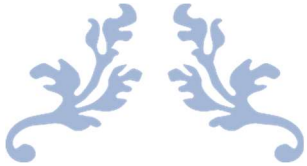

PROCESS SCHEDULING PROBLEM

Tesina di Decision Science

COCCO MATTIA 65336

2024/2025

Università degli Studi di Cagliari

Il problema

Nei calcolatori elettronici che impieghiamo ogni giorno, una serie di processi richiedono risorse computazionali e devono condividere la CPU senza poterla utilizzare contemporaneamente. Ogni processo in genere ha un tempo di esecuzione stimato e a volte una priorità associata, mentre la CPU ha a disposizione un certo numero di core, ciascuno dei quali può eseguire un solo processo per volta.

L'obbiettivo dello **scheduling dei processi** è allocare i processi ai core in modo ottimale, cercando di massimizzare il numero di processi eseguiti in un dato periodo di tempo o il valore complessivo delle loro priorità. Gli algoritmi implementati nello scenario reale devono raggiungere un compromesso: cercano di distribuire bene i processi per evitare stalli o starvation, ma non possono sempre puntare all'ottimalità assoluta di questa distribuzione, poiché ciò introdurrebbe latenza nel sistema.

In questa tesina si risolverà un problema di ottimizzazione riguardante lo scheduling di un insieme di processi su un sistema con 4 core tramite **CPLEX**, che garantisce l'**ottimalità** della soluzione trovata, andando a confrontare i risultati con quelli ottenuti applicando invece dei **metodi euristici**, capaci di trovare soluzioni ammissibili e generalmente buone: due algoritmi di scheduling molto famosi in letteratura chiamati **FCFS** e **SJF**.

Il First Come First Served manda in esecuzione i processi in ordine di arrivo, mentre lo Shortest Job First alloca le risorse prima ai processi con il minor tempo di esecuzione.

Gli esperimenti verranno ripetuti tenendo conto anche delle priorità associate ai processi, modificando opportunamente gli algoritmi; per semplicità non si terrà conto del parallelismo dell'assegnazione dei processi ai core.

Formalizzazione matematica

Notazione

Lo scenario precedentemente descritto è un problema di ottimizzazione, in particolare di **Programmazione Lineare Intera**, simile al problema dello zaino (knapsack problem), in quanto abbiamo un insieme di elementi (i processi) ai quali è associato un peso (il tempo di esecuzione) e un beneficio (l'esecuzione del processo stesso o in relazione alla sua priorità), sui quali dobbiamo cercare di produrre un beneficio totale massimo (il numero di processi eseguiti o la priorità totale dei processi eseguiti), senza superare la capacità massima dello zaino (il tempo limite di esecuzione di ciascun core della CPU).

Stando a questa descrizione, il problema può essere formalizzato come segue:

- n: numero di core della CPU
- m: numero di processi da eseguire
- t_j: tempo di esecuzione richiesto dal processo *j*
- p_j: priorità del processo *j* (da 1 a 10)
- T_{lim}: tempo a disposizione della CPU (quindi di ogni core)

Con le seguenti variabili decisionali binarie:

- $X_{ij} \in \{0, 1\}$: vale 1 se il processo *j* è assegnato al core *i*, altrimenti vale 0.

Funzione obbiettivo

Per quanto riguarda la funzione obbiettivo, si è scelto di condurre gli esperimenti con 2 funzioni obbiettivo diverse: una che non tenesse conto delle priorità associate ai processi e un'altra che invece permettesse di testare gli algoritmi in un'ottica di priority scheduling.

La prima funzione obbiettivo non tiene conto delle priorità e punta semplicemente a massimizzare il numero di processi eseguiti nel tempo limite, lo stesso risultato è ottenibile anche utilizzando la funzione obbiettivo che tiene conto delle priorità, adoperando un dataset i cui processi hanno lo stesso valore di priorità:

$$max \sum_{i=1}^{n} \sum_{j=1}^{m} xij$$

La seconda funzione obbiettivo, come anticipato, tiene conto delle priorità e punta a massimizzare il valore di priorità complessivo dei processi eseguiti nel tempo limite:

$$\boldsymbol{max} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \boldsymbol{xij} * \boldsymbol{pj}$$

Vincoli

Infine, sono stati introdotti i seguenti vincoli per regolare l'assegnazione dei processi ai core, rispettare i tempi limite e definire il dominio delle variabili.

Ogni processo ***j*** può essere assegnato al massimo a un core ***i***:

$$\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{xij} \ \leq \mathbf{1}, \quad \forall \boldsymbol{j} \in \{\mathbf{1}, \dots, \boldsymbol{m}\}$$

Il tempo di esecuzione totale dei processi ***j*** assegnati a un core ***i*** non deve superare il tempo limite $\boldsymbol{T_{lim}}$:

$$\sum_{\boldsymbol{j=1}}^{\boldsymbol{m}} \boldsymbol{tj} * \boldsymbol{xij} \ \leq \boldsymbol{Tlim}, \quad \forall \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{n}\}$$

Le variabili invece, come anticipato, sono binarie e devono dunque rispettare il vincolo:

$$\boldsymbol{xij} \in \{\mathbf{0}, \mathbf{1}\}, \qquad \forall \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{n}\}, \forall \boldsymbol{j} \in \{\mathbf{1}, \dots, \boldsymbol{m}\}$$

Questa formulazione del problema consente di modellare lo scheduling dei processi in modo da massimizzare il numero dei processi eseguiti nel tempo limite, oppure il valore della loro priorità complessiva, trovando la soluzione ottima per un dato insieme di processi e core, come verrà mostrato nel prossimo paragrafo tramite il modello CPLEX.

La soluzione ottima sarà così confrontabile con quella dei noti algoritmi (euristici) FCFS e SJF, evidenziando eventuali differenze in termini di qualità della soluzione.

Algoritmi

Generazione dati

Per iniziare è stata scritta una funzione che genera un set di dati su cui risolvere il problema:

- Un insieme di core con il relativo tempo limite
- Un insieme di processi con i relativi tempi di esecuzione e valori di priorità

Sulla base di:

- Numero di core
- Numero di processi da eseguire
- Tempo limite dei core
- Tempo di esecuzione massimo dei processi generati casualmente

La generazione dei dati è comunque pseudo-randomica, dato che il seme ci permette di ripetere gli esperimenti con lo stesso set di dati in modo da poter confrontare i diversi algoritmi.

```
def generaDati(numeroCore, numeroProcessi, tempoCore, tempoMaxProcessi):

    np.random.seed(59)
    Tlim= [tempoCore] * numeroCore
    t = np.random.randint( low: 1, tempoMaxProcessi, numeroProcessi)  # Tempi di esecuzione dei processi
    p = np.random.randint( low: 1,  high: 10, numeroProcessi)  # Priorità dei processi
    nomi = [f'processo_{i}' for i in range(numeroProcessi)]

    dati = {
        "Tlim": Tlim,
        "processi": [
            {"nome": nomi[i], "t": t[i], "p": p[i]} for i in range(numeroProcessi)
        ]
    }

    return dati
```

Creazione modello

Il modello per risolvere il problema dello scheduling dei processi è stato creato con **Docplex** ed è in grado di trovare la soluzione ottima del problema, associata a un valore della funzione obbiettivo e a una specifica configurazione di assegnazione processi-core.

In fase iniziale vengono estratti i dati ricevuti in input (del tipo appena descritto) e viene creato il modello vero e proprio, tramite la classe Model:

```python
def create_model(dati):

    Tlim = dati["Tlim"]  # Tempo massimo per ogni core
    processi = dati["processi"]  # Dati processi

    numeroCore = len(Tlim)
    numeroProcessi = len(processi)

    t = np.array([p["t"] for p in processi])  # Tempi di esecuzione
    p = np.array([p["p"] for p in processi])  # Priorità
    nomi = [p["nome"] for p in processi]  # Nomi processi

    # Creazione modello
    model = Model("process_scheduling")
```

Come illustrato in precedenza, le funzioni obbiettivo sono due e si differenziano perché una punta a massimizzare la priorità complessiva associata ai processi eseguiti nel tempo limite, mentre l'altra non ne tiene conto e punta a massimizzare il numero dei processi eseguiti.

```python
#ATTENZIONE: in seguito ci sono 2 funzioni obbiettivo, lasciarne solo 1 non commentata

    # Funzione obiettivo per massimizzare il valore totale delle priorità dei processi eseguiti
    model.maximize(model.sum(p[j] * x[i, j] for i in range(numeroCore) for j in range(numeroProcessi)))

    # Funzione obiettivo per massimizzare il numero di processi eseguiti
    model.maximize(model.sum(x[i, j] for i in range(numeroCore) for j in range(numeroProcessi)))
```

Le variabili x_{ij} invece, come anticipato, sono di tipo binario e assumono valore 1 se il processo ***j*** è assegnato al core ***i***, altrimenti 0.

```python
# Variabili binarie: 1 se il processo j è assegnato al core i, 0 altrimenti
x = model.binary_var_matrix(numeroCore, numeroProcessi, name="x")
```

In seguito è stato inserito il vincolo che impone l'assegnazione di un processo ***j*** a massimo un core ***i***:

```python
# Vincolo 1: Ogni processo può essere assegnato a un solo core
for j in range(numeroProcessi):
    model.add_constraint(model.sum(x[i, j] for i in range(numeroCore)) <= 1,  ctname: f"process_assignment_{j}")
```

E per concludere, è stato inserito il vincolo che impone che il tempo limite T_{lim} del core ***i*** non venga superato dal tempo di esecuzione totale dei processi ***j*** assegnatigli.

```python
# Vincolo 2: Tempo totale per core non deve superare il limite
for i in range(numeroCore):
    model.add_constraint(model.sum(t[j] * x[i, j] for j in range(numeroProcessi)) <= Tlim[i],  ctname: f"core_time_limit_{i}")

return model
```

Algoritmi euristici

Gli algoritmi con cui confrontare il modello CPLEX, come menzionato nell'introduzione, sono il First Come First Served e lo Shortest Job First, e non garantiscono l'ottimalità della soluzione trovata.

Questi algoritmi sono stati implementati su python in due versioni: una che tenga conto delle priorità dei processi e l'altra che le ignori. In particolare è opportuno ricordare che il FCFS manda in esecuzione i processi in ordine di arrivo, mentre lo SJF manda in esecuzione i processi in ordine di tempo di esecuzione crescente. Nella versione che tiene conto delle priorità, gli algoritmi funzionano analogamente, ma nel caso dello SJF, a parità di tempo di esecuzione tra due processi, si manda in esecuzione quello con priorità più alta.

Si noti che nel FCFS non cambia nulla all'atto pratico, perché si dovrebbero considerare le priorità di due processi solo in caso di parità di arrivo, che in questa implementazione non può accadere.

FCFS e SJF prendono in input un insieme di dati come il modello CPLEX e restituiscono in output un dizionario contenente la soluzione sottoforma di schema di allocazione, tempi totali dei core e un valore confrontabile con quello della funzione obbiettivo: il numero di processi eseguiti entro il tempo limite o il valore della priorità totale dei processi eseguiti in tale tempo limite, a seconda del fatto che si scelga di tenere conto o meno delle priorità associate ai processi:

```
return {
    "allocation": core_allocation,
    "tempiCores": tempiCores,
    "processiEseguiti": processiEseguiti,
    "prioritaTotale": prioritaTotale
}
```

Esempio su insieme ridotto

Prima di confrontare i tre metodi risolutivi del problema dello scheduling dei processi su vari set di dati generati casualmente, si è svolto un test, senza tenere conto delle priorità, su un insieme di dati ridotto, composto da:

- 4 core con tempo limite = 40.
- 10 processi da eseguire.
- Tempi di esecuzione dei processi compresi tra 1 e 40.
- Priorità dei processi comprese tra 1 e 10.

Per quanto riguarda il modello CPLEX, DOcplex genera in automatico un file, contenente numero di variabili e vincoli, tipo di problema, obbiettivo, funzione obbiettivo, valori di slack e altre informazioni:

```
Model: process_scheduling
 - number of variables: 40
   - binary=40, integer=0, continuous=0
 - number of constraints: 14
   - linear=14
 - parameters: defaults
 - objective: maximize
 - problem type is: MILP
// This file has been generated by DOcplex
// model name is: process_scheduling
```

Per questo insieme di dati, il modello ha trovato la soluzione ottima del problema, che è rappresentata dal valore 8 della funzione obbiettivo (sono stati eseguiti 8 dei 10 processi entro il tempo limite) e da un insieme di variabili x_{ij} che hanno assunto valore 1, indicando che il processo ***j*** è assegnato al core ***i***:

```
objective: 8
status: OPTIMAL_SOLUTION(2)
  x_0_1=1
  x_0_6=1
  x_1_2=1
  x_1_4=1
  x_2_0=1
  x_2_3=1
  x_3_5=1
  x_3_9=1
```

Dando in input lo stesso insieme di dati agli algoritmi euristici, otteniamo invece i seguenti risultati, che identificano delle soluzioni sub-ottime:

```
Numero di Processi eseguiti con FCFS:
7
Numero di Processi eseguiti con SJF:
7
```

Risultati e conclusioni

Per testare il modello CPLEX e gli algoritmi FCFS e SJF, sono stati condotti dieci test su insiemi di dati di dimensioni crescenti.

I set di dati sono stati generati casualmente a partire da questi parametri:

NOME	SEED	CORE	PROCESSI	Tlim	Tmax PROCESSO
s1	11	4	50	100	24
s2	19	4	100	220	30
s3	23	4	100	270	36
s4	38	4	200	500	50
s5	48	4	200	400	40
s6	59	4	200	500	30
s7	65	4	250	500	44
s8	76	4	250	550	45
s9	88	4	250	600	45
s10	93	4	250	700	43

Massimizza il numero dei processi eseguiti

Inizialmente, i test sono stati eseguiti senza tenere conto delle priorità, puntando a massimizzare il numero dei processi eseguiti entro il tempo limite, producendo questi risultati:

DATI	CPLEX	FCFS	SJF	PERCENTUALE FCFS	PERCENTUALE SJF
s1	41	37	40	-0.098	-0.024
s2	75	64	74	-0.147	-0.013
s3	76	66	75	-0.132	-0.013
s4	126	90	125	-0.286	-0.008
s5	127	86	125	-0.323	-0.016
s6	157	137	155	-0.127	-0.013
s7	152	91	151	-0.401	-0.007
s8	151	99	150	-0.344	-0.007
s9	161	112	160	-0.304	-0.006
s10	176	130	175	-0.261	-0.006

Le soluzioni individuate dal modello CPLEX sono ottime per questi problemi di ottimizzazione, mentre quelle trovate dagli algoritmi euristici sono tutte sub-ottime in questo caso.

Nei campi percentuali, viene calcolato di quanto sono peggiori le soluzioni trovate dai modelli euristici, rispetto all'ottimo individuato dal modello CPLEX.

Il metodo **FCFS** ha trovato soluzioni peggiori dell'ottimo in media del **24%** circa, mentre lo **SJF** ha trovato sempre soluzioni molto vicine all'ottimo del problema, differenziandosi in media dell'**1,13%**, non male per un metodo euristico, specialmente se osserviamo i **tempi di esecuzione** degli algoritmi in secondi, con precisione fino al microsecondo:

DATI	CPLEX	FCFS	SJF	PERCENTUALE FCFS	PERCENTUALE SJF
s1	0.016	0.001329	0.000034	-0.917	-0.998
s2	0.047	0.000238	0.00013	-0.995	-0.997
s3	0.016	0.000052	0.000041	-0.997	-0.997
s4	0.031	0.000096	0.000085	-0.997	-0.997
s5	0.046	0.000096	0.000082	-0.998	-0.998
s6	0.016	0.000093	0.000085	-0.994	-0.995
s7	0.047	0.000123	0.000108	-0.997	-0.998
s8	0.031	0.000117	0.000106	-0.996	-0.997
s9	0.063	0.000123	0.000105	-0.998	-0.998
s10	0.077999	0.000248	0.000209	-0.997	-0.997

Come si può notare, gli algoritmi euristici per questi set di dati hanno dei tempi di esecuzione molto inferiori rispetto a quelli del modello CPLEX.

Massimizza la priorità totale

I test sono stati ripetuti, considerando invece le priorità e puntando a massimizzare la priorità complessiva dei processi eseguiti entro il tempo limite, ottenendo i risultati:

DATI	CPLEXP	FCFSP	SJFP	PERCENTUALE FCFS	PERCENTUALE SJF
s1	223	223	192	0	-0.139
s2	431	402	384	-0.067	-0.109
s3	430	406	375	-0.056	-0.128
s4	716	609	639	-0.149	-0.108
s5	742	647	647	-0.128	-0.128
s6	917	903	825	-0.015	-0.1
s7	835	701	746	-0.16	-0.107
s8	874	765	765	-0.125	-0.125
s9	889	782	777	-0.12	-0.126
s10	1007	956	894	-0.051	-0.112

In questo scenario, il metodo euristico **FCFS** è riuscito a trovare una soluzione ottima.

Per il resto, le soluzioni individuate dagli algoritmi euristici sono anche in questo caso sub-ottime. FCFS e SJF performano in maniera simile, distaccandosi dalla soluzione ottima in media dell'**8,7%** e **11,8%** rispettivamente.

Osservando i **tempi di esecuzione**, anche in questo caso i metodi euristici si dimostrano nettamente più veloci del modello CPLEX:

DATI	CPLEXP	FCFSP	SJFP	PERCENTUALE FCFS	PERCENTUALE SJF
s1	0.031	0.000026	0.000025	-0.999	-0.999
s2	0.016	0.000102	0.000133	-0.994	-0.992
s3	0.063	0.000054	0.000045	-0.999	-0.999
s4	0.032	0.000102	0.000088	-0.997	-0.997
s5	0.031	0.000097	0.000088	-0.997	-0.997
s6	0.031	0.000092	0.000088	-0.997	-0.997
s7	0.047	0.000119	0.000109	-0.997	-0.998
s8	0.063	0.000119	0.000109	-0.998	-0.998
s9	0.078	0.000118	0.000116	-0.998	-0.999
s10	0.047	0.000256	0.000253	-0.995	-0.995

Conclusioni

Per concludere, considerando che:

- i metodi euristici si sono avvicinati spesso alle soluzioni ottime, in particolare lo SJF nello scenario senza priorità, dove la differenza era in media dell'1,13%, ma anche il FCFS nel contesto con le priorità, con una inferiorità media dell'8,7%;

- i metodi euristici hanno complessità computazionale di ***O(n processi x m core)***, inferiore all'esponenziale degli algoritmi implementati da CPLEX, che ha permesso a FCFS e SJF di ottenere delle soluzioni ammissibili per i problemi dati in tempi molto inferiori rispetto al modello CPLEX (in media del 99,5%);

in un contesto dove è molto importante la rapidità con cui si ottiene una soluzione, come quello dello scheduling dei processi, può avere senso scendere al compromesso di utilizzare un metodo euristico, che trova una soluzione ammissibile, buona, non ottima, ma permette di risparmiare molto tempo.

Si può così evitare l'introduzione di latenza nel sistema per ricercare una soluzione che in alcuni casi non è così migliore da giustificare i tempi di esecuzione molto più elevati.