappare i file in memoria è più macchinoso.
`mmap` esiste per i sistemi POSIX, non per ogni sistema operativo.
Non possiamo accedere a file che abbiano dimensioni superiori allo spazio degli indirizzi (che non è la memoria fisica).

Capitolo 9

Operazioni vettoriali

Supponiamo di voler sommare due vettori u e v, questo comporterebbe una serie di operazioni della forma $u_1 + v_1$, $u_2 + v_2$, ... Sarebbe comodo poter esprimere il concetto "esegui la stessa operazione su tutte le coppie di valori". Un metodo comune è quello di avere istruzioni che lavorano su k valori alla volta (es. da 2 a 16 valori).

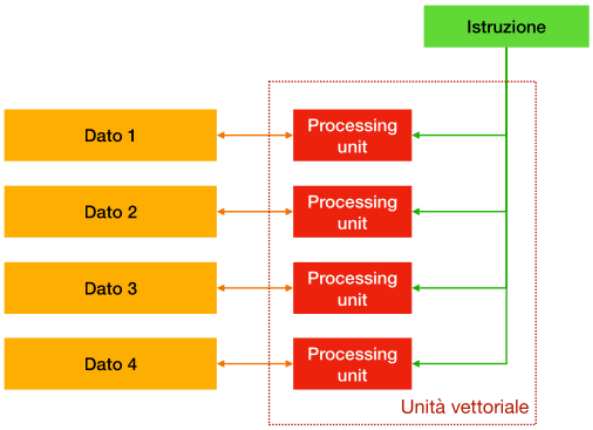


Figura 9.1: Struttura di una unità vettoriale

9.1 Tassonomia di Flynn (1972)

SISD (Single Instruction Stream, Single Data Stream) È il modello che pensiamo per il "normale" calcolatore con un singolo processore. Una singola sequenza di istruzioni che opera su un singolo flusso di dati (es. una operazione alla volta).

MISD (Multiple Instruction Streams, Single Data Stream) Architetture poco presenti, usate per ridondanza (es. più processori che eseguono la stessa istruzione sugli stessi dati e si verifica che tutti i risultati coincidano). Le architetture systolic array sono classificate come MISD ma, per la loro struttura particolare, la classificazione non è senza problemi.

MIMD (Multiple Instruction Streams, Multiple Data Streams) Più unità di esecuzione operano su più flussi di dati. Ne fanno parte i sistemi con più core, i sistemi distribuiti, etc...

SIMD (Multiple Instruction Streams, Single Data Stream) Più flussi di dati su cui viene applicata la stessa operazione. Nell'articolo del 1972 Flynn opera una suddivisione aggiuntiva:

- Array Processors: ogni unità riceve la stessa istruzioni ma opera su memoria separata.

- Pipelined Processors: ogni unità riceve la stessa istruzione ma la memoria è comune (e le istruzioni operano su parti di essa).
- Associative Processors: ogni unità riceve la stessa istruzione locale ma compie una scelta locale se eseguirla o no. Nella terminologia moderna sono "predicated instructions".

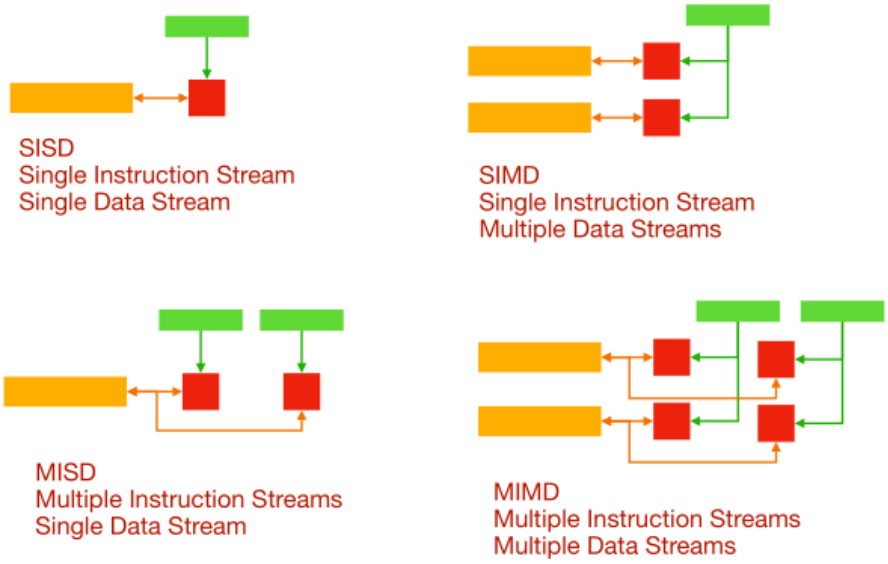


Figura 9.2: Tassonomia di Flynn

9.2 Registri vettoriali

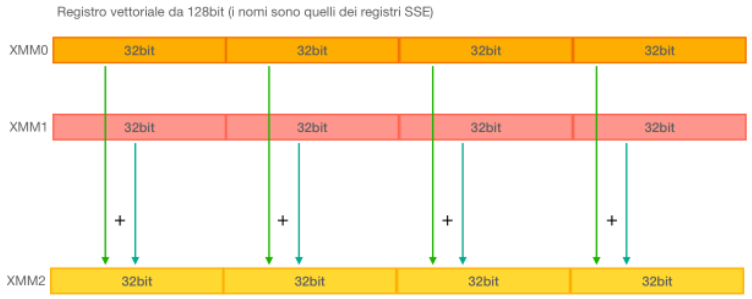


Figura 9.3: Registri vettoriali

Ogni estensione definisce un insieme di registri (xmm0-15, ymm0-15, zmm0-15 per SSE, AVX e AVX512) e operazioni che possono lavorare su di essi. Generalmente queste operazioni assumo che ogni registro abbia un certo numero di valori dello stesso tipo (es. 4 foat, 8 interi di 16 bit, etc...).
Per usarli possiamo:

1. Ignorare la loro esistenza (a volte le prestazioni sono migliori).
2. Affidarci al compilatore, in quanto gli ottimizzatori di molti compilatori forniscono dei sistemi di auto-vettorizzazione del codice e coprono diversi casi comuni (es. somma degli elementi di due vettori). Il problema però è che non sono perfetti, basta cambiare poco per perdere la vettorizzazione.

3. Usare delle intrinsics del compilatore che espongono alcune istruzione vettoriali come funzioni. Questo però limita la portabilità del codice su diverse architetture.

4. Usare delle estensioni vettoriali dei compilatore / librerie. Alcuni compilatori come GCC espongono un tipo vettore generico utilizzabile per delle operazioni vettoriali. Alcune librerie espongono funzioni che poi sono trasformate nelle istruzioni vettoriali specifiche.

5. Usare dei linguaggi con estensioni apposite per SIMD.

Intrinsics AVX Se vogliamo usare le intrinsics (nel nostro caso per AVX) dovremmo importare la libreria <immintrin.h>, utilizzare tipi di dato e funzioni apposite (es. `__m256` come 8 float di 32 bit o `__mm256d` per 4 double di 64 bit) e compilare con l'opzione `-mavx`.

Listing 9.1: Intrinsics

```
// Somma di 8 floats
__m256 _mm256_add_ps(__mm256 x, __mm256 y)

// Load di 8 floats all'indirizzo puntato
__mm256 _mm256_loadu_ps(float * b)

// Store di 8 floats all'indirizzo puntato
void _mm256_storeu_ps(float * v, __mm256 x);
```

Estensioni vettoriali per GCC GCC ci fornisce anche delle estensioni vettoriali generiche ()il codice sarà comparabile su più architetture). Possiamo definire dei tipi con la sintassi:

Listing 9.2: Estensioni GCC

```
typedef int v4si __attribute__ ((vector_size (4 * sizeof(int))));
typedef int v8si __attribute__ ((vector_size (8 * sizeof(int))));
typedef int v8f __attribute__ ((vector_size (8 * sizeof(float))));
```

In pratica diamo un nome a un vettore di n elementi dello stesso tipo, sarà compito del compilatore capire che istruzioni vettoriali usare.

Capitolo 10

Multithreading con OpenMP

OpenMP è un'API per il supporto della programmazione parallela in sistemi multi-processore e memoria condivisa. Questo tipo di parallelismo usa i thread, cioè più processori che eseguono il codice in parallelo accedendo però alla stessa memoria.
Un thread principale fa partire più thread secondari e il sistema distribuisce il lavoro tra i thread. Sono offerti una serie di costrutti utili alla parallelizzazione: eseguire del codice in parallelo, sincronizzare più thread, parallelizzare i cicli for (in modo quasi immediato).
Serve importare il <omp.h> che mette a disposizione una serie di funzioni per ottenere, per esempio, il numero di thread corrente, il numero di thread totali, etc... È necessario compilare con l'opzione `-fopenmp`. All'interno del codice la maggior parte del lavoro è svolto da della pragma[1].

10.1 Parallel

`#pragma omp parallel` è la prima pragma che è quella che indica che del codice debba essere eseguito in parallelo. Indica che il blocco di codice successivo (che deve essere racchiuso da graffe se è più di una singola istruzione) deve venire eseguito in parallelo.
Tutte le variabili definite fuori dal blocco sono comuni a tutti i thread (con rischio di race conditions se l'accesso non è regolato), tutte le variabili definite all'interno del blocco sono locali al singolo thread.

Listing 10.1: Pragma omp parallel

```
a();
#pragma omp parallel
{
    b();
    c();
}
d();
```

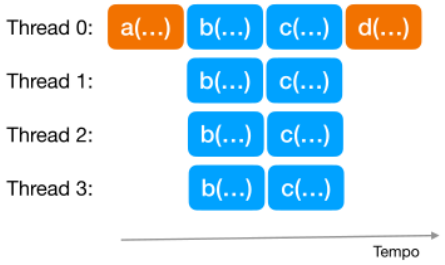


Figura 10.1: Esecuzione sui threads

[1] The `#pragma` directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.

La libreria <omp.h> mette a disposizione le funzioni:

- `omp_get_max_threads()` per ottenere il numero massimo di threads usati nelle sezioni parallele
- `omp_get_num_threads()` per ottenere il numero di thread che OpenMP sta usando in questa sezione parallel
- `omp_get_thread_num()` per ottenere l'identificativo del thread corrente.

È possibile specificare il numero di threads che devono eseguire una sezione parallela con `#pragma omp parallel num_threads(3)`.
Potenzialmente è possibile eseguire anche un solo thread con `#pragma omp single`

10.1.1 Sincronizzazione

Supponiamo di voler scrivere in una variabile comune a tutti i thread, per esempio sommandoci il valore di una loro variabile locale. Se sommiamo dentro la sezione parallela rischiamo delle race conditio, ma fuori dalla sezione parallela le variabili locali non esistono. Allora possiamo dire che una parte di una sezione parallela è una sezione critica.

Listing 10.2: Pragma omp parallel

```
#pragma omp parallel
{
    a()
    #pragma omp critical
    {
        b();
    }
    c();
}
```

Inserire una sezione critica equivale ad aggiungere un lock (specifico per quella sezione): il lock viene preso all'ingresso della sezione critica e rilasciato all'uscita. Chiaramente le sezioni critiche serializzano l'esecuzione, quindi ha senso usarle solo quando serve.

10.2 Parallel for

Un costrutto comune da parallelizzare è il ciclo `for`: ogni iterazione del ciclo opera su variabili differenti, solitamente su indici diversi degli array. Il metodo standard di parallelizzazione è quello di dare far fare parte del ciclo a ogni thread.

Listing 10.3: Pragma omp parallel for

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

Figura 10.2: Esecuzione sui threads

Questa pragma è la combinazione di `#pragma omp parallel` e `#pragma omp for`. Se necessario è possibile spezza il codice nelle due pragma.

nowait Per dire che non è necessario attendere che tutti i thread terminino è sufficiente aggiungere `nowait` alla pragma for:

Listing 10.4: Pragma omp parallel for nowait

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i = 0; i < 6; i++) {
        f(i);
    }
    g();
}
```

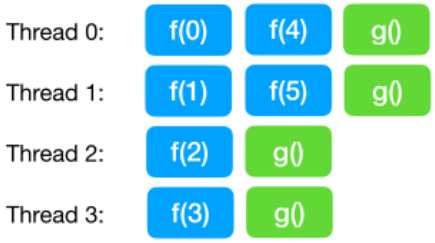


Figura 10.3: Esecuzione sui threads

10.2.1 Schedule

Possiamo decidere come sono spezzati tra i thread i cicli for.

Schedule statico Di default si assegna la prima iterazione al primo thread, la seconda al secondo, e via così ricominciando da capo (questo è lo scheduler statico con dimensione 1). Possiamo anche decidere di cambiare la dimensione a k, in quel caso daremo k iterazioni al primo thread, le k successive al secondo, etc...

Listing 10.5: Schedule

```
#pragma omp parallel
{
    #pragma omp for schedule(static, 2)
    for (int i = 0; i < 12; i++) {
        f(i);
    }
    g();
}
```

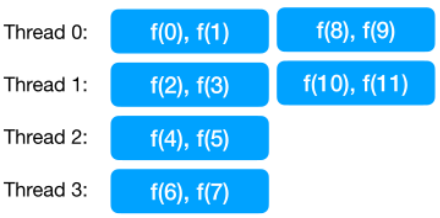


Figura 10.4: Esecuzione sui threads

Schedule dinamico Nel caso non tutte le iterazioni abbiamo la stessa lunghezza lo schedule statico potrebbe non essere ottimale. Possiamo usare uno schedule dinamico: dopo aver completato un blocco di iterazioni il thread verifica quale iterazione del ciclo for non sta venendo eseguita da nessuno e la prende.
Questa operazione di scheduling è costosa, quindi va fatta solo se c'è questo sbilanciamento, ed è utile aumentare la dimensione dei blocchi per trovare un trade-of tra quanto ribilanciare e quanto di frequente eseguire lo scheduling.

Listing 10.6: Schedule

```
#pragma omp parallel
{
    #pragma omp for schedule(dynamic, 1)
    for (int i = 0; i < 12; i++) {
        f(i);
    }
    g();
}
```

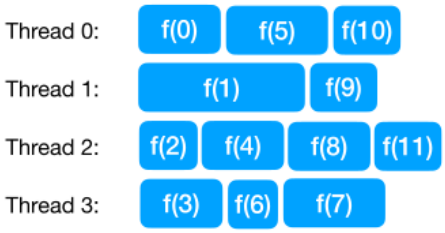


Figura 10.5: Esecuzione sui threads

10.2.2 Cicli innestati

L'operazione di collasso di più cicli innestati può essere fatta in automatico con l'opzione `collapse` di `#pragma omp for`:

Listing 10.7: Collapse

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 5; j++) {
        f(i,j);
    }
}
```

Questa opzione trasforma di due cicli innestati di lunghezza n e m in un solo ciclo di lunghezza $n \times m$.

10.3 Task

Supponiamo di avere funzioni (anche molto differenti) che possono eseguire in parallelo. Possiamo pensare di avere un thread principale che dispensa compiti ad un pool di threads pronti ad eseguirli.

Listing 10.8: Task

```
#pragma omp parallel
#pragma omp single
{
    f(2);

    #pragma omp task
    g(3);
```

```
8
9       #pragma omp task
10      f(12);
11
12      g(24);
13
14      f(3);
15
16      #pragma omp task
17      g(5);
18
19      g(8);
20  }
```

Figura 10.6: Esecuzione sui threads

Parte II

Python

Capitolo 11

Chiamare C da Python

È possibile chiamare una libreria condivisa C a partire da Python, serve però prestare attenzione al tipo degli argomenti, il tipo di ritorno delle funzioni e chi gestisce la memoria.
Il tipo "int" in Python non corrisponde al tipo "int" in C: n Python il tipo ha precisione arbitraria, quindi un numero di bit di rappresentazione non fissato, mentre in C ha un numero fissato di bit (32 o 64). I char in C sono pensati per rappresentare la codi ca ASCII, ma Python un carattere può essere un simbolo in Unicode e quindi occupare più di un byte).

11.1 Libreria ctypes

La libreria ctypes è inclusa nelle librerie standard di Python, permette di accedere a librerie condivise scritte in C, consente di effettuare conversione di tipo tra i tipi di C e i tipi di Python e permette di specificare il tipo degli argomenti delle funzioni C.

Tipo C	Tipo in ctypes	Tipo Python
int	c_int	int
unsigned int	c_uint	int
float	c_float	float
double	c_double	float
char	c_char	Oggetto di tipo “bytes” con un solo carattere
size_t	c_size_t	int

Figura 11.1: Conversione di tipo

Caricamento della libreria Ricordando che per è possibile compilare un file come libreria condivida con le opzioni `-fPIC` e `-shared` del compilatore: `gcc -fPIC -shared -o library.so library.c`. È possibile caricare la libreria in Python usando ctypes:

Listing 11.1: Libreria condivisa

```
1 ctypes.cdll.LoadLibrary("library.so")
```

Il valore ritornato è un oggetto che ci permette di accedere alla funzioni definite nella libreria. Ogni funzione C ha nell'oggetto creato da ctypes un attributo `argtypes` che permette di fornire una lista di tipi di argomenti e un attributo `restype` per specificare il tipo di ritorno.

Listing 11.2: lib.c

```c
float sum(int a, float b) {
    return a + b;
}
```

Listing 11.3: main.py

```python
lib = ctypes.cdll.LoadLibrary("lib.so")

lib.sum.argtypes = [
    ctypes.c_int, ctypes.c_float
]
lib.sum.restype = ctypes.c_float

lib.sum(1, 2.0)
```

Puntatori Sui può ottenere il puntatore a un oggetto con tipo fornito da ctypes (es. `c_int`) con `pointer()`:

Listing 11.4: Puntatore

```python
n = ctypes.c_int(4)
p = pointer(a)
```

Il tipo corrispondente (da usare se si speci cano gli argomenti delle funzioni) è `POINTER(c_int)`.

Capitolo 12

Programmazione ad oggetti

La programmazione orientata agli oggetti è un modo di strutturare i programmi aggregando in "oggetti" il comportamento e le proprietà/stati.

Definizione 12.1 (Classe)
In Python le classi sono un modo di definire strutture dati (e operazioni su di esse) da parte dell'utente.
◊

Tutte le classi sono usate per creare oggetti e tutti gli oggetti contengono delle caratteristiche/proprietà/attributi. Questi possono venire impostati al momento della creazione dell'oggetto: ll momento della creazione viene chiamato il metodo `__init__(self, ...)`. Questo metodo è utilizzato per inizializzare gli attributi di un oggett al momento della creazione e viene chiamato anche costruttore.
Il metodo `__init__` ha come primo argomento `self` che indica l'oggetto stesso (equivalente a `this` in Java), può essere interpretato anche come un puntatore C alla struttura stessa. Gli argomenti dopo `self` possono essere aggiunti per inizializzare lo stato dell'oggetto.
In alcuni casi ci sono caratteristiche comuni a tutte le istanze di una classe, in questo caso ha senso che siano definite a livello di classe e non di singola istanza. Queste variabili sono definite dentro la classe ma fuori dal metodo `__init__`. Saranno accessibili usando `NomeClasse.NomeAttributo`, ma anche come se fossero normali attributi di una istanza. Se li modifichiamo accedendo tramite una istanza verranno modificati solo per quella istanza, mentre se li modifichiamo accedendo tramite il nome della classe li modifichiamo per tutte le istanze.

12.1 Accesso pubblico o privato

Alcuni degli attributi e dei metodi potrebbero essere specifici dell'implementazione e vorremmo non renderli visibili all'esterno. Alcuni linguaggi di programmazione forzano il controllo degli accessi con i modificatori `public` (accessibile a tutti), `protected` (accessibile alla classe stessa e alle sottoclassi) e `private` (accessibile solo dalla classe stessa).
L'utilizzo di metodi e attributi privati ci permette di cambiare l'implementazione senza dover cambiare chi usa la classe perché non può accedere ai dettagli dell'implementazione, però Python non supporta i modificatori. Allora si utilizzano una serie di convenzioni per segnalare che alcuni attributi e metodi non sono pubblici:

- Metodi/attributi pubblici: nessun cambiamento
- Metodi/attributi protetti: il nome inizia con "`_`", ma possiamo comunque accedere senza problemi
- Metodi/attributi privati: il nome inizia con "`__`", in questo caso viene effettuato del name mangling (il noe con cui il metodo/attributo viene visto all'efetrno è `_nomeclasse__nomemetodo`)

12.2 Metodi di classe

Un metodo di classe è creato precedendo la definizione con `@classmethod` (il primo argomento per convezione si chiamerà `cls` e non `self`).

Listing 12.1: Metodo di classe

```
class MyClass:
    def __init__(self):
        self.var = None

    @classmethod
    def from_int(cls, num):
        myclass = cls()
        cls.var = num
        return cls
```

I metodi di classe possono essere chiamati prima che sia creato un oggetto, quindi possiamo usarli per fornire metodi alternativi di costruzione di oggetti (es. costruttore "standard" e costruttore che copia un'altra classe).

12.3 Dunder methods

Alcuni metodi hanno un significato speciale e vengono invocati con degli operatori. Ad esempio:

Metodo	Utilizzo	Argomenti
`__init__`	`Obj(...)`	`(self, ...)`
`__str__`	`str(obj)`	`(self)`
`__lt__`	`obj < ...`	`(self, other)`
`__le__`	`obj <= ...`	`(self, other)`
`__eq__`	`obj == ...`	`(self, other)`
`__getitem__`	Lettura di `obj[key]`	`(self, key)`
`__add__`	`obj + ...`	`(self, other)`
`__iadd__`	`obj += ...`	`(self, other)`
`__contains__`	`... in obj`	`(self, other)`

I metodi disponibili nello standard Python3 sono: `__abs__`, `__add__`, `__and__`, `__call__`, `__class__`, `__cmp__`, `__coerce__`, `__complex__`, `__contains__`, `__del__`, `__delattr__`, `__delete__`, `__delitem__`, `__delslice__`, `__dict__`, `__div__`, `__divmod__`, `__eq__`, `__float__`, `__floordiv__`, `__ge__`, `__get__`, `__getattr__`, `__getattribute__`, `__getitem__`, `__getslice__`, `__gt__`, `__hash__`, `__hex__`, `__iadd__`, `__iand__`, `__idiv__`, `__ifloordiv__`, `__ilshift__`, `__imod__`, `__imul__`, `__index__`, `__init__`, `__instancecheck__`, `__int__`, `__invert__`, `__ior__`, `__ipow__`, `__irshift__`, `__isub__`, `__iter__`, `__itruediv__`, `__ixor__`, `__le__`, `__len__`, `__long__`, `__lshift__`, `__lt__`, `__metaclass__`, `__mod__`, `__mro__`, `__mul__`, `__ne__`, `__neg__`, `__new__`, `__nonzero__`, `__oct__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rand__`, `__rcmp__`, `__rdiv__`, `__rdivmod__`, `__repr__`, `__reversed__`, `__rfloordiv__`, `__rlshift__`, `__rmod__`, `__rmul__`, `__ror__`, `__rpow__`, `__rrshift__`, `__rshift__`, `__rsub__`, `__rtruediv__`, `__rxor__`, `__set__`, `__setattr__`, `__setitem__`, `__setslice__`, `__slots__`, `__str__`, `__sub__`, `__subclasscheck__`, `__truediv__`, `__unicode__`, `__weakref__`, `__xor__`.

12.4 Copiare gli oggetti

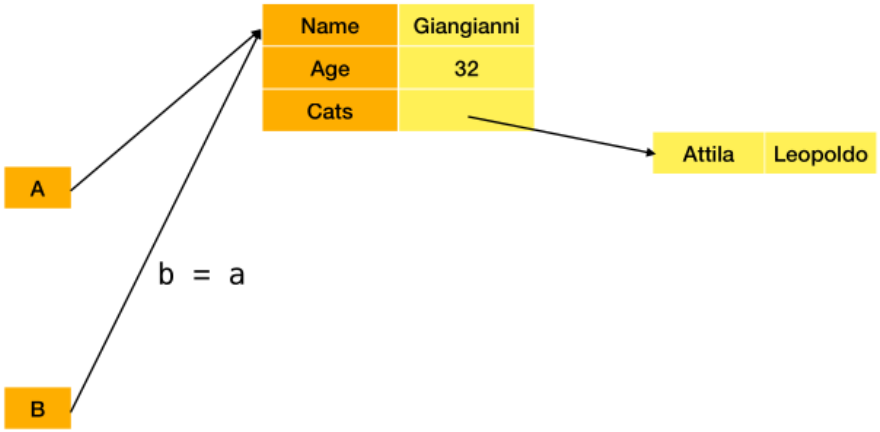
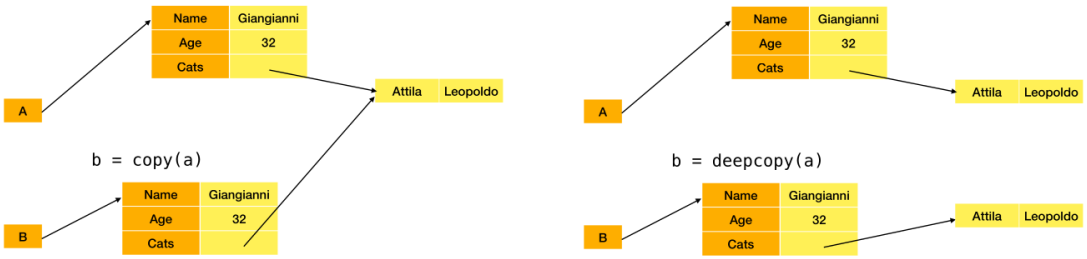


Figura 12.1: Istanza di un oggetto

Usando la libreria <copy> otteniamo le funzioni di copia degli oggetti:

- `copy`: fornisce una copia "shallow" dell'oggetto, quindi copia tutti gli attributi in una nuova istanza dell'oggetto
- `deepcopy`: fornisce una copia in cui anche tutti gli attributi vengono copiati, quindi riapplica ricorsivamente la copia sugli attributi



(a) copy (b) deepcopy

Figura 12.2: Copia di un oggetto

È possibile definire i dunder methods `__copy__(self)` e `__deepcopy__(self)` per fare operazioni speciali in fase di copiatura.

12.5 Ereditarietà

Spesso una serie di classi modellano concetti simili e quindi il contenuto dei metodi risulta praticamente duplicato. Questo è dovuto al fatto che i metodi solo legati a una classe e quindi non possiamo dire direttamente "usa lo stesso metodo per entrambe queste classi".
È possibile creare una classe base e due classi derivate che ereditano i metodi/attributi dalla classe base:

Listing 12.2: Ereditarietà

```
1 class Base:
2     # code...
```

```
3
4 class A(Base):
5     # code...
6
7 class B(Base):
8     # code...
```

Nelle classi derivate è possibile ridefinire dei metodi che erano già definiti nella classe base, in questo caso si ottiene un overriding del metodo. Per riutilizzare codice della superclasse è necessario usare la funzione `super()`.
Una classe può derivare da più di una classe base, ma è possibile che più di una classe base abbiamo un metodo nominato allo stesso modo, in questo caso in Python verrà preso il metodo della prima classe da cui ereditiamo.

12.6 Eccezioni

In Python (come in molti linguaggi moderni) possiamo usare le eccezioni. Quando viene lanciata una eccezione si risale lo stack delle chiamate fino a trovare chi può gestirla.
Possiamo gestire le eccezioni come:

Listing 12.3: Eccezione

```
1 try:
2     # code with raise ExceptionClass
3 except ExceptionClass:
4     # exception handler
5 except MyException as error:
6     # ...
```

Possiamo avere più clausole `except` con classi diverse: verrà usato il codice nella prima che fa match. Quindi usiamo le clausole `except` sempre dalla classe più specifica a quella più generica.
Possiamo rilanciale l'eccezione all'interno di `except` chiamando `raise` (senza nulla dopo).
Possiamo aggiungere una clausola `finally:` dopo i vari `except` per indicare del codice da eseguire sempre (indipendentemente dal fatto che l'eccezione ci sia stata o no).
Le eccezioni vengono sollevate con `raise`:

Listing 12.4: Eccezione

```
1 raise Exception()
2 raise MyException("Message")
```

La classe `Exception` ha un metodo `add_note(note)` che serve per aggiungere informazioni all'eccezione.

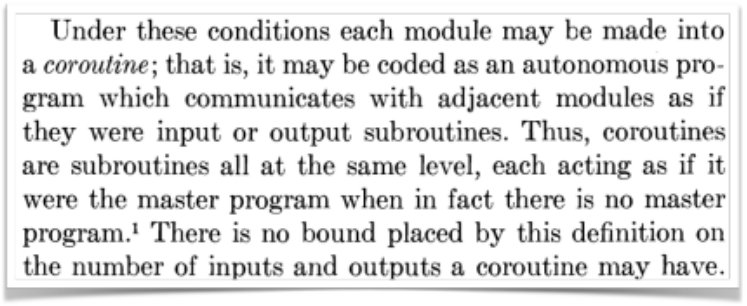
Capitolo 13

Generatori

Quando iteriamo a volte non è necessario avere tutta la lista/array su cui iteriamo (soprattutto se è potenzialmente infinito), è sufficiente poter generare gli oggetti uno alla volta.
Possiamo utilizzare i generatori un caso particolare di coroutine. Una coroutine ha due principali caratteristiche:

1. Lo stato delle variabili locali viene preservato tra chiamate successive
2. La coroutine è sospesa quando il controllo esce al di fuori della coroutine, quando riprende il controllo l'esecuzione riprende da dove si era interrotta

Le coroutine furono scoperte alla fine degli anni '50 e formalmente definite nel 1963:



Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

Conway, Melvin E., *Design of a Separable Transition-diagram Compiler*
Communications of the ACM. ACM. 6 (7): 396–408, 1963

Figura 13.1: Definizione di coroutine

Nel caso particolare dei generatori, in Python viene usata la keyword `yield` per rendere disponibile il valore all'esterno di una funzione e interrompere momentaneamente l'esecuzione della funzione.
Il generatore viene costruito facendo una normale chiamata alla funzione `x = f(...)`, mente per ottenere il primo valore (e i successivi) viene chiamato `next(x)` (se non c'è viene prodotta l'eccezione `StopIteration`).

Listing 13.1: yield

```
def squares(n_elems):
    for n in range(n_elems):
        yield n**2

x = squares(10)
print(next(x))
print(next(x))

# Oppure tramite un iteratore
```

```
for i in s:
    print(i)
```

13.1 Comprehension

Così come abbiamo list comprehension possiamo effettuare comprehension coi generatori:

Listing 13.2: Comprehension

```
# Per liste
[x for x in range(10)]

# Per generatori
(x for x in range(10))
```

Mentre la lista deve essere generata tutta (e quindi stare in memoria), il generatore può costruire gli elementi uno alla volta.

Capitolo 14

Iteratori

Possiamo iterare sugli argomenti di una collezione usando anche oggetti che rispettano un "iterator protocol", ovvero si comportano come iteratori. Ci basta definire due motodi:

- `__iter__(self)` svolge una inizializzazione (se necessaria) e ritorna l'iteratore
- `__next__(self)` ritorna il valore successivo nella sequenza (o genera una eccezione `StopIteration` se non c'è un valore successivo)

Un oggetto che implementa quei due metodi può essere usato come un generatore nei cicli `for` oppure ottenendo l'iteratore con la funzione `iter` e chiamando `next` su di esso.

Listing 14.1: Iteratore

```
class MySet:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        self._pos = 0
        return self

    def __next__(self):
        if self._pos >= len(self._data):
            raise StopIteration
        else:
            res = self._data[self._pos]
            self._pos += 1
            return res

s = MySet([1, 2, 3, 4])

for x in s:
    print(x)

x = iter(s)
print(next(x))
```

14.1 Itertools

Per lavorare su iteratori esiste la libreria itertools che fornisce ad esempio le funzioni:

- `chain(a, b, ...)`: itera su tutti gli elementi di a, poi su tutti quelli di b, etc...
- `product(a, b, ...)`: itera su tutte le combinazioni di valori di a, b, ... (è come avere un ciclo innestato)

- `pairwise(a)`: itera su coppie della forma $(a[0], a[1])$, $(a[1], a[2])$, ...
- `permutations(a)`: itera su tutte le permutazioni degli elementi di a
- `combinations(a, m)`: itera su tutte le tuple ordinate di m elementi di a senza ripetizioni
- `repeat(x, m)`: ripete il valore x per m volte (se m non è specificato, l'iteratore è infinito)

Capitolo 15

Closures

È possibile de nire funzioni che ricevono funzioni come argomenti e ritornano funzioni[1].

Listing 15.1: Funzioni innestate

```
def f():
    def g(x):
        return x+4
    return g

h = f()
h(3)
```

Se dentro una funzione "interna" usiamo una variabile locale della funzione esterna, si ottiene una closure che è composta da due part:

1. Una funzione f
2. Un ambiente che a ognuna della variabili libere di f associa il valore che aveva al momento della definizione di f

Listing 15.2: Closure

```
def f(x):
    def g(y):
        return x+y
    return g

h = f(5)
h(3)
```

15.1 Lambda

Possiamo creare funzioni senza dar loro un nome usando la keyword `lambda`.
Il nome stesso delle funzioni anonime è "lambda functions" (riferimento al lambda calcolo di Alonso Church). Nel caso di Python sono molto più limitate che in altri linguaggi: sono limitate ad una singola espressione, quindi niente flusso di controllo complesso o espressioni multiple.

Listing 15.3: Lambda

```
f = lambda x: x + 3
```

[1] Poter direttamente manipolare funzioni come se fossero valori significa che le funzioni sono "first-class objects"

15.2 Lavorare con funzioni

Molte operazioni su collezioni sono riconducibili a particolari applicazioni di funzioni su di esse:

- `map(f, collection)`: applica la funzione f a tutti gli elementi della collezione
- `filter(predicate, collection)`: applica il predicato (funzione che ritorna Booleani) agli elementi della collezione e mantiene solo quelli per cui il predicato è vero
- `reduce(f, collection)`: usa la funzione f (che riceve due argomenti) per ridurre la collezione ad un solo valore (importato da <functools>). Ad esempio se f è `lambda x, y: x + y` e la collezione `[1,2,3,4]` allora `f(f(f(1,2), 3), 4)` corrisponde a $((1+2)+3)+4$

A volte può essere utile applicare una funzione solo parzialmente. Ad esempio data $f(x,y)$ e un valore x, vogliamo definire $f_x(y) = f(x,y)$. Questa operazione, detta di "currying", è il trasformare un funzione che prendere argomenti multipli in una sequenza di funzioni che prendono un argomento alla volta. Possiamo usare `partial(f, args)` importandolo da <functools>.

15.3 Decoratori

Idea di base:

- Stiamo definendo una funzione f
- Abbiamo una funzione g che prende come argomento una funzione e ritorna una funzione
- Voglia sostituire f con $g(f)$.

Ricordiamo che `*args` negli argomenti della funzione andrà a raccogliere una lista con tutti gli argomenti passati, mentre come argomento di una funzione spezza una lista di argomenti e li passa alla funzione come singoli argomenti. Per i keywords usiamo analogamente `**kwargs`.
I decoratori vengono definiti formalmente come:

Listing 15.4: Decoratore

```
def decorator(f):
    def wrapped(*args, **kwargs):
        return f(*args, **kwargs)
    return wrapped
```

Il decoratore ritorna una nuova funzione che racchiude f (per aggiungere funzionalità).
Possiamo poi utilizzare un decoratore prima della definizione della funzione con la notazione `@decoratore` , che equivale a definire `g = decorator(f)`:

Listing 15.5: Decoratore

```
@decorator
def f(x, y):
    # code...
```

È possibile definire delle funzioni che ritornano decoratori, in pratica sfruttiamo le closure per costruire un decoratore diverso a seconda degli argomenti.

Listing 15.6: Decoratore con argomenti

```
def decorator(parameter)
    def inner(f):
        def wrapper():
            return f(parameter)
        return wrapper
    return inner
```

```
7
8  @decorator(1)
9  def f(x):
10     # code...
```

Data una funzione possiamo accedere a diverse informazioni come nome, docstring, etc... (es. `f.__name__`). Usare i decoratori inoltre "maschera" il nome vero della funzione. Usare `@wraps(f)` di <functools> ci permette di mantenere nome e docstring corrette anche usano i decoratori.

Capitolo 16

Numpy

Numpy è una libreria per array ad alte prestazioni, usata come fondamento di altre librerie (es. pandas).
Le liste Python sono più simili a tabelle hash: operazioni di `append` efficiente, ma utilizzano più memoria del minimo necessario. Gli array in numpy sono più simili a vettori in C: append può richiedere riallocazione, contiguità in memoria dei dati.
Gli array in numpy sono più compatti: più veloci delle liste quando è possibile svolgere la stessa operazione su tutti gli elementi, più lenti se la loro dimensione deve essere continuamente modificata (es. append), funzionano meglio se possono lavorare su tipi omogenei (es. tutti float o tutti interi) piuttosto che eterogenei.

16.1 Creazione di un array

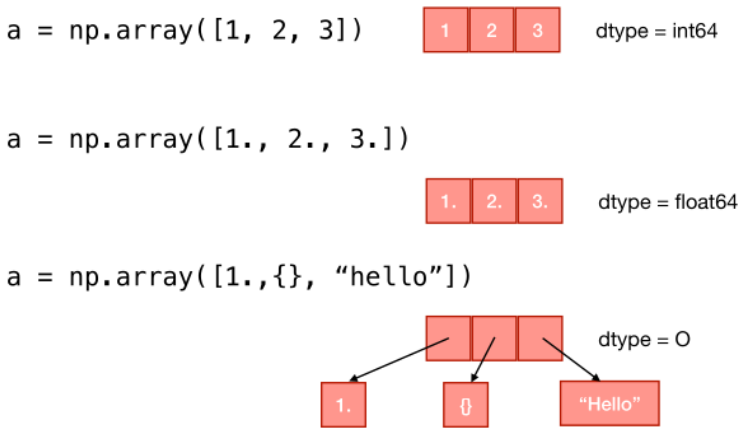


Figura 16.1: Creazione di un array

In numpy vi è una gerarchia di tipi e generalmente è meglio avere tutti gli elementi dello stesso tipo per garantire buone prestazioni:

- Tipi interi con segno: int8, int16, int32, int64
- Tipi interi senza segno: uint8, uint16, uint32, uint64
- Tipi floating point: float16, float32, float64

Alcune funzioni di creazione di un'array sono:

- `np.zeros(n)` genera una array di n float64 inizializzati a 0

- `np.zeros(n, dtype = type)` genera un array di n zeri del tipo specificato
- `np.ones(n)` come
- `np.zeros(n)` ma inizializzati a 1
- `np.full(n, x)`
- `np.zeros(n)` ma inizializzati a x
- `np.empty(n)` genera un array di n float64 non inizializzati
- `np.ones_like(a)` genera un array di n float54 copiano lo shape del vettore passato

Definito `rng = np.random.default_rng()` (per alcune funzioni), è possibile riempire un'array usando dei valori pretabiliti:

- `np.arange(end)` $\longrightarrow [0, 1, \dots end - 1]$
- `np.arange(begin, end)` $\longrightarrow [begin, begin + 1, \dots end - 1]$
- `np.arange(begin, end, step)` $\longrightarrow [begin, begin + step, \dots end - 1]$
- `np.arange(begin, end, len)` $\longrightarrow [begin, begin + \delta, \dots end]$ t.c. il numero di elementi sia `len`
- `rng.integers(a, b, n)`: n interi random da un'uniforme in $[a, b)$ (b è incluso se `endpoint = True`).
- `rng.random(n)`: n float random in $[0, 1)$.
- `rng.standard_normal(a, b, n)`: n float random da una normale standard
- `rng.normal(a, b, n)`: n float random da una normale di parametri $\mu = a$ e $\sigma = b$

Proprietà Supponendo di avere un array numpy a:

- `a.dtype` ritorna il tipo degli elementi dell'array
- `a.shape` ritorna una tupla con le dimensioni dell'array
- `a.size` ritorna il numero di elementi totali
- `a.astype(type)` converte (se possibile) gli elementi al tipo indicato

16.2 Indicizzazione

Possiamo usare un array numpy come una lista Python per l'indicizzazione:

- `a[i]`: singolo elemento
- `a[a:j]`: range di indici tra i (incluso) e j (escluso)
- `a[begin:end:step]`

Contrariamente alle liste Python non viene creata una copia, ma solo una vista (view). Solo alcuni metodi ci permettono di tornare una copia dell'array (oppure usando l'attributo `.copy`).
Nel caso venga passato un array di indici, allora viene tornata una copia: `B = A[[1, 3, 5]]`.
Nel caso venga passato un array di Booleani della stessa lunghezza del'array originale, allora avviene una maschera Booleana di selezione.

Condizioni Booleane Possiamo esprimere una condizione su un array e ci viene ritornato un vettore di Booleani.

Listing 16.1: Maschera Booleana

```
A = np.array([1, 2, 3, 4, 5])
B = A > 3 # [False, False, False, True, True]
```

quindi possiamo usare questa condizione per indicizzare gli array `B = A[A > 3]`. Inoltre possiamo combinare gli array di Booleani con `&` (and), | (or), `^` (xor) e `~` (not).
Numpy inoltre ci forniscono le funzioni:

- `np.any(condizione)`
- `np.all(condizione)`
- `np.clip(a, min, max)`: se un elemento è inferiore a `min` viene cambiato con il valore di `min`, viceversa per `max`
- `np.where(condizione)`: ritorna una tupla di indici dove la condizione è verificata

altre sono: `np.sqrt(A)`, `np.exp(A)`, `np.log(A)`, `np.sin(A)` (cos, tan, arcsin, ...), `np.floor(A)`, `np.ceil(A)`, `np.round(A)`, `np.dot(A, B)` (esprimibile anche come `A @ B`), `np.sort(A)` (ritorna una copia ordinata).
Gli array hanno anche i metodi utili: `A.sum()`, `A.mean()`, `A.var()`, `A.std()`, `A.min()`, `A.max()`, `A.argmin()`, `A.argmax()`.

Capitolo 17

Parallelismo

17.1 Global Interpreter Lock

Il GIL è un lock che protegge l'accesso a tutti gli oggetti Python. Quindi un solo thread può interpretare il bytecode Python (il formato intermedio in cui viene "compilato" Python). Le operazioni che non sono l'interpretazione del codice Python possono avvenire in parallelo (I/O, utilizzo di operazioni numpy, etc...), ma l'esecuzione di codice Python puro è comunque limitata dal fatto che un solo thread alla volta può eseguirla.
Quando usiamo librerie non legate al GIL (es. che chiamano codice C) non vi sono restrizioni sul parallelismo (es. possiamo chiamare una libreria che usa OpenMP). Se eseguiamo codice Python puro in più processi possiamo eseguire in parallelo: ogni processo ha un suo interprete con un suo GIL.

17.2 Multiprocessing

È possibile utilizzare delle librerie che permettono di lavorare con più processi: incluso in Python è il modulo "multiprocessing".
La librerie multiprocessing permette di creare e controllare processi: oggetti Process (che possono essere avviati), oggetti Pool rappresentanti un insieme di processi a cui è possibile delegare delle operazioni da svolgere).

Process Viene crato con `Process(target=funzione, args=(...))` dove il target è la funzione da eseguire e gli args sonogli argomenti da passare alla funzione. Una volta creato viene avviato con il metodo `.start()` e possiamo attendere che termini con il metodo `.join()`

Queue Viene creata con `Queue()` e rappresenta una coda utilizzabile da più processi per inviare e rivere dati:

- `q.put(object)`: aggiunge un elemento alla cosa (possibilmente bloccante)
- `q.get()`: ottiene la testa (possibilmente bloccante)

È possibile scegliere di avere una eccezione invece di una operazione bloccante.

Lock Viene creato con `Lock()` e rappresenta un mutex che è possibile acquisire con `.acquire()` e rilasciare con `.release()` (come ogni lock c'è rischio di deadlock).

Value e Array Value e array rappresentano un modo di condividere la memoria tra più processi:

- `Value(typecode, start_value)`

- `Array(typecode, start_values)`

dove `typecode` può essere "i" (intero), "d" (double) o "f" (float).[1]

Pool Viene creato con `Pool(processes=num_processes)` e rappresenta un insieme di processi a cui possiamo delegare di svolgere dei compiti. Ad esempio il metodo `.map(function, iterable)` distribuisce l'applicazione della funzione a tutti gli elementi fra i diversi processi. È possibile usare `.close()` per liberare le risorse una volta che tutti i processi hanno terminato i task assegnati.

17.3 Joblib

Per parallelizzazione di cicli joblib prevede una interfaccia comoda e più metodi di parallelizzazione: `Parallel(n_jobs=num_jobs)(iterabile)`. In questo caso la funzione da applicare a ogni elemento è bene che sia racchiusa in delayed: `delayed(f)(x)`.

17.4 Just-In-Time compilation

La compilazione just-in-time (JIT) è una compilazione compiuta durante l'esecuzione del programma: si spende del tempo a compilare, ma se il codice è eseguito più volte questo potrebbe velocizzare l'esecuzione.
Una librerie che fornisce questa funzionalità a Python è <numba> (che usa LLVM): trtamite il decoratore `@jit` è possibile compilare una funzione in codice nativo.

[1] In entrambi i casi di default ogni di questi ha un lock che protegge l'accesso alla variabile.

Capitolo 18

Profiling

Per valutare dove un programma sta utilizzando la maggior parte del suo tempo (e cosa ne limita le performance) è necessario compiere delle misurazioni.
I processori moderni sono dotati di performance counters in hardware, si tratta di registri che contano alcuni eventi (es. numero di istruzioni eseguite, numero di cache miss, etc...). Hanno costo prossimo a zero (si tratta di un contatore binario collegato a qualche componente già esistente), verificando il contenuto dei contatori ad intervalli regolari possiamo ottenere delle informazioni su cosa sta facendo un programma.

18.1 Profiling statico

Un profiler statistico compie dei sample di un programma ad intervalli regolari. Questo ci permette di ottenere informazioni su, per esempio, quanto tempo il programma ha speso nell'eseguire certe funzioni o istruzioni.
Usando `perf`[1] possiamo misurare le prestazioni di un programma tramite il comando:

Listing 18.1: perf

```
1 perf stat ./program
```

questo ci fornisce statistiche riguardanti il numero di istruzioni eseguite (e istruzioni per ciclo di clock) e il numero/percentuale di branch-misses. Aggiungengo il paramentro `-e cache-references,cache-misses` viene riportato il numero di accessi alla cache e quante volte abbiamo avuto dei misses.

Perf report `perf record ./program` crea un report che può essere analizzato con il comando `perf report ./program`: un programma con interfaccia testuale che ci permette di verificare quali funzioni e quali istruzioni hanno impiegato più tempo.

Simulazione Un approccio differente rispetto a quello di prendere dei sample del programma mentre esegue, si simula l'esecuzione di un programma vedendo tutte le istruzioni che eseguirebbero branch o accessi alla memoria e simulando l'effetto sulle cache possiamo anche simulare l'effetto di cache (o, in generale di hardware) diverso da quello presente sulla macchine.
Usando `valgrid` possiamo compiere queste simulazioni: `valgrind --tool=cachegrind --branch-sim=yes ./program`. Simula il programma dando un report di utilizzo della cache e del branch predictor, pero simula solo due livelli di cache: L1 (divisa tra dati e istruzioni) e LL ("last level") che unisce L2, L3, etc...
Cachegrind genera anche un file "cachegrind.out.[numero]". Questo le contiene le informazioni per le singole funzioni, è leggibile usando `cg_annotate cachegrind.out.[numero]` e, se abbiamo compilato

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`

con l'opzione `-g`, diventa possibile annotare il codice sorgente riga per riga. `cg_annotate --auto=yes cachegrind.out.[numero]` mostra le statistiche riga per riga e con opzioni come `--show=Dr,D1mr,DLmr,Bc,Bcm` possiamo limitare le statistiche da mostrare.

Python Python ha un profiler integrato: cProfile. È un profiler deterministico e ha diverse limitazioni, ad esempio è ifficile fare il profiling di codice nativo o di codice parallelo. Ci sono pro ler più moderni, tra questi Scalene[2].

[2] `https://github.com/plasma-umass/scalene`