# Introduzione al calcolo parallelo e cloud computing

Emanuele Aliverti
aliverti@stat.unipd.it
9 Maggio 2019

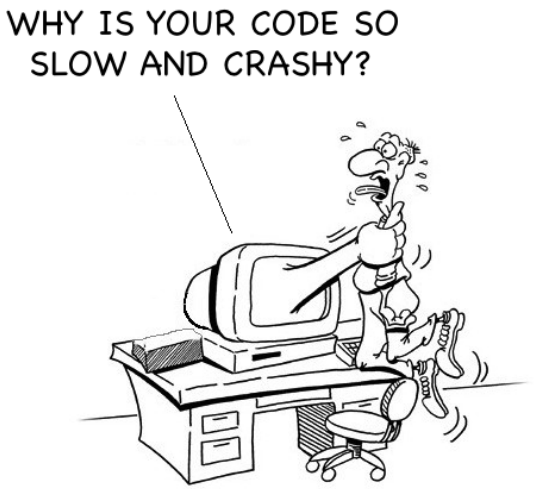
# Perché parlare di calcolo parallelo - algoritmi

- Complessità degli algoritmi
- Algebra *densa*

| Algoritmo | Costo |
|---|---|
| Inversione matriciale $(n \times n)$ | $O(n^3)$ |
| Determinante $(n \times n)$ | $O(n^3)$ |
| SVD $(n \times p)$ | $O(np^3)$ |
| Determinante $(n \times n)$ | $O(n^3)$ |
| Cholesky $(n \times n)$ | $O(n^3)$ |

- Noi siamo interessati all'**analisi dei dati**, dove l'algebra lineare ha un ruolo cruciale

# Impariamo a riconoscere i nostri limiti




1. Limiti della **CPU**: potenza di calcolo
2. Limiti della **RAM**: capacità di memoria
3. Limiti **I/O**: lettura / scrittura di file su disco
4. Limiti di **trasferimento**: copia dati richiede troppo tempo

# Di cosa parliamo oggi

Problemi

1. Limiti della **CPU**
2. Limiti della **RAM**
3. Limiti **I/O**
4. Limiti di **trasferimento**

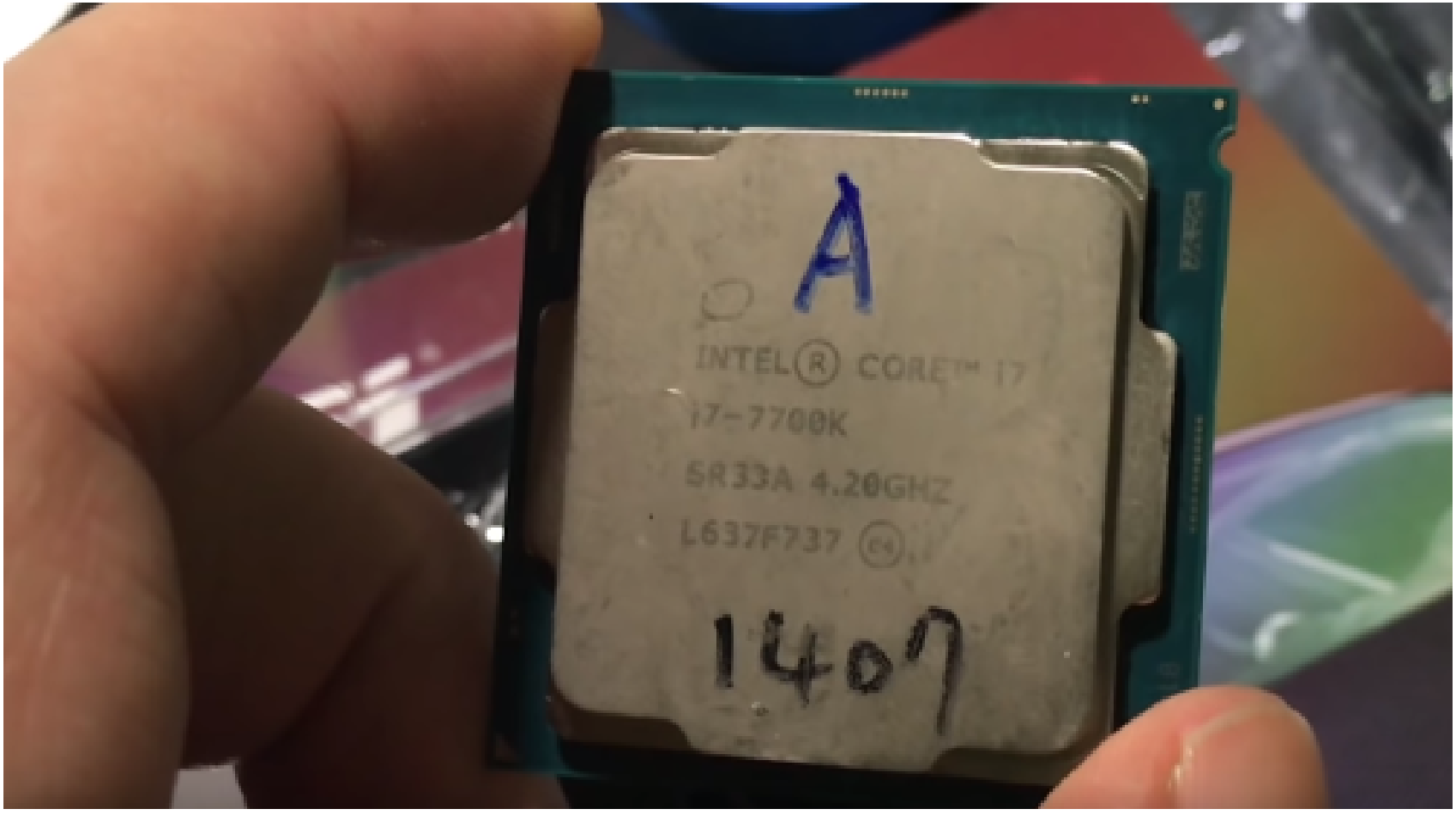
Possibili soluzioni

1. Calcolo parallelo (in alcuni casi)
2. Aumentare la RAM
3. Software efficiente
4. a volte, per posta (sì, sul serio!)

# Ad esempio

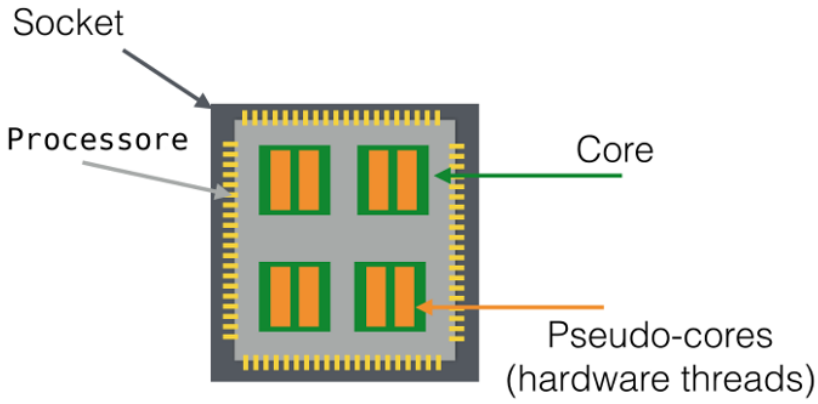
```
lscpu

Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   158
Model name:              Intel(R) Core(TM) i7-7700HQ
Stepping:                9
CPU MHz:                 2800.000
```

# Quanto è grande un processore?



# Quindi, su un computer normale

- 1 *socket*
- 4 core (quad-core)
- 2.8 Ghz (clock-speed)
- 2 threads per ogni core
- **8** CPU = numero di operazioni in simultanea




```
parallel::detectCores()

8
```

In teoria, potremmo avere un codice 8 volte più rapido.
In pratica, non è così semplice!

# Limiti

- Alcune operazioni *non possono* essere eseguite in parallelo!
- Diversi algoritmi sono **sequenziali** (ad esempio?)
- In casi realistici, solo una frazione $f$ del codice totale può essere eseguita **simultaneamente**
- In alcuni casi $f \approx 1$: ad esempio, somma delle colonne di una matrice

# Limiti

$$\text{guadagno} = \frac{\text{tempo base}}{\text{tempo parallelo}}$$

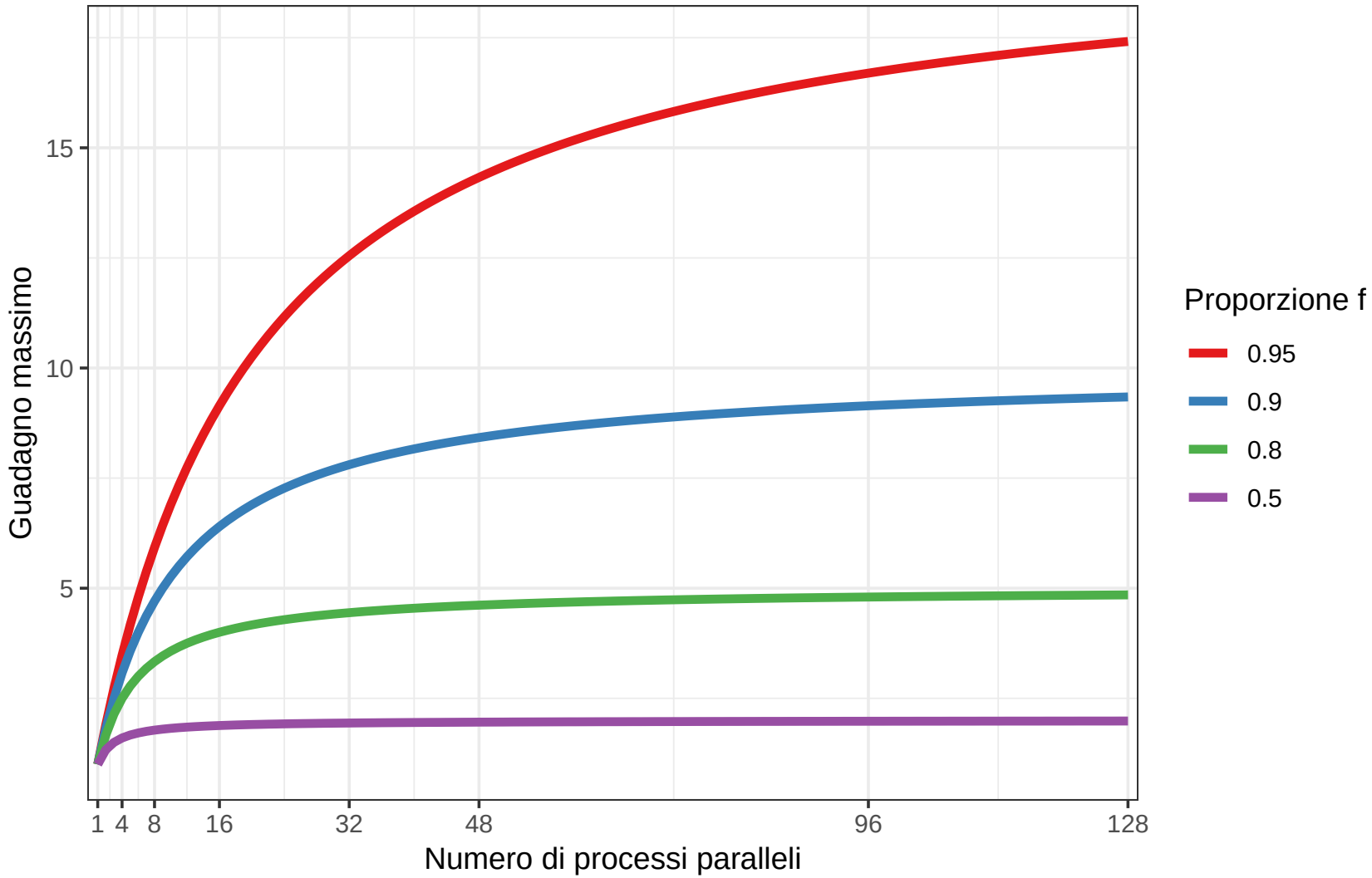
- $f$ proporzione di carico computazionale che può essere eseguita simultaneamente
- $k$ numero di processi eseguiti in parallelo

$$\text{guadagno} \leq \frac{1}{f/k + (1-f)}$$

- Quindi, per $k \to \infty$

$$\text{guadagno} \leq \frac{1}{1-f}$$

# Legge di Ahmdahl



# Messaggio

1. Se $f < 1$ l'efficienza massima è **limitata**.
   - Ad esempio, con $f = 0.95$ il guadagno massimo è 20.

2. Il limite corrisponde ad un concetto **teorico** ed **ideale**. Nella pratica, rappresenta spesso una stima molto ottimistica
   - Anche nel caso $f \approx 1$, vi sono alcuni ostacoli che bisogna tenere in considerazione

# Calcolo parallelo in pratica - ostacoli

1. **communication overhead**
   - trasferimento tra processi. Ad esempio di dati, sia in input che in output. Oppure codice o librerie
   - guadagno divisione del lavoro $\ll$ costo

2. **load balance**
   - assegnazione efficiente dei lavori ai diversi processori.
   - non sempre un'assegnazione "democratica" è la più efficiente.
   - Alcuni compiti potrebbero terminare prima, lasciando delle risorse inutilizzate

# Approcci alla parallelizzazione

1. A livello di **dati**
   - Divisone in *chunk*
   - Carico ridotto per ogni processore
2. A livello di **algoritmo**
   - Copia di *tutti i dati*
   - Parte diversa (ed indipendente) di algoritmo

} Esplicito

3. A livello di **operazioni**
   - Librerie di algebra lineare
   - Operazioni base in parallelo

} Implicito

# A livello di dati

- Consideriamo, come esempio illustrativo, un modello lineare con $p = 1$, $n$ osservazioni (supponiamo $n$ molto grande)
- $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$

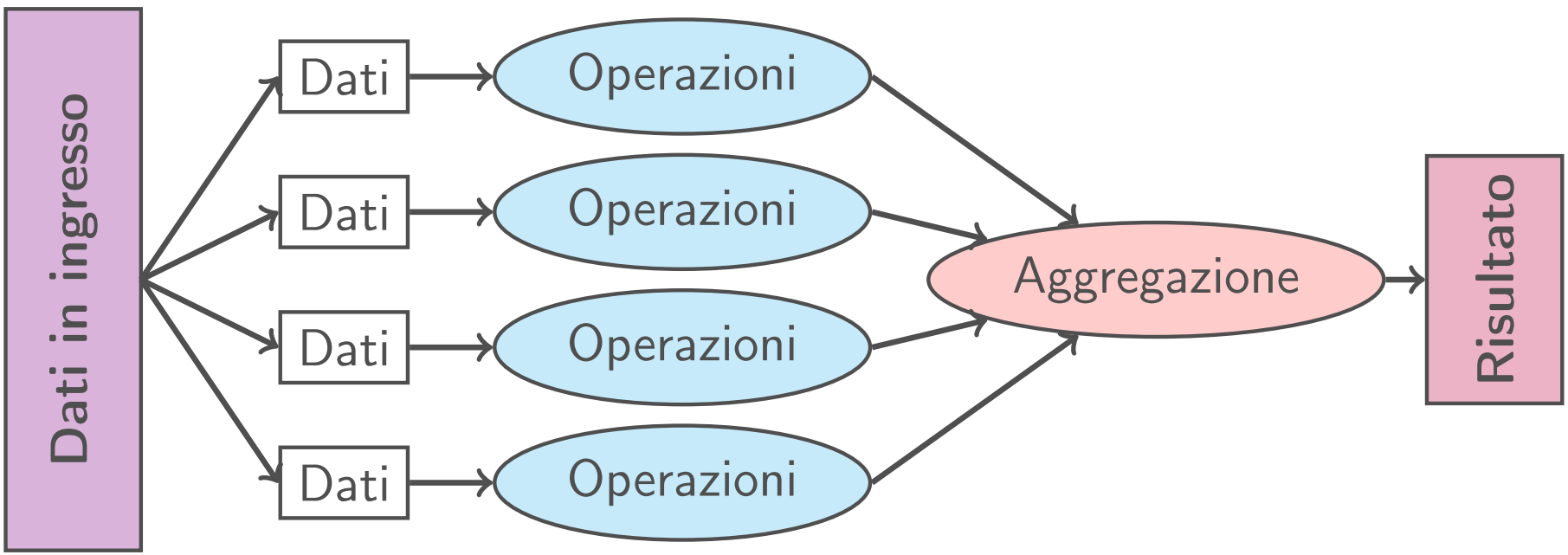
$$\hat{\beta}_1 = \frac{\text{cov}(x, y)}{\text{var}(x)} \quad \hat{\beta}_0 = M_y - \hat{\beta}_1 M_x$$

  dove $M_x = \frac{1}{n} \sum_{i=1}^{n} x_i$ ed $M_y = \frac{1}{n} \sum_{i=1}^{n} y_i$
- $k + 1$ processori
- Dividiamo quindi i dati in $k$ *chunk* di dimensione $n_k = n/k$

$$x = (x^{(1)}, x^{(2)}, \ldots, x^{(k)})^T \quad y = (y^{(1)}, y^{(2)}, \ldots, y^{(k)})^T$$

# A livello di dati - idea generale

1. Ogni processore $j$ riceve la coppia di vettori $(x^{(j)}, y^{(j)})$, $j = 1, \ldots, k$
2. Esegue alcune operazioni
3. Trasmette i risultati ad un processore centrale, che provvede ad aggregarli in modo opportuno (il processore $(k+1)$-esimo)




- Approccio "Map-Reduce"

# A livello di dati - implementazione

- Le stime ai minimi quadrati sono funzione delle seguenti quantità (globali)
  - $M_x$
  - $M_y$
  - $\text{cov}(x, y)$
  - $\text{var}(x)$
- Vogliamo ottenerle calcolando alcune parti separatamente
- Ricordiamo che la covarianza può essere espressa come

$$\text{cov}(x, y) = M_{xy} - M_x M_y$$

dove $M_{xy} = \frac{1}{n} \sum_{i=1}^{n} x_i y_i$

# A livello di dati - $\mathrm{cov}(x, y)$

- Calcolo delle medie parziali dei *chunk*. Per $j = 1, \ldots, k$

$$M_x^{(j)} = \frac{1}{n_j} \sum_{i=1}^{n_j} x_i^{(j)} \qquad M_y^{(j)} = \frac{1}{n_j} \sum_{i=1}^{n_j} y_i^{(j)}$$

- Media = media delle medie
- Prodotti misti nei *chunk*. Per $j = 1, \ldots, k$

$$M_{xy}^{(j)} = \frac{1}{n_j} \sum_{i=1}^{n_j} x_i^{(j)} y_i^{(j)}$$

- Aggregazione

$$\mathrm{cov}(x, y) = \frac{1}{k} \sum_{j=1}^{k} M_{xy}^{(j)} - \underbrace{\left( \frac{1}{k} \sum_{j=1}^{k} M_x^{(j)} \right)}_{M_x} \underbrace{\left( \frac{1}{k} \sum_{j=1}^{k} M_y^{(j)} \right)}_{M_y}$$

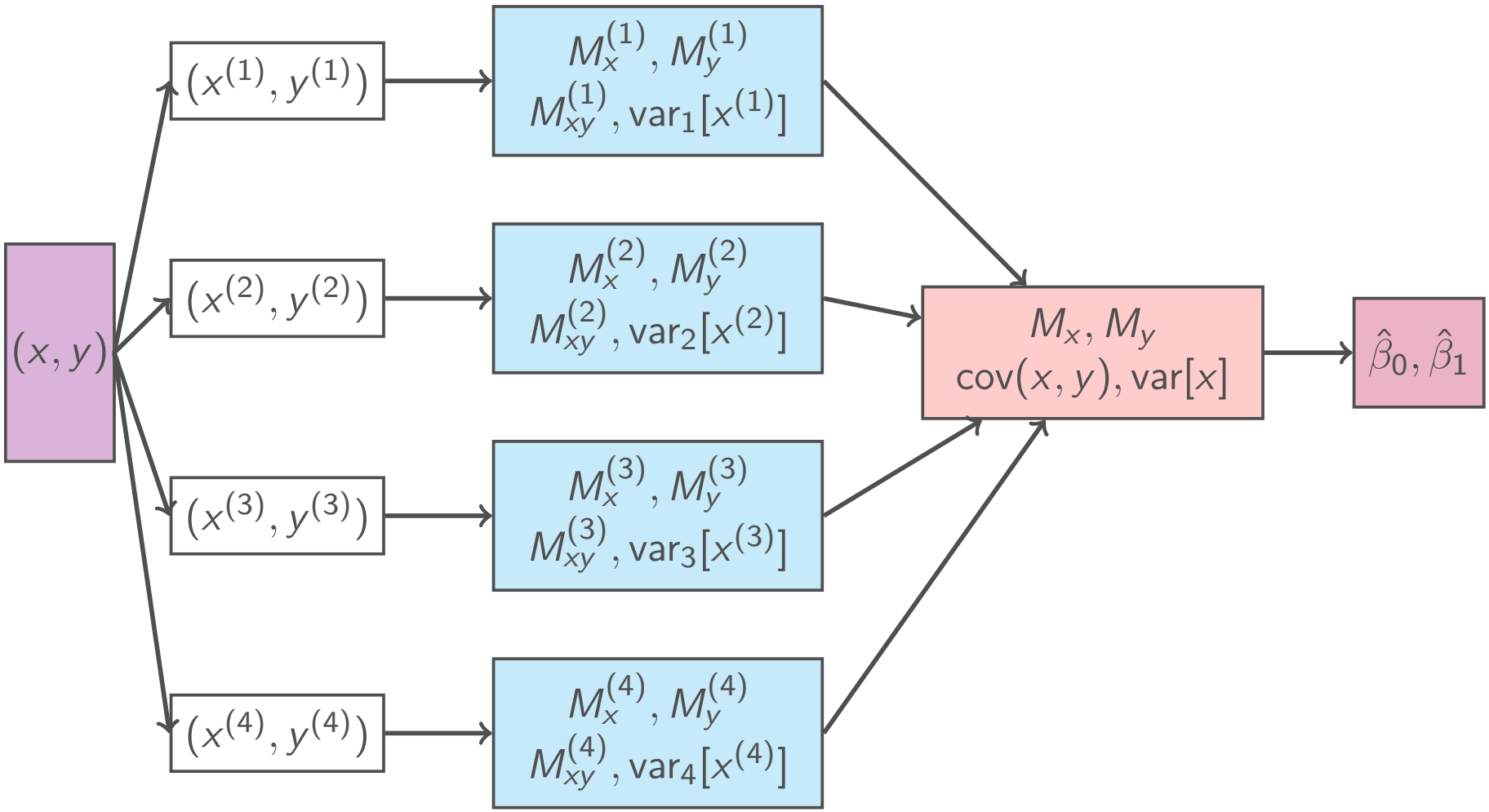
# A livello di dati - var($x$)

- Spesso è disponibile un'implementazione di `var` efficiente
- Per $j = 1, \ldots, k$, calcolo varianza di $x^{(j)}$ nel *chunk*: $\text{var}_j[x^{(j)}]$

$$\text{var}[x] = \mathbb{E}[\text{var}[x \mid z]] + \text{var}[\mathbb{E}[x \mid z]]$$

$$\text{var}[x] = \frac{1}{n}\left(n_j \sum_{j=1}^{k} \text{var}_j[x^{(j)}] + k n_j \left[\text{var}[(M_x^{(1)}, \ldots, M_x^{(k)}]\right]\right)$$

# Map-Reduce per il modello lineare



# Commenti

- La divisione in *chunk* riduce l'onere computazionale di ogni processore
- Molto utile anche per le statistiche descrittive (media, varianze, massimo, minimo..)
- Diversi algoritmi complessi ammettono scomposizioni in *chunk*

- Non sempre è applicabile
- "Stimatori" locali $\rightarrow$ globali

# Approcci alla parallelizzazione

1. A livello di **dati**
   - Divisone in *chunk*
   - Carico ridotto per ogni processore
2. A livello di **algoritmo**
   - Copia di *tutti i dati*
   - Parte diversa (ed indipendente) di algoritmo

} Esplicito

3. A livello di **operazioni**
   - Librerie di algebra lineare
   - Operazioni base in parallelo

} Implicito

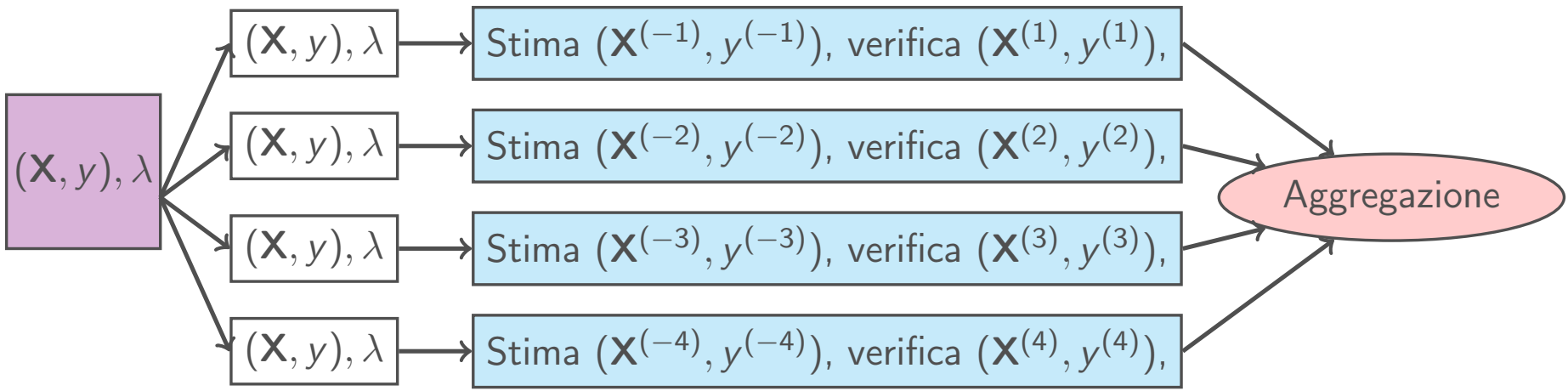
# A livello di algoritmo

- Rispetto all'**algoritmo**
- Assegnando ad ogni processore **tutti** i dati
- Valutando una parte diversa di algoritmo (ed indipendente dalle altre), ed aggregando i risultati

- Ad esempio, nel caso della regolazione di modelli

# A livello di algoritmo - convalida incrociata

- $k = 4$ processori, CV a 4-fold.



# A livello di algoritmo - convalida incrociata

- L'intero dataset viene copiato in ogni core
- Rischio di *communication overhead*!
- In generale si assegnano direttamente diversi valori di $\lambda$

- Utile per operazioni che in single core non sono particolarmente onerose
- Ma possono richiedere molto tempo se eseguite sequenzialmente (ad esempio, griglia di valori)

# Approcci alla parallelizzazione

1. A livello di **dati**
   - Divisone in *chunk*
   - Carico ridotto per ogni processore
2. A livello di **algoritmo**
   - Copia di *tutti i dati*
   - Parte diversa (ed indipendente) di algoritmo

} Esplicito

3. A livello di **operazioni**
   - Librerie di algebra lineare
   - Operazioni base in parallelo

} Implicito

# A livello di operazioni

- Le operazioni algebriche (prodotto matriciale, inversione, scomposizioni...) hanno un ruolo cruciale nell'analisi dei dati

- Ad esempio

$$\hat{\beta} = (X^\intercal X)^{-1} X^\intercal y$$

$$\mathbb{E}[\beta_t \mid y^t] = G_t m_{t-1} + R_t X^\intercal (V_t + X_t R_t X_t^\intercal)^{-1} (y_t - X_t G_t m_{t-1})$$

- Diverse operazioni matriciali possono essere "naturalmente" eseguite in parallelo

# BLAS e LAPACK

- Spoiler: R, Python, Matlab non sanno fare operazioni matriciali!
- Funzioni come `solve(X)`, `numpy.linalg.inv(X)`, `inv(X)` richiamano delle *routine* di livello più basso
- **BLAS**: Basic Linear Algebra Subprograms
  - *"standard low-level routines for linear algebra libraries"*
  - operazioni fondamentali: somma e prodotto tra matrici
  - Sviluppata in Fortran dal 1979 (v 3.8.0 del 12/11/2017)
- **LAPACK**: Linear Algebra PACKage
  - utilizza BLAS
  - sistemi, scomposizioni (QR, spettrale...)
- Tutti i sistemi operativi (Linux, iOS, Windows) includono queste librerie di default

# A livello di operazioni

- BLAS e LAPACK sono designate per architetture single-thread
- Naturale, vista la diffusione praticamente universale

- Esistono alcune alternative **multithread**
  - OpenBLAS
  - Intel MKL
  - cuBLAS (GPU)

```
top - 12:06:21 up  3:22,  1 user,  load average: 1.49, 0.86, 0.79
Tasks: 266 total,   2 running, 264 sleeping,   0 stopped,   0 zombie
%Cpu(s): 51.3 us,  2.2 sy,  0.0 ni, 46.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16144900 total, 12014820 free,  1985540 used,  2144540 buff/cache
KiB Swap: 15625212 total, 15624444 free,      768 used. 13581944 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5918 meme      20   0  693704 128648  20676 R 400.0  0.8   1:16.98 R
```

# Approcci alla parallelizzazione

1. A livello di **dati**
   - Divisone in *chunk*
   - Carico ridotto per ogni processore
2. A livello di **algoritmo**
   - Copia di *tutti i dati*
   - Parte diversa (ed indipendente) di algoritmo

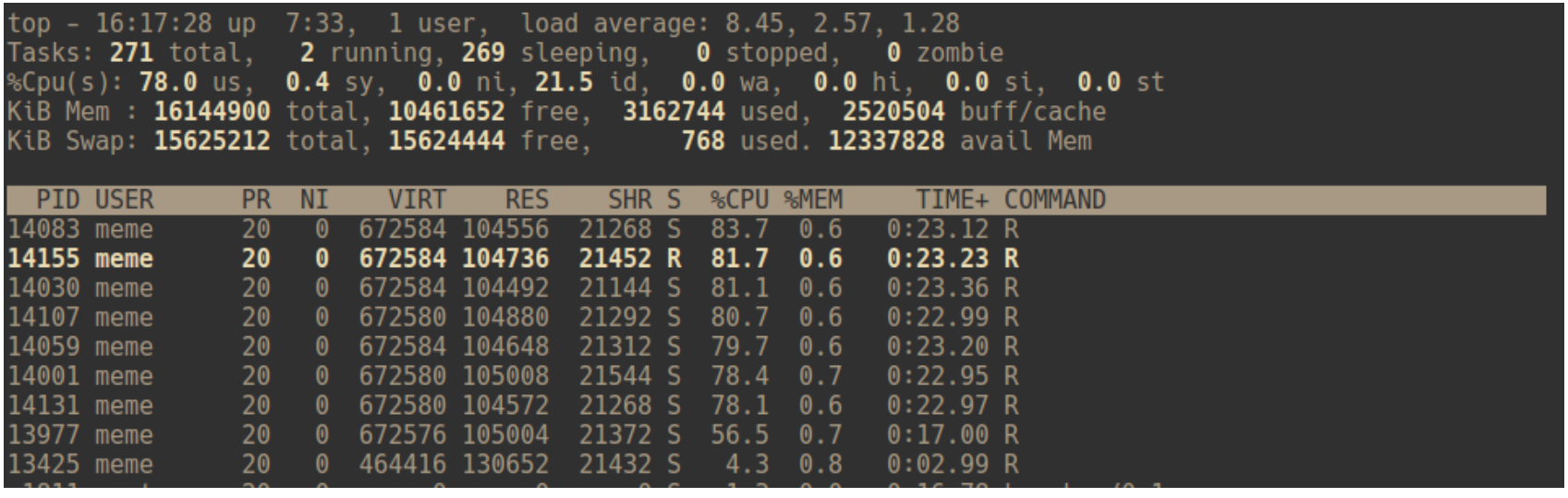
} Esplicito

3. A livello di **operazioni**
   - Librerie di algebra lineare
   - Operazioni base in parallelo

} Implicito

# Commenti generali

- Solo alcuni approcci potrebbero essere applicabili (o efficienti) per uno specifico problema
- Usare diversi approcci in contemporanea porta a risultati disastrosi!




- Molto importante capire cosa stiamo facendo

# Quindi, che facciamo?

- Pacchetti per il calcolo parallelo (`foreach`, `snofall`)
- Alcuni pacchetti R come `glmnet` e `boot` permettono di eseguire alcune operazioni in parallelo in modo automatico
- versioni di R / Python con librerie per il calcolo parallelo implicito
- Esistono ambienti appositamente sviluppati per il calcolo parallelo sui big data
- Divisione del lavoro tra processi (load-balance, communication overhead), ottimizzano I/O
- Uno di questi è **Apache Spark**

# Apache Spark - cosa ci serve sapere

- Piattaforma per l'analisi distribuita dei big data
- Basato su MapReduce (come filosofia)


- Gestione processi (Spark *core*)
- Gestione basi di dati (Spark SQL)
- Lettura / scrittura file (Spark Streaming)
- Modellistica (MLlib)

- Soprattutto, interfaccia per Python, R, Scala, Java, SQL
- progettato per funzionare su un numero qualsiasi di CPU, ma di fatto funziona bene se ne abbiamo molte (anche cluster)
- Ma il mio PC non ha molte CPU!

# Dove troviamo tante CPU?

- I PC comuni montano processori con un numero di CPU modesto (4-20)
- E ha senso che sia così (personal)!
- Se ne servono molte, possiamo appoggiarci a macchine *remote*
- Spesso hanno un numero di CPU elevato, e quantità di RAM importanti

# Perché il Cloud (Computing) - per le aziende


- Alcune aziende che noleggiano macchine cloud sono:
  - Google Cloud Platform (GCP)
  - Amazon Web Services (AWS)
  - Aruba

1. **Costi**: talvolta più contenuti
2. **Risorse**: dinamiche (RAM, CPU, storage). Pago ciò che uso
3. **Sicurezza**: standard elevatissimi (privacy / backup)
4. Raggiungibili da qualsiasi località

- Compagnie come Netflix, AirBnb e Adobe usano i servizi AWS

# Perché il Cloud Computing - per noi statistici

- Amministratore (root)
- Macchine Virtuali dinamiche on demand a prezzi accessibili

| Machine type | CPUs | Memoria | USD/h |
|---|---|---|---|
| f1-micro | 1 | 0.6 GB | gratis |
| n1-standard-4 | 4 | 15 GB | 0.1900 |
| n1-standard-64 | 64 | 240 GB | 3.0400 |
| n1-highmem-16 | 16 | 104 GB | 0.9472 |
| n1-highmem-32 | 32 | 208 GB | 1.8944 |
| n1-highcpu-16 | 16 | 14.40 GB | 0.5672 |
| n1-highcpu-64 | 64 | 57.6 GB | 2.2688 |
| n1-megamem-96 | 96 | 1433.6 GB | 10.6740 |

- Consulenza, ricerca, etc

# Osservazioni generali

- In modalità single core, PC comuni (e.g. ASID) sono spesso più potenti delle macchine cloud
- Il vantaggio di avere tanti processori a disposizione diventa rilevante nel momento in cui usiamo architetture in grado di sfruttarli in modo efficiente! (ad esempio, Spark)

- A volte gli algoritmi che usiamo non richiedono molte risorse, ma sono molto lenti poiché necessitano di molte iterazioni / replicazioni.
- Anche in questi casi, l'uso di macchine remote può essere utile, poiché sposta questo carico dal nostro PC a risorse dedicate