

Programmazione Multicore

Alessio Marini, 2122855

Appunti presi durante il corso di **Programmazione Multicore** nell'anno **2025/2026** del professore Daniele De sensi.

Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏.

Contatti:

🐙 [alem1105](#)

✉ marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. Parallel Computing	3
1.1. Type of Parallel Systems	5
2. Distributed memory programming with MPI	9
2.1. Communicators	10

1. Parallel Computing

Dal 1986 al 2003 le velocità dei microprocessori aumentava di un 50% all'anno, ovvero un 60x in 10 anni. Dal 2003 in poi però questo incremento ha iniziato a rallentare, ad esempio dal 2015 al 2017 c'è stato soltanto un 4% all'anno ovvero un 1.5x in 10 anni.

Ci sono motivi fisici dietro a questo fenomeno e per questo, invece di continuare a costruire processori più potenti abbiamo iniziato ad inserire più processori all'interno dello stesso circuito.

I programmi seriali ovviamente non sfruttano questi benefici e continuano a venir eseguiti in un singolo processore, anche se ovviamente significa che possiamo eseguirli in più istanze contemporaneamente. Sono quindi i programmatori che devono sapere come utilizzare queste tecnologie per scrivere programmi che le sfruttano.

Ci serve tutta questa efficienza?

Per la maggior parte dei programmi no ma esistono dei campi di ricerca che dove c'è bisogno di eseguire tantissime operazioni in poco tempo o comunque su tantissimi dati, ad esempio:

- LLMs
- Decoding the human genome
- Climate modeling
- Protein folding
- Drug discovery
- Energy research
- Fundamental physics

Il motivo fisico che abbiamo accennato prima del perché costruiamo questi sistemi paralleli è dovuto al fatto che le performance di un processore aumentano con l'aumentare della densità di transistor che ha, questo comporta alcune cose:

- Transistor più piccoli -> Processori più veloci
- Processori più veloci -> Aumenta il loro consumo energetico
- Consumo più alto -> Aumenta la temperatura del processore
- Temperatura alta -> Comportamenti del processore inaspettati

Quindi anche se alcuni programmi possiamo eseguirli in più istanze e aumentare la loro efficienza spesso non è la strada migliore e dobbiamo imparare a scrivere del codice che usa la parallelizzazione. Con il tempo sono stati scoperti dei «pattern» facilmente convertibili in codice parallelo ma spesso la strada migliore è quella di fare un passo indietro e ripensare un nuovo algoritmo. Non sempre saremo in grado di parallelizzare completamente il codice.

Esempio

Codice Seriale - Compute n values and add them together

```
1 sum = 0;
2 for (i = 0; i < n; i++) {
3     x = Compute_next_value( ... );
4     sum += x;
5 }
```

Codice Parallelo - abbiamo p cores dove $p \ll n$ e ogni core calcola la somma di $\frac{n}{p}$ valori

```
1 my_sum = 0;
2 my_first_i = ... ;
3 my_last_i = ... ;
4 for (my_i = my_first_i; my_i < my_last_i; my_i++) {
5     my_x = Compute_next_value( . . . );
6     my_sum += my_x;
7 }
```

Quindi ogni core usa delle variabili per memorizzare il suo primo e ultimo valore da sommare, in questo modo ogni core può eseguire del codice indipendentemente dagli altri core.

Ad esempio se abbiamo 8 core e 24 valori, ogni core somma 3 valori:

- 1, 4, 3 - 9, 2, 8 - 5, 1, 1 - 6, 2, 7 - 2, 5, 0 - 4, 1, 8 - 6, 5, 1 - 2, 3, 9

E quindi avremo come somme:

- 8, 19, 7, 15, 7, 13, 12, 14

Ci basta quindi sommare la somma di tutti i core e ottenere il risultato finale.

Però usando questa soluzione, nello step finale abbiamo che un core solo (il principale) effettua la somma dei risultati degli altri, mentre loro appunto non stanno facendo nulla.

Per risolvere questo possiamo ad esempio accoppiare il core 0 con il core 1 e fare in modo che il core 0 sommi al suo risultato quello del core 1, poi possiamo fare lo stesso lavoro con il 2 e il 3 ed avere quindi nel core 2 la somma di 2 e 3 ecc...

Ripetiamo tutto questo accoppiando 0-2, 4-6... e poi continuiamo a ripetere accoppiando 0-4...

Vediamo graficamente:

TODO: image

Se confrontiamo i due metodi:

TODO: image

Notiamo che con il primo metodo, quello a sinistra, se abbiamo 8 cores facciamo 7 somme aggiuntive, in generale $p - 1$ somme.

Con il secondo metodo se abbiamo 8 cores facciamo 3 somme aggiuntive (sono sempre 7 ma sono parallele e avvengono nello stesso momento), in generale abbiamo $\log_2(p)$ somme.

Quindi se ad esempio avessimo avuto $p = 1000$ con il primo metodo avremmo avuto 999 somme mentre con il secondo soltanto 10.

Come scriviamo codice parallelo?

- **Task Parallelism:** Dividere alcune task fra i cores. L'idea è quella di eseguire compiti diversi in parallelo.
- **Data Parallelism:** Partizionare i dati fra i cores, fargli risolvere operazioni simili e risolvere il problema raccogliendo i dati. In generale quando i cores svolgono la stessa operazione ma su pezzi di dati diversi.

Esempio

Devo valutare 300 esami da 15 domande ciascuno e ho 3 assistenti:

- Data Parallelism: Ogni assistente valuta 100 esami
- Task Parallelism:
 - L'assistente 1 valuta tutti gli esami ma soltanto le domande 1-5
 - L'assistente 2 valuta tutti gli esami ma soltanto le domande 6-10
 - L'assistente 3 valuta tutti gli esami ma soltanto le domande 11-15

L'esempio che abbiamo fatto precedentemente con le somme dei vari cores, è stato parallelizzato on Data o Task Parallelism?

TODO: image

Se ogni core può lavorare in modo indipendente dagli altri allora la scrittura del codice sarà molto simile a quella di un programma seriale. In generale dobbiamo coordinare i cores, questo perché:

- Communication: Ad esempio perché ogni core manda una somma parziale ad un altro
- Load Balancing: Nessun core deve svolgere troppo lavoro in più rispetto ad altri perché altrimenti qualche core dovrà aspettare che alcuni finiscano e questo significa perdita di risorse e potenza.
- Synchronization: Ogni core lavora al suo ritmo ma dobbiamo assicurarci che nessuno vada troppo avanti. Ad esempio se un core compila una lista dei file da comprimere ed i cores che comprimono partono troppo presto potrebbero perdersi qualche files.

Noi scriveremo codice esplicitamente parallelo usando 4 diverse estensioni delle API di C:

- Message-Passing Interface (MPI) [Library]
- Posix Threads (Pthreads) [Library]
- OpenMP [Library + Compiler]
- CUDA [Library + Compiler]

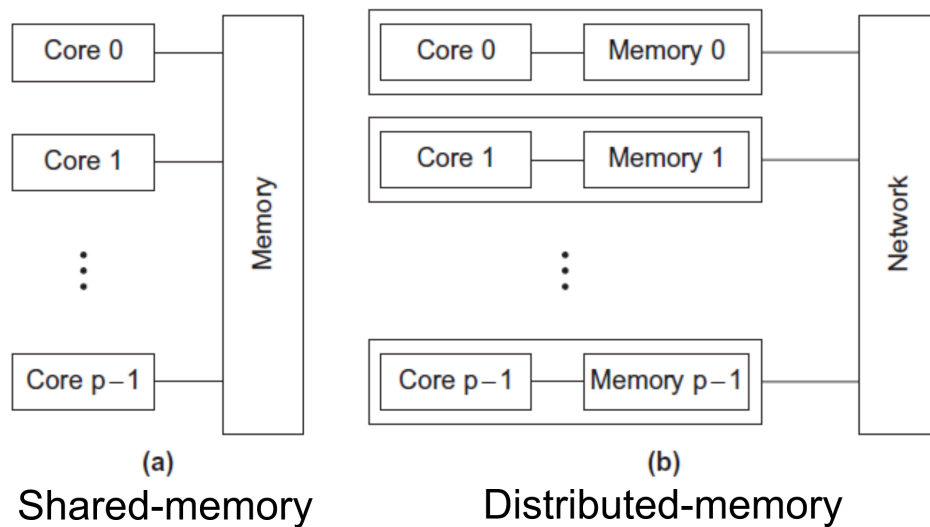
Useremo anche librerie ad alto livello già esistenti che però hanno un compresso per quanto riguarda facilità di utilizzo e performance.

1.1. Type of Parallel Systems

- **Shared Memory:** I core lavorano sulla stessa memoria e si coordinano leggendo una specifica zona di memoria.

- **Distributed Memory:** Ogni core ha la sua memoria dedicata e per coordinarsi si scambiano messaggi su una rete dedicata.

In generale quindi devono comunque avere un modo per coordinarsi.



- **Multiple-Instruction Multiple-Data (MIMD):** Ogni core ha la sua unità di controllo e può lavorare indipendentemente dagli altri. Come ad esempio la CPU classica dei PC, ogni core può fare qualcosa di diverso.
- **Single-Instruction Multiple-Data (SIMD):** Ogni core può lavorare su un pezzo di dato diverso, ma tutti devono lavorare per la stessa istruzione. Ad esempio possono lavorare tutti su un vettore ma ognuno su una parte di vettore diversa.

How will we write parallel programs?

	Shared Memory	Distributed Memory
SIMD	CUDA	
MIMD	Pthreads/ OpenMP/ CUDA	MPI

Concurrent

Diverse tasks possono essere in esecuzione in ogni momento. Questi possono essere anche seriali, ad esempio dei sistemi operativi su un singolo core.

Parallel

Diverse tasks possono **collaborare** per risolvere lo **stesso problema**.

Distributed

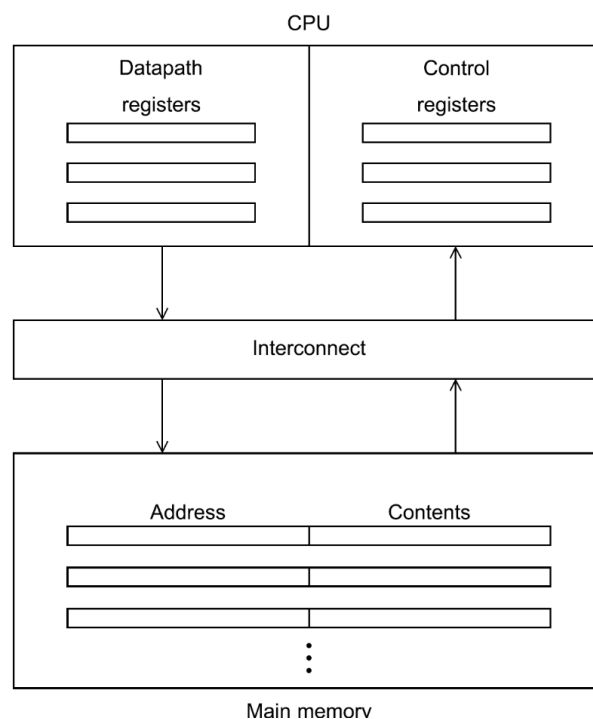
Il lavoro viene diviso su macchine separate che comunicano tramite una rete. Il sistema appare come sistema unico.

Da notare che i sistemi paralleli e distribuiti sono comunque concorrenti, infatti molte attività vengono svolte nello stesso momento e si contendono le risorse del sistema.

• Von Neumann architecture

Di solito quando programmiamo non ci preoccupiamo di come è fatto l'hardware su cui stiamo lavorando, di solito possiamo astrarre. Però se vogliamo scrivere codice efficiente ci tornerà utile capire come funziona l'hardware e scrivere codice ottimizzato per quell'hardware.

Vediamo la **Von Neumann Architecture**

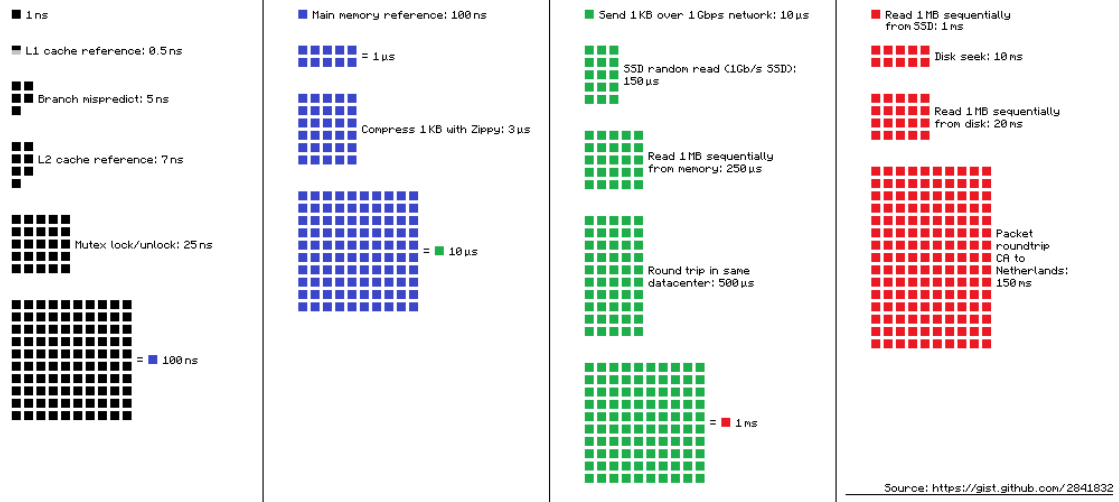


- **Main Memory:** È formata da tante locazioni di memoria che contengono dei dati, ciascuna è identificata da un **indirizzo**
- **CPU:** Esegue le istruzioni e decide quali eseguire. I registri sono delle memorie estremamente veloci ma anche molto piccole che servono a memorizzare dati importanti come lo stato d'esecuzione dei processi. Ad esempio uno dei più importanti è il **Program Counter (PC)** che contiene l'indirizzo della prossima istruzione da eseguire.
- **Interconnect:** Serve a mettere in comunicazione la memoria e la CPU, di solito è un **bus di sistema** ma in alcuni casi potrebbe essere più complesso.

Una macchina che segue il modello di Von Neumann esegue un'istruzione alla volta, ogni istruzione opera su una piccola parte di dati che vengono memorizzati nei registri. La CPU può leggere e scrivere dati nella memoria ma la separazione di queste due componenti causa quello che viene chiamato **Von Neumann Bottleneck**, infatti il bus di sistema o in generale la connessione fra i componenti determinate la velocità di trasferimento dei dati, di solito è più lenta sia della memoria che della CPU.

Una comparazione

Latency Numbers Every Programmer Should Know

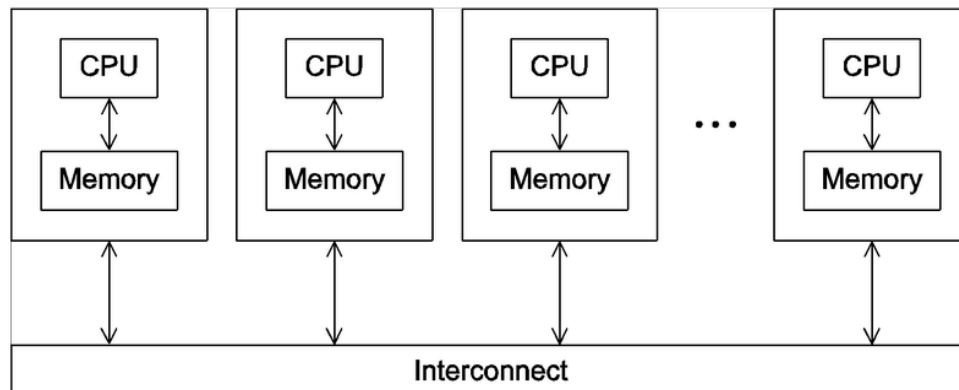


Source: <https://gist.github.com/2841832>

Assumendo 1GB/sec SSD

2. Distributed memory programming with MPI

Possiamo astrarre un **distributed memory system** in questo modo:



Single-Program Multiple-Data

Noi **compiliamo un solo programma** ma poi questo verrà eseguito da più processi. Useremo un **if-else** per cambiare il comportamento dei processi quindi, ad esempio:

- Se sono il processo 0 faccio X
- Altrimenti faccio Y

Ricordiamo che i processi non condividono memoria quindi la comunicazione avviene tramite **message passing**.

Per programmare con la libreria **MPI** abbiamo bisogno di aggiungere l'header `mpi.h` e usare degli identificatori, di solito iniziano con `MPI_`.

Esempio


```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(void) {
5     MPI_Init(NULL, NULL);
6     printf("hello, world\n");
7     MPI_Finalize();
8     return 0;
9 }
```

- `MPI_Init` inizializza `mpi` per tutto il necessario, possiamo passare dei puntatori ai parametri del main, se non ci servono possiamo anche passare `NULL`, come l'esempio sopra.
- `MPI_Finalize` fa capire a `mpi` che il programma è finito e può pulire tutta la memoria allocata.

Il programma non deve trovarsi tutto all'interno di questo costrutto ma è importante che ci sia la parte che vogliamo venga eseguita da più cores.

Compilazione


```
1 mpicc -g -Wall -o mpi_hello mpi_hello.c
```

 Bash

- `mpicc` è il compilatore
- `-g` fa visualizzare delle informazioni di debug
- `-Wall` attiva tutti gli avvisi
- `-o` specifica il nome del file di output

Esecuzione


```
1 mpiexec -n <number of processes> <executable>
```

 Bash

Serve per eseguire il programma e ci permette di specificare con quanti core lanciarlo.

Debugging

```
1 mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test
```

 Bash

Debuggere programmi non seriali è più complicato, ad esempio se eseguiamo lo stesso programma con un solo processo il problema potrebbe non verificarsi mentre se lo eseguiamo con un certo numero di core sì.

Con il comando sopra possiamo lanciare 5 processi, il quinto verrà eseguito con il debugger, che si chiama `ddd`.

2.1. Communicators

Sono dei processi che possono mandarsi dei messaggi fra di loro, quando chiamiamo `MPI_Init` viene inizializzato anche un comunicatore per tutti i processi, questo è chiamato `MPI_COMM_WORLD`.

Dopo un `INIT` possiamo «catturare» l'identificatore di ogni processo e anche il numero totale, attraverso:

```
1 int MPI_Comm_size(  
2     MPI_Comm comm /* in */,  
3     int* comm_sz_p /* out */  
4 );
```

 C

```
1 int MPI_Comm_rank(  
2     MPI_Comm comm /* in */,  
3     int* my_rank_p /* out */  
4 );
```

 C

Prendono in input il comunicatore e un puntatore ad intero e salvano in questi interi i valori di, rispettivamente, quanti processi ci sono e il grado del processo attuale.

Esempio di utilizzo

```

1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(void) {
5      int comm_sz, my_rank;
6      MPI_Init(NULL, NULL);
7      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
8      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9      printf("hello, world from process %d out of %d\n", my_rank, comm_sz);
10     MPI_Finalize();
11     return 0;
12 }

```

Se eseguiamo questo programma noteremo che ad ogni esecuzione l'ordine dei print cambia, questo perchè non sappiamo con esattezza quale processo finirà per primo quindi otterremo sempre un ordine diverso. Possiamo cambiare questo comportamento facendo comunicare i processi.

Per far comunicare i processi utilizziamo due funzioni `MPI_Send` e `MPI_Recv`:

```

1  int MPI_Send(
2      void*          msg_buf_p    /* in */,
3      int            msg_size     /* in */,
4      MPI_Datatype   msg_type     /* in */,
5      int            dest         /* in */,
6      int            tag          /* in */,
7      MPI_Comm       communicator /* in */
8  );
9
10 int MPI_Recv(
11     void*          msg_buf_p    /* out */,
12     int            buf_size     /* in */,
13     MPI_Datatype   buf_type     /* in */,
14     int            source       /* in */,
15     int            tag          /* in */,
16     MPI_Comm       communicator /* in */,
17     MPI_Status*    status_p     /* in */
18 );

```

Da notare che `msg_size` e `buf_size` indicano il numero di elementi e non il numero di byte.

Il campo `status_p` di tipo `MPI_Status*` è un puntatore ad una struttura composta in questo modo:

```

1  typedef struct MPI_Status {
2      int MPI_SOURCE; // rank del mittente
3      int MPI_TAG;    // tag del messaggio
4      int MPI_ERROR;  // eventuale codice d'errore
5  } MPI_Status;

```

Serve a ricavare dei dati dal messaggio appena ricevuto, tramite una funzione aggiuntiva chiamata `MPI_Get_count` definita:

```
1 int MPI_Get_count(  
2     MPI_Status* status_p /* in */,  
3     MPI_Datatype type /* in */,  
4     int* count_p /* out */  
5 );
```

Possiamo capire quanti elementi stiamo ricevendo.

Proviamo adesso a scrivere una nuova versione di `hello world` che rispetta l'ordine delle stampa, facendo in modo che un processo padre riceva gli output dei processi figli.

È importante anche sapere che in MPI si usano dei tipi di dato specifici e non quelli classici di C:

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

```
1 #include <stdio.h>  
2 #include <string.h>  
3 #include <mpi.h>  
4  
5 const int MAX_STRING = 100;  
6  
7 int main(void) {  
8     char greeting[MAX_STRING];  
9     int comm_sz;  
10    int my_rank;  
11  
12    MPI_INIT(NULL, NULL);  
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
15  
16    if (my_rank != 0) {  
17        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
```

```

18     MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
19 } else {
20     printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
21     for (int q = 1; q < comm_sz; q++) {
22         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD,
23                 MPI_STATUS_IGNORE);
24         printf("%s\n", greeting);
25     }
26 }
27 MPI_Finalize();
28 return 0;
29 }

```

Sending Order

MPI garantisce che i messaggi che si inviano una coppia di processi arrivino nell'ordine di invio, quindi se ad esempio il processo p1 invia a p2 3 messaggi m1, m2 ed m3 questi arriveranno sempre nell'ordine di invio. Questo ordine però non è garantito quando ci sono mittenti diversi:

- Un processo p1 invia il messaggio m1 a p0 al tempo t0
- Un processo p2 invia il messaggio m2 a p0 al tempo t1

Ci aspettiamo che arrivi prima il messaggio m1 ma potrebbe non accadere sempre.

Oltre al comunicatore visto prima `MPI_COMM_WORLD` possiamo anche crearne di nostri per avere più libertà ed implementare sistemi più complessi.

Quindi per fare comunicare un insieme di processi possiamo o creare un comunicatore per ogni gruppo oppure usare i **tag**.

Message Matching

Prendiamo come esempio due processi che hanno rank `q` ed `r` che si scambiano dei messaggi:

```
1 // rank q
2 MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);
3
4 // rank r
5 MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm,
  &status);
```

È importante che i campi `dest` e `src` corrispondano ovvero:

- In `dest` dobbiamo avere il rank di `r` e su `src` il rank di `q`.
- `send_comm`, `recv_comm` devono essere lo stesso comunicatore.
- `send_tag`, `recv_tag` devono essere lo stesso tag.

Oltre a queste condizioni un messaggio può essere ricevuto correttamente se:

- `recv_type` = `send_type` - Quindi i tipi devono combaciare
- `recv_buf_sz` \geq `send_buf_sz` - Il buffer del ricevente deve essere grande abbastanza da contenere tutto il buffer del mittente.

Un processo può ricevere un messaggio senza conoscere:

- Quanti dati riceve.
- Il mittente (`MPI_ANY_SOURCE`)
- Il tag del messaggio (`MPI_ANY_TAG`)

Il comando `MPI_Send` può comportarsi in modo diverso a seconda di:

- Dimensione del messaggio
- Disponibilità del ricevente
- Risorse del sistema

Per un `MPI_Send` ci sono due diversi protocolli di implementazione:

- **Eager Protocol:** Viene usato per messaggi piccoli, MPI copia i dati in un buffer interno e li spedisce immediatamente anche se il ricevente non ha ancora effettuato una receive. In questo caso restituisce subito un valore e quindi non è bloccante, finché il destinatario non riceve il messaggio questo vivrà nel buffer interno.
- **Rendezvous Protocol:** Si usa per messaggi grandi, il mittente non invia subito i dati ma manda prima una **request to send** e se il destinatario risponde in modo positivo allora i dati vengono inviati. In questo caso la send è bloccante, fino a quando il ricevente non è pronto.

Ogni implementazione decide la soglia per usare questi protocolli, in OpenMPI abbiamo:

- `< 8 KB` Eager
- `≥ 8 KB` Rendezvous

La receive però a differenza della send blocca sempre il programma in attesa di ricevere qualcosa, a meno che non si utilizzi una versione non bloccante come `MPI_Irecv`.

È importante capire però che anche se oggi un programma che scriviamo funziona perchè la send ritorna subito con messaggi piccoli non dobbiamo scrivere codice che si basa su questo comportamento perchè se lo portiamo su altre macchine con diverse implementazioni potremmo ottenere dei comportamenti diversi.

Per scrivere buon codice bisogna pensare alle send come se fossero sempre bloccanti.

Attenzione!

1. Se un processo prova a ricevere un messaggio che però non combacia con nessuno di quelli inviati, il processo rimarrà bloccato per sempre.
2. Se una send è bloccante e nessuno la riceve allora anche il processo mittente rimarrà bloccato.
3. Se la send è bufferizzata come ad esempio nel protocollo eager e nessuno riceve i dati, allora i dati nel buffer andranno persi. O perchè il programma termina o perchè in generale non verranno mai prelevati dalla memoria.
4. Se il rank del processo destinatario è uguale a quello del mittente il processo potrebbe bloccarsi, o peggio, abbinarsi ad un messaggio sbagliato. (Questo comportamento può essere usato per inviare messaggi a se stessi ma va usato con attenzione perchè potrebbe portare a dei deadlock.)