Forme di parallelismo (note)
AESO, Anno Accademico 2022–23

M. Danelutto

21 ottobre 2022

Indice

Queste note sono un riassunto molto conciso delle cose che servono per comprendere le forme di parallelismo utilizzate nei processori che abbiamo visto nell'ultima parte del corso. Prima prendiamo in considerazione quali sono le *misure* di interesse per il parallelismo. Poi facciamo vedere alcuni esempi di parallelismo nel processore visti da questo punto di vista un po' più generale di quello sbrigativamente adottato nel libro di testo.

0.1 Computazioni sequenziali e parallele

Introduciamo con alcuni piccoli esempi il concetto di attività parallela in contrapposizione al concetto di attività sequenziale.

Immaginiamo di aver bisogno di tradurre un libro dall'inglese all'italiano. Una singola persona può cominciare a tradurre il libro dalla prima pagina e proseguire fino all'ultima pagina. La singola persona che traduce è in questo caso il nostro "processore" e la traduzione avviene sequenzialmente, una pagina dopo l'altra. Assumendo che mediamente la persona impieghi un certo tempo t_p per tradurre una singola pagina e che il libro contenga m pagine, il tempo necessario per la traduzione sarà $m \times t_p$.

Immaginiamo adesso di avere a disposizione n persone in grado di tradurre testo dall'inglese all'italiano, tutte in grado di tradurre una singola pagina in un tempo pari al t_p di prima. Quello che potremmo fare è assegnare ad ognuno degli n traduttori un n-esimo delle pagine da tradurre. Dunque dividiamo il libro in n parti da $\frac{m}{n}$ pagine ed assegnamo ognuna delle parti ad uno dei traduttori.

Tutti i traduttori lavoreranno per un tempo pari a $\frac{m}{n} \times t_p$ ed al termine potremmo riunire le diverse parti del libro tradotto a formare il risultato finale "intero libro tradotto".

In questo caso abbiamo utilizzato un insieme di n "processori" che hanno lavorato in parallelo su compiti completamente indipendenti fra di loro. Il risultato netto è stata una chiara diminuzione del tempo necessario per tradurre il nostro libro. Se assumiamo di aver speso un certo t_{split} per dividere il libro in parti uguali ed assegnarle ai traduttori e un certo tempo t_{merge} per ricomporre il libro tradotto, potremmo dire che la traduzione del libro ha richiesto un tempo pari a

$$t_{split} + \frac{m \times t_p}{n} + t_{merge}$$

a fronte di un tempo "sequenziale" pari al valore precedentemente calcolato come

$$m \times t_p$$

Quanto abbiamo guadagnato in questo caso, in termini di tempo necessario per la traduzione? Siamo andati

$$\frac{m \times t_p}{t_{split} + \frac{m \times t_p}{n} + t_{merge}}$$

volte più veloci (rapporto fra il tempo impiegato per tradurre il libro da soli e quello impiegato per tradurre il libro con n traduttori *in parallelo*). Se i tempi per dividere il libro in parti e per ricomporre il risultato finale fossero trascurabili rispetto al tempo necessario per tradurre $\frac{m}{n}$ pagine, potremmo dire che il guadagno è stato

$$\frac{m \times t_p}{\frac{m \times t_p}{n}} \equiv n$$

ovvero è direttamente proporzionale al numero di traduttori ("processori") utilizzati.

Consideriamo un altro esempio, completamente diverso: costruire una serie di oggetti (m oggetti in tutto) ciascuno dei quali richiede per costruzione una sequenza di operazioni $f_1, \ldots, f_k$ ciascuna delle quali richiede un tempo t_i per essere completata.

Se facessi il lavoro da solo, impiegherei un tempo pari a

$$\sum_{i=1}^{k} t_i$$

tempo per costruire un oggetto e di conseguenza un tempo pari a

$$m \times \sum_{i=1}^{k} t_i$$

per costruirne m.

Se fossimo k costruttori potremmo però organizzarci a "catena di montaggio": il primo costruttore potrebbe compiere l'operazione f_1 e passare il risultato dell'azione al secondo costuttore. Questo potrebbe eseguire l'operazione f_2 e passare il risultato al terzo costruttore, e così via. Alla fine della catena, l'ultimo costruttore riceverebbe il pezzo già lavorato con le operazion $f_1, \ldots, f_{k-1}$ svolgerebbe l'operazione f_k e produrrebbe finalmente il primo oggetto. Tuttavia, mentre il costruttore i esegue l'operazione f_i su un pezzo appena ricevuto dal costruttore $i-1$ quest'ultimo potrebbe eseguire l'operazione f_{i-1} sul pezzo successivo e il costruttore $i+1$ potrebbe eseguire l'operazione f_{i+1} sul pezzo precedente della seguenza di oggetti in costruzione.

Questo modo di procedere fa avanzare i pezzi da un costruttore all'altro con un ritmo dettato dal costruttore più lento. Nell'ipotesi che tutte le operazioni richiedano lo stesso tempo t_o per essere eseguite, questo significa che ogni t_o ciascun costruttore passa al costruttore successivo il pezzo appena lavorato e comincia ad eseguire l'operazione f_i sul pezzo successivo. Se invece ognuna delle operazioni richiedesse un tempo t_i diverso, dovremmo considerare il ritmo di avanzamento degli oggetti come $max\{t_1, \ldots, t_k\}$, visto che l'operazione più lenta bloccherebbe lo scorrimento finchè non venisse completata indipendentemente dal fatto che le altre operazioni potrebbero essere già concluse.

In queste condizioni, il tempo necessario per produrre m oggetti potrebbe essere visto come somma di due tempi:

- il tempo necessario al primo oggetto ad uscire dalla catena di montaggio (riempimento della catena: dopo questo tempo tutti i costruttori lavorano su un pezzo diverso)
- il tempo necessario all'ultimo costruttore a completare gli altri $m-1$ pezzi (ovvero $(m-1)t_o$ oppure $(m-1) \times max\{t_1, \ldots, t_k\}$ a seconda dei due casi).

Con queste considerazioni il tempo per produrre m pezzi sarebbe quindi

$$\sum_{i=1}^{k} t_i + (m-1) \times max\{t_1, \ldots, t_k\}$$

Nell'ipotesi che $m >> k$ (cioè che dobbiamo produrre molti più pezzi del numero dei costruttori) possiamo trascurare sia la sommatoria iniziale che il -1 nel moltiplicatore del massimo e dire che il tempo per produrre gli m oggetti è circa

$$m \times max\{t_1, \ldots, t_k\}$$

Il guadagno rispetto al tempo impiegato da un singolo costruttore sarebbe dunque di

$$\frac{m \times \sum_{i=1}^{k} t_i}{m \times max\{t_1, \ldots, t_k\}} \cong \frac{m \times k \times t_o}{m \times t_o} = k$$

volte.

Questi due esempi mostrano due istanze di parallelismo “spaziale” e “temporale”, secondo la terminologia adottata nel libro di testo. Nel primo caso, si tratta di parallelismo spaziale perchè l’elaborazione avviene su dati diversi in “luoghi” diversi. Nel secondo caso è parallelismo “temporale” perchè in qualche modo l’elaborazione di un dato avviene in “luoghi” diversi in tempi diversi.

0.2 Misure

In questa sezione, definiamo formalmente un certo numero di misure che serviranno per valutare gli effetti dell’introduzione di forme di parallelismo.

- **Latenza**
  La latenza misura il tempo fra l’inizio di un calcolo e la produzione del relativo risultato. La latenza si indica normalmente con L.

- **Tempo di completamento**
  Misura relativa all’esecuzione di un certo numero m di calcoli (normalmente calcolo di una stessa funzione f) su un insieme di dati in ingresso $x_1, \ldots, x_m$. Il tempo di completamento (T_C) rappresenta l’intervallo fra il tempo in cui inizia il primo calcolo e quello in cui termina l’ultimo calcolo.

- **Tempo di servizio**
  Misura relativa all’esecuzione di un certo numero m di calcoli (normalmente calcolo di una stessa funzione f) su un insieme di dati in ingresso $x_1, \ldots, x_m$. Il tempo di servizio (T_S) è il tempo che intercorre fra la produzione di due risultati consecutivi o fra l’inizio di due calcoli consecutivi.

- **Throughput**
  E’ l’inverso del tempo di servizio, ovvero il numero di calcoli completati per unità di tempo. Spesso viene detto anche “banda di elaborazione” (B).

Dalla definizione segue che diminuire la latenza significa rendere eseguire una singola computazione più velocemente, mentre diminure il tempo di servizio (aumentare il throughput) significa produrre più risultati nella stessa quantità di tempo.

0.3 Misure derivate

Utilizzando le misure (primitive) appena definite possiamo definire alcune misure, tutte definite in funzione del grado di parallelismo n:

- **Speedup**
  E' rapporto fra il miglior tempo sequenziale e il tempo utilizzato in parallelo con grado di parallelismo pari a n:

$$\text{Speedup}(n) = \frac{T_{seq}}{T_{par}(n)}$$

- **Scalabilità**
  E' il rapporto fra il tempo impiegato a calcolare con grado di parallelismo 1 e quello impiegato con grado di parallelismo n:

$$\text{Scalab}(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

- **Tempo ideale**
  E' il rapporto fra il tempo sequenziale e il grado di parallelismo n:

$$T_{id} = \frac{T_{seq}}{n}$$

- **Efficienza**
  E' il rapporto fra il tempo ideale e quello ottenuto con grado di parallelismo n:

$$\text{eff}(n) = \frac{T_{id}}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} = \frac{T_{seq}n}{n \times T_{par}(n)} = \frac{\text{speedup}(n)}{n}$$

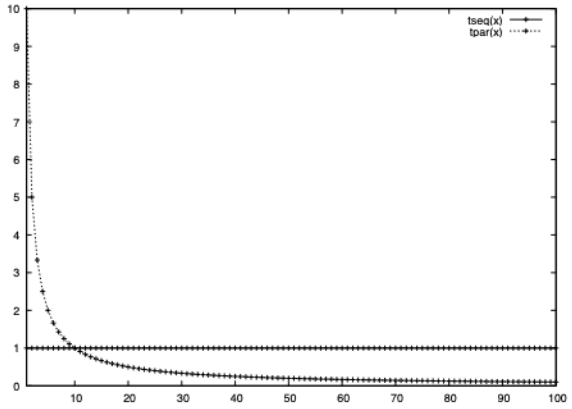
Lo speedup misura di quanto miglioriamo il tempo dell'esecuzione sequenziale ed è normalmente un valore che è compreso fra la retta $y = x$ (bisettrice del primo quadrante) e l'asse delle ascisse, ovvero vale che

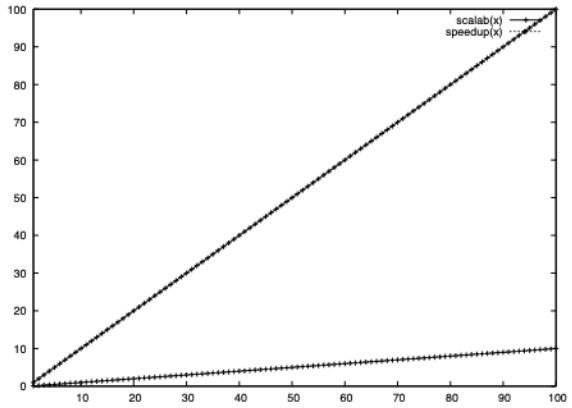
$$\text{speedup}(n) \leq n$$

La scalabilità misura invece quanto, data una soluzione parallela, questa soluzione riesce ad adattarsi a gradi di parallelismo diversi. Dal momento che ci confrontiamo con il caso "soluzione parallela con grado di parallelismo 1" sostanzialmente misuriamo quanto aumentando il grado di parallelismo aumenti anche il guadagno. Questo però non dice nulla relativamente allo speedup. Potremmo avere una soluzione sequenziale che calcola il risultato in $T_{seq} < T_{par}(1)$ e quindi ci potremmo trovare nella condizione che il tempo parallelo diventa migliore del tempo sequenziale solo dopo un certo grado di paralleliesmo, pur in presenza una scalabilità lineare.

La situazione è riassunta dai due grafici che seguono, che rappresentano una situazione abbastanza tipica e non ideale: il primo rappresenta il tempo impiegato per calcolare quanto dobbiamo calcolare in funzione del grado di parallelismo (asse delle ascisse). La retta rappresenta il tempo sequenziale.

La curva rappresenta il tempo parallelo. Il secondo rappresenta la scalabilità (retta più alta) e lo speedup (retta più bassa). Notate che la scalabilità è ideale (per grado di parallelismo n abbiamo una scalabilità di n mentre per lo speedup abbiamo uno speedup di $\frac{n}{10}$ per grado di parallelismo n.





L'efficienza ci dice quanto riusciamo a sfruttare le risorse a disposizione ed è normalmente un valore che è compreso fra la retta $y = 1$ e l'asse delle ascisse, ovvero vale che

$$\text{eff}(n) \leq 1$$

Nel caso dei grafici precedenti, l'efficienza sarebbe sempre pari a 0.1, ovvero molto bassa rispetto all'ideale, cioè 1.

0.4 Forme di parallelismo

Descriviamo in maniera un po' più formale le forme di parallelismo che utilizzeremo poi per la realizzazione del processore.

0.4.1 Pipeline

Un pipeline è costituito da un certo numero di *stadi* $S_1, \ldots, S_k$ ognuno dei quali calcola una certa funzione (assumiamo che S_i calcoli f_i. L'uscita dello stadio i è ingresso dello stadio $i+1$. Il primo stadio riceve dall'esterno un certo numero di task $x_1, \ldots, x_m$. I task arrivano in istanti distinti, tipicamente ogni T_a. L'ultimo stadio produce un numero pari a m di valori in uscita, diretti all'esterno del pipeline. Il valore $j-$esimo y_j è il risultato del calcolo di $f_m(f_{m-1}(\ldots f_2(f_1(x_j))\ldots))$.

Per il pipeline con k stadi che processa uno stream di m elementi, detti rispettivamente L_i e T_i la latenza e il tempo di servizio dello stadio i, sotto condizione che $T_a < max\{T_1, \ldots, T_k\}$ abbiamo che:

$$T_S(k) = max\{T_1, \ldots, T_k\}$$

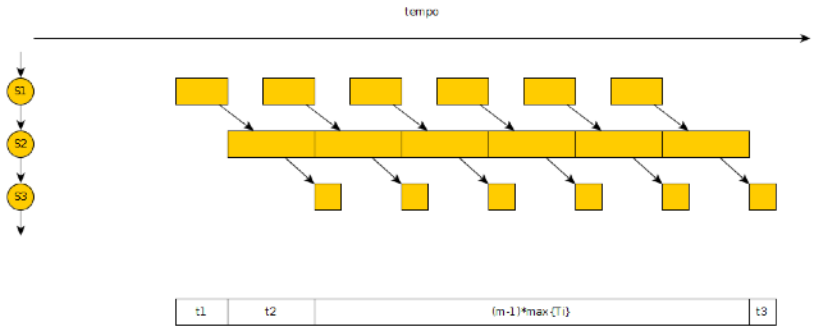
$$T_C(k,m) = \sum_{i=1}^{k} T_i + (m-1) * T_S$$

e il massimo speedup che possiamo ottenere rispetto al caso sequenziale è pari a k, numero degli stadi. In realtà, senza alcuna assunzione su T_a vale che

$$T_S = max\{T_a, T_1, \ldots, T_k\}$$

dal momento che indipendentemente dalla velocità di esecuzione dei task nel pipeline non possiamo andare più veloci del tempo impiegato dai task per arrivare al pipeline stesso.

Il grafico che segue fa vedere come vengono calcolati diversi task nel tempo (tempo che va da sinistra versa destra) nei tre stadi (dall'alto verso il basso). La parte in basso fa vedere il tempo di completamento come sommatoria dei tempi dei vari stadi (due all'inizio e uno alla fine, visto che lo stadio più lungo è quello intermedio) più numero dei task meno uno volte il tempo dello stadio più lento.



Rispetto alla terminologia del libro, il pipeline esprime parallelismo “temporale”.

Un pipeline permette di aumentare il throughput ma non di diminuire la latenza di una computazione sequenziale.

Farm

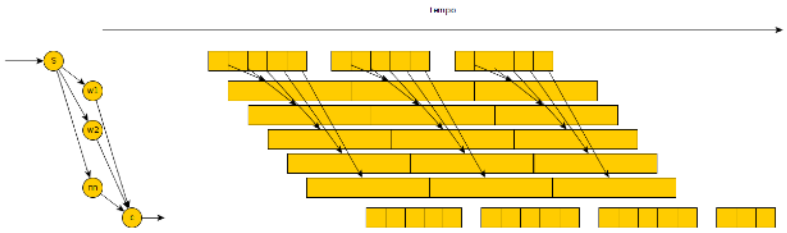
Un farm è costituito da un certo numero n_w di worker, ognuno dei quali riceve in ingresso un dato x, calcola la stessa funzione f e produce in uscita un risultato $f(x)$. Il farm riceve dall'esterno un certo numero di task $x_1, \ldots, x_m$ e produce come risultati un numero pari a m di valori in uscita, diretti all'esterno del farm, di valore $f(x_1), \ldots, f_{x_m}$. Come per il pipeline, i task in ingresso arrivano in istanti di tempo diversi, ogni T_a.

Per il farm con n_w worker che processa m task in ingresso che arrivano ogni T_a, assumento di impiegare T_{sched} per assegnare un task in ingresso ad un worker, T_{coll} per raccogliere un valore calcolato da un worker e spedirlo in uscita al farm e T_w per calcolare una singola $f(x)$ in uno dei worker, vale che:

$$T_S(n_w) = max\{T_a, T_{sched}, T_{coll}, \frac{T_w}{n_W}\}$$

$$T_C(n_w, m) = T_{sched} + \frac{m}{n_W} T_w + T_{coll} \cong m \times T_S$$

Il grafico che segue fa vedere come si svolge la computazione di un certo numero di task nel tempo (da sinistra verso destra) sui vari worker (dall'alto verso il basso).



Il massimo speedup ottenibile con un farm è pari a n_w. Come il pipeline, il farm permette di aumentare il throughput ma non di diminuire la latenza di una computazione seqeunziale.

Map

Una map processa un certo insieme di dati $\{x_1, \ldots, x_m\}$ utilizzando un certo numero n_w di worker. Ciascuno dei worker è in grado di calcolare una funzione f su un qualunque x_i producendo un valore $f(x_i)$. La map riceve l'insieme in input, impiega un certo tempo T_{split} per dividere l'insieme in n_w sottoinsiemi e assegnarli ai worker, i worker calcolano il loro sottoinsieme ed infine la map impiega un certo tempo T_{merge} per ricostruire il risultato finale a partire dai risultati parziali prodotti dai worker, ovvero a partire dall'insieme degli $f(x_i)$.

Per una map con n_W worker che processa un insieme di m elementi computado su ciascun elemento una funzione f che impiega un tempo T_f vale che:

$$L(n_w) = T_C(n_w, 1) = T_{split} + \frac{T_{seq}}{n_W} * T_{coll} = T_{split} + \frac{m * T_f}{n_W} * T_{coll}$$

Il massimo speedup rispetto alla computazione sequenziale è n_W. Qualora si utilizzi la map per calcolare uno stream di insiemi in input, allora il tempo di servizio andrebbe calcolato come:

$$T_s(n_w) = L(n_w)$$

La map dunque permette di dimunire la latenza e di incrementare in throughput di una computazione sequenziale.

La map rapprensenta un pattern di parallelismo "spaziale" secondo la notazione del libro di testo.

0.5 Composizione di forme di parallelismo

Pipeline e farm lavorano su *stream* (sequenze i cui valori esistono in istanti diversi) di dati e sono dette forme di parallelismo stream (stream parallelism). La map lavora su un singolo insieme di dati, scomponendolo in sottoinsiemi e

calcolando il risultato finale come funzione dei risultati calcolati in parallelo sui sottoinsiemi ed è detta forma di parallelismo sui dati (data parallelism).

Forme di parallelismo su stream possono a loro volta avere worker o stadi paralleli di tipo:

- sequenziale
- stream parallel
- data parallel

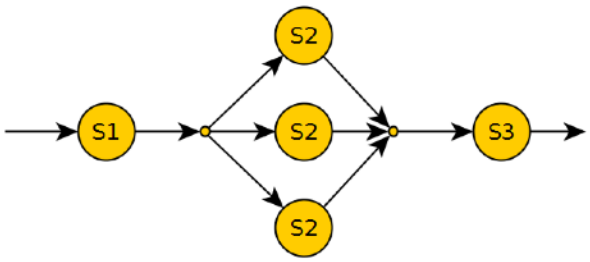
Forme di parallelismo sui dati possono a loro volta avere worker paralleli di tipo:

- sequenziali
- data parallel

ma non stream parallel.

I modelli di prestazioni (T_S e T_C si compongono di conseguenza.

Per esempio, supponiamo di avere un pipeline con tre stadi: il primo e l'ultimo sequenziali con $L1 = t_1$ e $L_3 = t_3$ e il secondo di tipo farm, con n_W worker e $T_w = t_2$.



Per calcolare il tempo di servizio del pipeline:

$$T_{S_{pipe}}(3) = max\{t_1, T_{S_{farm}}, t_3\} \quad (1)$$

dal momento che per gli stadi sequenziali vali chiaramente che $T_S = L$. Considerando il tempo di servizio del farm:

$$T_S(n_W) = max\{T_{sched}, \frac{T_w}{n_W}, T_{coll}\}$$

e sostituendolo nella Eq. 1:

$$T_{S_{pipe}}(3) = max\{t_1, max\{T_{sched}, \frac{T_w}{n_W}, T_{coll}\}, t_3\}$$

ovvero

$$T_{S_{pipe}}(3) = max\{t_1, T_{sched}, \frac{T_w}{n_W}, T_{coll}, t_3\}$$

0.5.1 Ottimizzazione delle computazioni parallele

Per migliorare la performance di una computazione parallela:

- si individua la misura di interesse (e.g. T_S o T_C)
- si individuano eventuali colli di bottiglia (e.g. stadi lenti in un pipeline)
- si cerca di individuare una migliore decomposizione parallela che rimuova il collo di bottiglia (e.g. si trasforma lo stadio lento del pipeline sequenziale in un farm)

Per esempio, consideriamo un pipeline di tre stadi, ognuno dei quali sia inzialmente sequenziale. Ognuno degli stadi calcola sequenzialmente una `map`. Una rappresentazione dell'algoritmo sequenziale in pseudo codice c-like potrebbe essere la seguente:

```
1  /* stadio 1 */
2  for(i=0; i<N; i++)
3    a[i] = f(b[i]);
4  /* stadio 2 */
5  for(i=0; i<N; i++)
6    c[i] = a[i]+b[i];
7  /* stadio 3 */
8  for(i=0; i<N; i++)
9    d[i] = g(c[i]);
```

Supponiamo che il tempo impiegato per calcolare le funzioni `f` e `g` siano rispettivamente t_f e t_g e il tempo necessario a calcolare la somma di due elementi dei vettori (nel secondo stadio) sia t_+ Il tempo necessario a calcolare questi tre stadi su un singolo elemento in input (vettore `b` di `N` elementi) sarà

$$N \times t_f + N \times t_+ + N \times t_g = N(t_f + t_+ + t_g)$$

Il tempo necessario per calcolare sequenzialmente uno stream di m vettori `b` sarà quindi

$$mN(t_f + t_+ + t_g)$$

Supponiamo che $t_f << t_g$ e che t_f sia dello stesso ordine di grandezza di t_+. Chiediamoci quanto tempo spendiamo calcolando i nostri risultati utilizzando un pipeline di tre stadi sequenziali. Il tempo sarà approssimato da

$$m \times max\{t_f, t_+, t_g\} \cong m \times t_g$$

Chiaramente è limitato dal fatto che il terzo stadio sia quello più lento. Sfruttando il fatto che sappiamo che potenzialmente tutti e tre gli stadi sono `map` potremmo pensare di parallelizzare il terzo stadio come `map` utilizzando tanti worker quanti necessari per far diventare il tempo di calcolo simile a quello degli altri stadi, ovvero

$$n_W \cong \frac{t_g}{t_f}$$

Questo è chiaramente possibile se e solo se $\frac{t_g}{t_f}$ rimane maggiore del t_{split} e del t_{merge} spesi per distribuire sottoinsiemi del dato in ingresso ai worker a per collezionare i risultati parziale e ricostruire il risultato finale, diversamente il tempo della map rimarrebbe limitato a $max\{t_{split}, t_{merge}\}$.

Nella migliore delle ipotesi, settando il grado di parallelismo $n_w = \lceil \frac{t_g}{t_f} \rceil$ il tempo di completamento scenderebbe a

$$m \times max\{t_f, t_+, \frac{t_g}{\lceil \frac{t_g}{t_f} \rceil}\} \cong m \times t_f$$

Avremmo anche potuto fare di più. In effetti avremmo implementare delle map in tutti e tre gli stadi. In teoria questo avrebbe permesso di far scendere il tempo di calcolo di tutti e tre gli stadi a $max\{t_{split}, t_{merge}\}$ e quindi avrebbe potuto far scendere il tempo necessario a calcolare m task a

$$m \times max\{t_{split}, t_{merge}\}$$

Va comunque tenuto presente che in questo caso siamo partiti da una computazione che di fatto era un `pipeline` di `map` momentaneamente implementate come sequenziali. Non sempre è così. Se gli stadi non avessero potuto essere trasformati in `map`, avremmo potuto comunque migliorare le prestazioni del pipeline in termini di throughput invece che di tempo di completamento. Se avessimo avuto comunque un massimo nel tempo di calcolo del terzo stadio e questo non potesse essere parallelizzato con una `map`, avremmo potuto trasformare lo stadio in un `farm`. Questo avrebbe ridotto il tempo di servizio del pipeline, riducendo il tempo di servizio del terzo stadio e, di conseguenza, anche il tempo di completamento, pur non riducendo il tempo necessario per il calcolo di un singolo task.

0.6 Esempi

In questa sezione discutiamo alcuni esempi di utilizzo del parallelismo nella microarchitettura. Alcune di queste cose le trovate anche nel libro di testo, nella parte che descrive le caratteristiche avanzate dei processori, ma le ripresentiamo con la terminologia discussa in queste note.

0.6.1 Processore pipeline

Il processore pipeline visto nel capitolo della microarchitettura sfrutta il parallelismo pipeline. Gli stadi si occupano delle varie fasi relative all'esecuzione di una istruzione:

- **fecth**: prelievo dell'istruzione (accesso alla IM all'indirizzo PC)
- **decode**: accesso al register file e/o estrazione delle costanti dalla parola dell'istruzione

- **esecuzione**: utilizzo della ALU per calcolo dei risultati e/o degli eventuali indirizzi per gli accessi in memoria data
- **memoria**: accesso eventuale alla memoria dati per load e store
- **write back**: scrittura nel register file (o nel PC, in caso di salti) dei risultati dell'esecuzione dell'istruzione

Il fatto che la forma di parallelismo si pipeline ha diverse implicazioni:

- il tempo di servizio è dato dal massimo dei tempi di servizio degli stadi (leggi: la lunghezza del ciclo di clock va dimensionata sullo stadio più lento)
- il tempo di completamento per lunghe sequenze di calcoli (ovvero di istruzioni) diventa proporzionale al tempo di servizio dello stadio più lento (m istruzioni impiegano circa $m\tau$ per essere calcolate, con τ lunghezza del ciclo di clock, ovvero tempo di servizio dello stadio più lento)
- se avessimo uno stadio molto lento potremmo cercare di limitarne le conseguenze replicandolo (ovvero trasformandolo in un `farm`). La cosa si può applicare ai soli stadi senza stato (per esempio la ALU) e ha degli impatti sulla performance in caso di dipendenze logiche.
- per abbassare il tempo di servizio potremmo spezzare lo stadio lento in più sotto stadi. Questo si puà fare in alcuni casi, non sempre e ha comunque degli impatti sulla performance in caso di dipendenze logiche, come nel caso precedente.

0.6.2 Unità vettoriale

Quando ci troviamo nella condizione di dover eseguire molte istruzioni su dati diversi di una struttura dati tipo vettore, possiamo pensare di implementare una map:

- carichiamo più dati in un registro possibilmente senza spendere più tempi di accesso in memoria (e.g. utilizzando memorie interallaciate e collegamenti di ampiezza maggiore della singola parola), e
- utilizziamo più ALU in parallelo per operare sulle diversi porzioni dei registri

Di fatto stiamo utilizzando una `map` e quindi:

- riduciamo il tempo (latenza) necessario per calcolare operazioni su tutti gli elementi del vettore di un fattore che può arrivare al massimo a $\frac{N}{n_{alu}}$ (con N numero di elementi nel vettore e n_{alu} numero delle ALU nell'unità vettoriale

Le istruzioni vettoriali (estensioni SSE, AVX (mondo Intel) o NEON (mondo ARM) dell'insieme di istruzioni scalari) permettono di calcolare operazioni fra registri vettoriali visti come vettori di registri scalari. In pratica, permettono di eseguire molto velocemente operazioni tipo

$$\forall i : a[i] = b[i] \; OP \; c[i]$$

con OP operazione aritmetica tipo ADD, SUB, MUL ... o logica tipo AND, OR, ...