Laboratorio di Reti e Sistemi Distribuiti

15: Introduzione ai Sistemi Distribuiti

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 13th May 2025

Indice

Tassonomia di Flynn

Michael J. Flynn (Stanford) ha proposto un modello per classificare i sistemi (distribuiti e non) in base a:

- numero di flussi di istruzioni
- numero di flussi di dati

Questa classificazione è diventata uno standard nella progettazione e nello studio dell'elaborazione parallela e dell'architettura dei calcolatori.

- **SISD**: Single Instruction Single Data
- **SIMD**: Single Instruction Multiple Data
- **MIMD**: Multiple Instruction Multiple Data
- **MISD**: Multiple Instruction Single Data

Single Instruction Single Data (SISD)

Un solo flusso di istruzioni, un solo flusso di dati (cioè l'architettura classica dei vecchi processori seriali).

Esempio

Un processore che esegue una somma a + b, poi c + d, uno alla volta.

Implementazioni

CPU monoprocessore 80386, 80486, Pentium o semplici microcontrollore.

Single Instruction Multiple Data

- Un solo flusso di istruzioni, più flussi di dati
- L'istruzione è replicata su molti dati in parallelo.
- Usato in elaborazioni vettoriali o multimediali.

Esempio

GPU che usano CUDA o OpenCL (Open Computing Language)

```
1 float prezzi[4] = {10, 20, 30, 40};
2 float risultati[4];
3
4 for (int i = 0; i < 4; i++) {
5     risultati[i] = prezzi[i] * 2;
6 }
```

Il codice può essere o compilato in istruzioni SIMD (Es. C++/CUDA) o interpretato in istruzioni SIMD (Es. pytorch) ed eseguito in parallelo

Multiple Instruction Multiple Data

- Flussi multipli di istruzioni, flussi multipli di dati
- Ogni core può eseguire istruzioni diverse su dati diversi.
- È il tipo più flessibile e più comune nei sistemi multi-core.

Esempio

Un core calcola la somma di due array mentre un altro core esegue un algoritmo di sorting.

Implementazione

Sistemi multi-core (Intel i7, AMD Ryzen) Cluster HPC, supercomputer (MPI, OpenMP)

Multiple Instruction Single Data

- Flussi multipli di istruzioni, un solo flusso di dati
- È molto raro e usato in contesti fault-tolerant
- Stesso dato è processato da più istruzioni, spesso per ridondanza.

Esempio

Un singolo dato (es. sensore critico) viene processato da più algoritmi per controllo incrociato.

Implementazione

Sistemi critici nel dominio aerospaziale (es. sistemi di volo ridondanti)
Architetture come la Space Shuttle flight computer

Sotto-tassonomia MIMD

I sistemi MIMD (Multiple Instruction, Multiple Data) possono essere suddivisi in due categorie principali:

1. **Sistemi Multiprocessore (Cluster)**
   - Tutti i processori condividono la memoria.
   - Sono spesso chiamati SMP (Shared Memory Multiprocessors) o sistemi fortemente accoppiati (Tightly-Coupled).
   - Esempio: server multi-core con memoria RAM condivisa.
2. **Multi-computer**
   - Sistemi composti da più computer che non condividono la memoria ma comunicano tramite reti o switch.
   - Ogni nodo è un computer indipendente.
   - Includono anche i cosiddetti sistemi MPP (Massively Parallel Processors), con decine di migliaia di processori.

UMA vs NUMA

UMA – Uniform Memory Access

- Tutti i processori accedono alla memoria fisica condivisa con lo stesso tempo di accesso. Tale tempo non dipende dalla "distanza" tra cpu e memoria.
- Poco scalabile

NUMA – Non-Uniform Memory Access

- Ogni processore ha una memoria locale, ma può accedere anche alla memoria degli altri processori.
- Il tempo di accesso alla memoria varia: è più veloce se la memoria è locale, più lento se è remota (cioè appartenente a un altro nodo/CPU).
- Molto scalabile

Cluster

Definizione

Cluster: un'architettura distribuita costituita da più nodi (server o macchine fisiche/virtuali) coordinati per eseguire attività computazionali parallele, aumentare la disponibilità del sistema o garantire tolleranza ai guasti.

Un cluster è né UMA né NUMA nel senso tradizionale: ogni nodo ha memoria locale privata, separata dagli altri. I nodi non condividono una RAM centrale, quindi si parla di memoria distribuita piuttosto che condivisa.
Quindi, un cluster è **un'architettura a memoria distribuita**, dove ogni nodo è generalmente UMA o NUMA internamente, ma non nel complesso.
Se un sistema non è a memoria condivisa allora ricade nella categoria (vasta) dei sistemi **message passing**

Ray.io

Ray.io è un framework (middleware) open source per il calcolo distribuito e parallelo in Python. È progettato per essere scalabile, flessibile e con un curva di apprendimento poco ripida. È particolarmente usato in:

- Machine learning distribuito
- Ottimizzazione iperparametri
- Federated learning
- Reinforcement learning
- Servizi scalabili con microservizi (Ray Serve)

```
1   pip install "ray[default]"
```

Ray.io

```python
import ray
import time

ray.init()  # Avvia Ray in locale

@ray.remote
def worker_function(x):
    return x * x

# Esecuzione in parallelo
futures = [worker_function.remote(i) for i in range(4)]
results = ray.get(futures)  # Ottieni i risultati
print(results)  # Output: [0, 1, 4, 9]
```

Cos'è un Remote in Ray?

In Ray, una funzione decorata con @ray.remote diventa una funzione remota. Quando viene invocata tramite il metodo .remote(), non viene eseguita immediatamente; invece, viene creato un task che verrà eseguito su un worker disponibile nel cluster. La chiamata a .remote() restituisce un ObjectRef, che è un identificatore univoco del risultato del task. Per ottenere il risultato, è necessario chiamare ray.get() su questo ObjectRef.

Flusso interno di un Remote

- **Decorazione:** La funzione viene decorata con @ray.remote, trasformandola in una funzione remota.
- **Invocazione:** Chiamando .remote() sulla funzione, Ray crea un task che rappresenta l'esecuzione della funzione.
- **Schedulazione:** Il task viene inviato al Raylet, il componente di Ray responsabile della gestione dei worker.
- **Esecuzione:** Il Raylet assegna il task a un worker disponibile che esegue la funzione remota.
- **Ritorno del Risultato:** Il risultato dell'esecuzione viene restituito come un ObjectRef.
- **Recupero del Risultato:** Chiamando ray.get() sull'ObjectRef, il driver riceve il risultato dell'esecuzione.

Esecuzione di Ray in locale

```
1 import ray
2 ray.init()
```

Questo avvia Ray su una singola macchina, ma sfrutta tutti i core disponibili in modo parallelo.

Ray clustering

Su una macchina **head**

```
ray start --head --port=6379
```

Sulle macchine **worker**:

```
ray start --address='192.168.1.100:6379'
```

Su una qualunque macchina del cluster:

```
import ray
ray.init(address='192.168.1.100:6379')
```