

Esercitazioni di Sistemi di Calcolo Parallelo

Paolo Avogadro

DISCo, Università degli Studi di Milano-Bicocca

U14, Id&aLab T36

paolo.avogadro@unimib.it

a.a. 2019-2020

Aula Lezione T014 , edificio U14

per motivi contingenti: WEBEX

- Mercoledì' 14:30-16:30 Numero Sessione 121 762 2323
- Giovedì' 10:30-12:30 Numero Sessione 121 187 9238



Struttura del Corso

Per ogni argomento verranno presentati degli elementi essenziali di teoria, e verranno proposti degli esercizi. Per questo, e' molto utile avere a disposizione dei computer con cui poter fare gli esercizi, ed interfacciarsi con il server.

Numero di lezioni (ore) previste (± 2 per argomento):

- MPI 8 lezioni
- openMP 3 lezioni
- CUDA 6 lezioni
- Hadoop 3 lezioni

Finalita' delle esercitazioni:

- prendere contatto con il modo di pensare associato al calcolo parallelo
- essere in grado di riconoscere le problematiche di un codice
- scrivere codici base
- essere in grado di ottimizzare un codice parallelo

Esame

L'esame consiste di 2 parti

- un seminario su un argomento di calcolo parallelo da concordare con il Prof. Dominoni
- un elaborato in cui si dimostra la propria capacita' di utilizzare le tecnologie proposte nel corso (da concordare con il prof. Dominoni)

Consigli sull'elaborato:

Lo scopo dell'elaborato e' di dimostrare che:

- conoscete le tecnologie
- conoscete la teoria di base per misurare i risultati ottenuti
- eseguite i codici in varie condizioni di utilizzo (variando il numero di processi/thread, la struttura del kernel, le memorie usate...)
- siete in grado di fare delle ottimizzazioni di base

Attenzione:

- **NON** e' necessario produrre il codice piu' veloce mai visto per un particolare algoritmo (pero' se va veloce e' bene...)
- **NON** e' consigliabile scegliere un algoritmo particolarmente difficile da parallelizzare
- **NON** e' necessario costruire tutto il codice da zero: ricordate che lo scopo di questo corso e' avvicinarvi al calcolo parallelo, non alla programmazione! Voi dovete dimostrare di sapere parallelizzare/ costruire un codice parallelo, non dovete dimostrarci come fate i/o, ...
- Durante il corso vi verra' fatto vedere il prodotto tra matrici con tutte le tecnologie presentate ... si suppone che voi facciate qualcosa di diverso

Introduzione a MPI

MPI = Message Passing Interface.

Cos'è?

- È una specifica, non un'implementazione. Esistono varie implementazioni, per esempio **openMPI** (che prenderemo come riferimento), **MPICH** (comunque presente sul server), ecc.
- MPI serve (come fa intuire il nome) per gestire lo scambio di messaggi tra vari “centri di calcolo”.
- diverse implementazioni presentano lievi differenze, che però possono creare **grossi** problemi, ovvero lo stesso codice con due implementazioni differenti può non funzionare.
- I codici MPI possono essere del tipo SPMD: Single Program Multiple Data (noi ci concentreremo su questo paradigma)
- è possibile anche lanciare codici MPMD: Multiple Program Multiple Data (non trattato in questo corso)

Queste dispense sono liberamente ispirate a:

- diapositive del CINECA sul calcolo parallelo:
<http://www.hpc.cineca.it/content/training>
- Tutorial di MPI: <https://computing.llnl.gov/tutorials/mpi/>
- Corsi della Cornell: <https://cvw.cac.cornell.edu/topics>

MPI in pratica

- Può essere utile pensare ai computer su cui all'inizio sono stati fatti girare codici MPI, ovvero grosse macchine con molte cpu, e ognuna delle cpu con una propria memoria. In pratica era come avere molti computer connessi tra loro, e quindi serviva un protocollo per scambiare messaggi in modo "sicuro": MPI.
- Quando viene scritto ed eseguito un codice MPI, lo **stesso** codice viene mandato a **TUTTI** i computer di una "rete" (il termine computer è molto generico, esempio possono essere i vari core di un laptop, o computer veri e propri di una rete locale, i nodi di un supercomputer ...).
- In qualche modo ha senso pensare che ognuno dei "computer" coinvolti abbia a disposizione la propria memoria, ed esegua una istanza del codice: questa istanza verrà riferita come **processo/process**.
- le funzioni di MPI si "preoccupano" di fare comunicare correttamente ed **efficacemente** i **processi** (tramite algoritmi appositi la comunicazione viene **ottimizzata**).
- **Attenzione**: tante più sono le unità di calcolo coinvolte, tanto più la comunicazione può diventare pesante nell'esecuzione del calcolo.

MPI dettagli

Lo standard MPI e' nato nel 1991 da un gruppo di ricercatori, da allora ci sono state molte revisioni e innovazioni. In quel periodo il problema era principalmente legato ai supercalcolatori.

Oggi, i computer stanno raggiungendo i limiti fisici per quanto riguarda le frequenze di clock e la potenza dissipata.

In **tutti** gli ambiti (anche sui computer di casa), si tende a creare sistemi in cui ci sono molte unita' di processamento, questo perche' molti tipi di problemi possono essere "spezzettati" su varie macchine o "parallelizzati".

ATTENZIONE: Non tutti i tipi di calcolo sono parallelizzabili (e' molto piu' facile parallelizzare il calcolo della media che quello della mediana).

MPI caratteristiche varie:

- gestione dei processi (definizione, identificazione di gruppi o singoli processi)
- funzioni per lo scambio di messaggi.
- nuovi tipi di dato e macro che aiutino i programmatori.
- il risultato di un calcolo MPI e' (in genere) **NON DETERMINISTICO** (ma noi lo vogliamo deterministico)
- La versione attuale di MPI e' la 3.1 (a). Dalla versione 3.0 sono state introdotte le funzioni **non blocking** (spiegate poi nel corso)
- MPI e' compatibile con FORTRAN, C e C++. Oltre ai comandi del linguaggio usato vengono aggiunte delle funzioni dello standard MPI (e altro). In questo corso ci limiteremo ad esempi di C/C++.
- in ogni versione di MPI vengono definite delle nuove funzioni/subroutine, e il loro effetto (in genere legate allo scambio di informazioni tra diversi processi).
- "Spannometricamente" parlando, scrivere un codice parallelo con MPI e' un ordine di grandezza piu' complesso che scrivere il medesimo codice che gira su un singolo processore.

In teoria sono **sufficienti** (ma non **super efficienti**) 6 funzioni MPI di base, per scrivere un codice parallelo. Queste funzioni erano gia' presenti nelle prime versioni di MPI. Con l'avvento di MPI 3.0 ci sono piu' di 430 funzioni (vedi

http://mpi.deino.net/mpi_functions/) che servono ad aiutare la comunicazione tra processi e la scrittura del codice.

le funzioni MPI

In C (ricordiamo che e' caSE sensitive) la notazione standard per tutte le funzioni MPI e' la seguente:

`MPI_Xxxx` (i.e. `MPI_Barrier`)

- nota che la prima X (lettera qualsiasi) e' maiuscola (le altre possono essere minuscole ma non e' obblgatorio).
- il numero di lettere per identificare una funzione non e' solo 4....
- Tutte le routine MPI che non funzionano dovrebbero abortire, questo pero' e' poco utile al programmatore, esiste quindi un modo per fare resituire un valore di errore (che pero' in generale dipende dall'implementazione).
- Se l'esecuzione e' avvenuta senza problemi si ottiene `MPI_SUCCESS` come risultato della funzione.

Un concetto fondamentale: Communicator



Figure: rappresentazione grafica del “mondo comune”.

Il *Communicator* e' una delle strutture piu' importanti per MPI.
In un certo senso e' l'universo in cui vivono i vari processi e attraverso a cui comunicano.

Un *Communicator* e' una struttura che contiene uno o piu' gruppi di processi, un insieme di informazioni ad essi collegate (chiamato context) ed eventualmente la *topologia*. In un communicator ogni processo ha un unico numero identificativo, chiamato **rank** che va da 0 a $n - 1$ dove n e' la quantita' di processi presenti nel communicator (chiamata **size** del communicator).

Dimensione del communicator: $SIZE = 5$

Rank = 0



Rank = 1



Rank = 3



Rank = 2



Rank = 4



Communicator... continua

- un communicator puo' essere pensato come una handle ad un gruppo
- Un communicator risiede fisicamente su ogni singolo processo che lo riguarda
- `MPI_COMM_WORLD` e' il nome del *communicator* **FONDAMENTALE** che viene creato quando si inizializza un ambiente MPI. Ogni volta che c'e' un codice MPI `MPI_COMM_WORLD` viene creato. Non puo' essere distrutto.
- con MPI si possono definire piu' communicator contemporaneamente
- due processi possono spedirsi messaggi **SOLO** all'interno dello stesso communicator

Compilare un codice MPI... sul proprio pc

Esiste un server dedicato al corso su cui ci sono delle installazioni di MPI, pero' e' **caldamente** consigliato avere una versione di MPI che compili direttamente su dei propri pc (per esempio un laptop). In questo modo anche se il server e' giu', si possono comunque seguire gli esercizi.

Si sono verificati problemi con varie versioni di MPI, per esempio con OSX, per cui il suggerimento che posso dare e' il seguente:

- installare Virtualbox sul proprio pc (e' gratis)
- installare una versione di Linux (io uso Fedora, ma anche UBUNTU o altre distibuzioni non dovrebbero avere problemi). Consiglio almeno 2 Gb di memoria. Consiglio inoltre di creare un HD dinamico (che si ingrandisca in funzione dell'uso) ma di tenere i file che faremo girare nel corso su una cartella condivisa in modo da non espandere eccessivamente l'HD.
- installare `gcc`
- installare `openMPI` (e' gratuito) io, con una versione vecchia di Fedora ho utilizzato:
`sudo yum install openmpi-devel`, (oppure: `sudo yum install openmpi-devel.i686`) con una versione piu' recente di Fedora si usa `dnf` invece che `yum` per l'installazione) Al seguente sito ci sono risorse utili per l'installazione (anche se temo sia un po' vecchio):
http://wiki.nmr-relax.com/OpenMPI#Install_OpenMPI_on_linux_and_set_environments

Compilare sul server del corso: openMPI

Prima ci si connette al server...

```
ssh nomeutente@s-nomadis-06.mlab.disco.unimib.it
```

In caso di **problemi di connessione** (il server alle volte e' giu') contattare il **Prof. Dominoni** (io non sono un amministratore e non posso fare nulla a riguardo).

Per compilare sul server bisogna prima importare il modulo necessario, scrivendo:

```
module add
```

con il `tab` il computer mostra le scelte dei moduli da poter aggiungere. In particolare a noi interessa il compilatore:

```
module add mpi/openmpi-x86_64
```

Attenzione il modulo e' caricato solo nella shell in cui si lancia il comando. Per questo motivo se entro con un'altra shell e cerco di eseguire il codice... non funge! devo fare `module add ...` anche dall'altra shell

Attenzione bis se uno fa `add` sia di `openMPI` che `MPICH` ci sono dei conflitti quando si compila o fa girare. Nel dubbio si puo' eseguire un `module purge` eliminando i moduli gia' aggiunti e caricando quello che si vuole successivamente.

Per compilare:

```
mpicc -o mpi_hello.x mpi_hello.c
```

eseguire un codice openMPI

Per lanciare l'esecuzione di un programma e' necessario dire quanti **processi** sono impegnati.
Per esempio:

```
mpirun -np 4 hello_world.x
```

in questo caso voglio che vengano usati 4 processi per fare girare il codice.

Nel caso in cui il numero di **processori** disponibili sia minore di 4, allora il numero di **processi** che partono contemporaneamente sara' minore di 4.

Su alcune macchine (OSX alle volte) se si chiedono (tramite **-np**) piu' processi dei processori/core disponibili si puo' incorrere in un errore

Tipi “semplici”

Quando si passano delle strutture di dati ad una funzione MPI, bisogna indicare il **tipo** di dato. Non si usano i tipi intrinseci del linguaggio del codice (p.es. FORTRAN) perché in questo modo si ha maggiore portabilità’.

In pratica ad ogni tipo del FORTRAN (o del C) corrisponde un tipo in MPI che va indicato alla funzione. In questo senso i seguenti tipi sono “semplici” ovvero hanno una diretta corrispondenza con i tipi di dato cui siamo normalmente abituati.

In C i tipi sono scritti tutti in maiuscolo, per esempio:

- `MPI_INT` interi del C
- `MPI_FLOAT` float del C
- `MPI_DOUBLE` double (occhio che non si traduce automaticamente da double a float)
- `MPI_CHAR` corrisponde ai signed char del C
- `MPI_SHORT` corrisponde ai signed short del C
- `MPI_LONG` corrisponde signed long int del C

riassumendo: per molti tipi basta aggiungere `MPI_` davanti al tipo in C e scrivere tutto in maiuscolo.

altre strutture/tipi di MPI

Alcuni esempi di tipi di dato che sono disponibili solo nel framework di MPI.

- `MPI_Comm` e' l'oggetto communicator
- `MPI_Status` e' la struttura che definisce lo stato dei messaggi inviati e ricevuti
- `MPI_2INT` 2 interi
- `MPI_SHORT_INT` short e intero
- `MPI_LONG_INT`
- `MPI_LONG_DOUBLE_INT`
- `MPI_FLOAT_INT`
- `MPI_DOUBLE_INT`

Primo Codice: `MPI_Init()` e `MPI_Finalize()`

Quando si scrive un codice implementando lo standard MPI e' necessario aggiungere un header:

- `use mpi_f08` **FORTRAN 90:**
- `#include <mpi.h>` **C**

Sono poi necessarie 2 funzioni:

`MPI_Init()` e' la funzione che inizializza l'ambiente di esecuzione di MPI, deve essere chiamata **SEMPRE** (ma solo una volta!) da tutti i processi e prende due parametri di input:

- un puntatore al numero di argomenti
- un puntatore al vettore degli argomenti

(e' possibile passare anche una `NULL` ad entrambi gli argomenti)

- la routine restituisce a tutti i processi una copia della "argument list".
- una chiamata tipica della funzione e' la seguente: `MPI_Init(&argc, &argv);`

Alla fine del codice serve: `MPI_Finalize()`

`MPI_Finalize()` finalizza la fase di comunicazione, deve essere chiamata **SEMPRE** da **tutti** i processi alla fine del codice. Chiude in modo "pulito" l'ambiente di comunicazione.

- Dopo avere lanciato questa funzione e' bene chiudere il codice e non fare niente d'altro!
- dopo `MPI_Finalize` non e' proprio possibile chiamare una funzione MPI, neanche `MPI_Init()`!

Ciao Mondo 1

Esercizio:

- Prendere il codice (per un singolo processore) scritto in C qui sotto
- Inserire le due funzioni fondamentali di MPI
- Compilare il codice
- Eseguire il codice con 10 processi.

```
#include <stdio.h>           // header per il modulo standard input/output
#include "mpi.h"             // header per poter usare MPI
int main(int argc, char** argv) { // main function del programma

    printf("\tCiao Mondo!\n"); // stampa a video dopo un TAB, Ciao Mondo e vai a capo

    return 0;               // dato che il main e' int restituisci 0
}
```

Se `mpicc` non funge, (ma noi speriamo funga) si devono usare delle flag del tipo:

```
gcc -I/usr/include/openmpi-x86_64 -pthread -Wl,-rpath -Wl,/usr/lib64/openmpi/lib
-Wl,-enable-new-dtags -L/usr/lib64/openmpi/lib -lmpi
```

Ciao Mondo 2

Esercizio:

- Prendere il codice (per un singolo processore) scritto in C qui sotto
- Inserire le due funzioni fondamentali di MPI
- Compilare il codice
- Eseguire il codice con 10 processi.

```

#include <stdio.h>           // header per il modulo standard input/output
#include "mpi.h"            // header per poter usare MPI
int main(int argc, char** argv) { // main function del programma
    MPI_Init(&argc, &argv);    // prima funzione MPI
    printf("\tCiao Mondo!\n"); // stampa a video dopo un TAB, Ciao Mondo e vai a capo
    MPI_Finalize();           // chiudi in modo pulito MPI
    return 0;                 // dato che il main e' int restituisci 0
}

```

Soluzione

- `MPI_Init` e `MPI_Finalize` sono le due funzioni essenziali per MPI
- in base a quanti processi lanciamo ci sarà un numero corrispondente di scritte a video "Ciao Mondo": **tutti** i processi eseguono il medesimo codice per intero (SPMD).
- La formattazione dell'output presenta qualche stranezza? I tab vengono usati in posizioni random!

Diapositive dell'anno scorso

- Le diapositive dell'anno scorso e anche i file che vanno poi completati si possono trovare al seguente indirizzo:
<https://github.com/4physics/mpi-ita-2018-19>
- nella cartella chiamata **NO** ci sono i codici che vengono proposti per gli esercizi.

MPI_Comm_rank

questa funzione viene specificata nel seguente modo:

```
int MPI_Comm_rank( MPI_Comm comm, int *miorank )
```

tipo
nome comunicator
tipo
puntatore al nome variabile dove metto il rank

- Serve a determinare il **rank** del **processo** chiamante all'interno del communicator chiamato `comm`, l'informazione viene messa nella variabile `miorank` (nome di esempio ma scegliibile a piacere, chiaramente dovrete avere dichiarato la variabile in precedenza).
- Il **rank** e' un numero intero che va da 0 alla dimensione del communicator (**size**) meno 1.
- Il rank di un processo (nel communicator) e' usato in pratica come una sorta di "indirizzo" del processo stesso quando si spediscono o ricevono messaggi da esso.
- **Attenzione:** non esiste IL rank di un processo: si ha sempre il rank **rispetto** ad un communicator.
- esempio di una chiamata a questa funzione:

```
MPI_Comm_rank( MPI_COMM_WORLD, &miorank ) ;
```

dove `miorank` e' il nome precedentemente assegnato alla variabile e che deve contenere il rank stesso.

Domanda: perche' ho messo un & davanti al nome della variabile?

MPI_Comm_size

La procedura `MPI_Comm_size` viene specificata nel seguente modo:

```
int MPI_Comm_size ( MPI_Comm comm, int *nproc )
```

- il primo argomento della funzione e' una variabile di input contenente il nome del communicator a cui ci si riferisce (in questo caso chiamato `comm`).
- il secondo argomento della funzione (`nproc`) e' una variabile di output che conterra' la grandezza (**size**) del communicator stesso. Occhio! qui si passa il **puntatore** alla variabile.
- una chiamata tipica alla funzione e' la seguente:

```
MPI_Comm_size ( MPI_COMM_WORLD, &nproc ) ;
```

dove `nproc` e' una variabile intera definita precedentemente che contiene il rank del processo.

Chi sono io?

- Prendere il codice precedente (Ciao Mondo) e modificarlo in modo che ogni processo stampi a video il proprio **rank**
- e anche la dimensione (**size**) del communicator
- trucco: per stampare a video si puo' usare qualcosa di simile a:
`printf("Ciao, sono il processo %d di %d", miorank, miasize);`

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv) {

MPI_Init(&argc, &argv);

MPI_Finalize();
return 0;
}
```

Chi sono io? soluzione

- Prendere il codice precedente (Ciao Mondo) e modificarlo in modo che ogni processo stampi il proprio **rank**
- e anche la dimensione (**size**) del communicator

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv) {
    int me, nproc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    printf("Ciao, sono il processo %d di %d \n", me, nproc);
    MPI_Finalize();
    return 0;
}
```

Si noti che ri-lanciando il codice in momenti differenti si possono ottenere scritte a video differenti. Per esempio, potrà apparire la scritta del processo 1 e poi dello 0, o viceversa. Questo è un altro esempio di comportamento **NON DETERMINISTICO** di un codice MPI: il risultato dipende da quanto sono occupati i processi che eseguono il codice.

Attenzione: per un codice semplice come questo i risultati saranno spesso identici per il poco carico computazionale, quindi fare qualche esperimento potrebbe non essere sufficiente per determinare se il codice funziona in modo deterministico.

SPMD: diversi processi-diversi dati:

Single Process Multiple Data

- MPI si occupa di gestire il **trasferimento** di messaggi ed informazioni tra processi
- Quando si lancia un codice MPI, per un supercomputer, si dovrà scrivere uno **script** che indica quale codice viene lanciato, quali sono i file di input etc...
- E' possibile utilizzare alcune funzioni MPI per fare del parallel-IO, ovvero si può gestire una notevole mole scrittura/lettura in parallelo.
- Nei casi più semplici, come quelli che vedremo noi, si sfrutta il **rank** per decidere che un particolare processo debba accedere ad un file particolare e poi si può fare circolare l'informazione raccolta tramite le normali funzioni di message passing...

Point-to-point: Spedire Messaggi

Un messaggio e' composto di:

envelope (busta): con sorgente (**rank** del sender), destinazione (**rank** del processo receiver), communicator (dove stanno comunicando i processi), **tag** (intero non negativo usato per identificare il messaggio, utile se il sender ne manda piu' di uno per volta).

body (corpo):

- *buffer* = message data
- *datatype* = il tipo dei data of the message
- *count* = quanti di questi dati sono mandati

Parliamo di **point-to-point** communication perche' un solo processo comunica con un solo processo.

La **lunghezza** del messaggio deve essere controllata indicando *esplicitamente* il numero di "oggetti elementari" inviati e il numero di oggetti che si aspetta i ricevere. Se le informazioni mandate sono di piu' di quelle che il ricevitore si aspetta si ha un OVERFLOW ERROR, nel caso opposto non si ha un errore esplicito (MA DE FACTO e' un PROBLEMA). Con MPI si vuole che la struttura dell'informazione che viene mandata sia proprio quella che ci si aspetta di ricevere.

- **idle** il processo aspetta...
- **deadlock** qualche processo e' fermo ad aspettare qualcosa che non succedera' mai...
- **La coerenza/correttezza semantica** di una operazione di send e receive: il fatto che arrivi l'informazione che voglio mandare. Per esempio, supponiamo che il processo 3 invii al processo 4 una variabile: $a = 0$. L'azione di comunicazione viene completata quando il processo 4 riceve. Se 4 chiama la funzione di ricezione 10s dopo l'invio, nel frattempo il valore di a , potrebbe essere cambiato (esempio $a = 1$) e in questo modo si perde coerenza semantica: la stessa variabile viene inviata e ricevuta, ma assume un valore diverso da quello che volevo inviare!
Coerenza semantica implica che la informazione mandata e ricevuta non vengano modificate durante la comunicazione.

Questo **buffo** nome... quando si cerca in internet si trovano diversi significati associati ad uno stesso termine e questo puo' creare confusione:

- **buffer** alle volte questo termine viene riferito al corpo del messaggio (spesso)
- alle volte invece, tutto il messaggio viene chiamato **buffer**...
- per **buffer**, alle volte, ci si riferisce all'area di memoria dedicata a mantenere il messaggio durante una comunicazione tra processi.

MPI_Send

La funzione e' usata per **mandare** messaggi ad altri processi:

```
int MPI_Send(
void          *oggetto, (alle volte questo viene chiamato "buffer")
int          quantiDati,
MPI_Datatype tipoDati,
int          rankDestinatario,
int          tag,
MPI_Comm     nomeCommunicator
);
```

- MPI_Send e' una funzione **blocking**: finche' la funzione non "ritorna" il processo si blocca. **PERICOLO di DEADLOCK**
- Perche' il puntatore a `oggetto` ha come tipo `void`? perche' il tipo dei dati e' specificato comunque dopo!
- Perche' devo dare un puntatore all'array da mandare e non tutto il l'array? cosi' risparmio spazio! E' MPI a lavorare per trovare i dati; inoltre in C, visto che le informazioni sono passate per valore normalmente si passano dei puntatori.
- Il sistema vuole solo sapere dove inizia l'area di memoria dove ci sono i dati.

Consigli 1

Supponiamo di avere costruito un array con 5 ingressi chiamato `ciccio`:

```
ciccio[0]=0 ciccio[1]=1 ciccio[2]=4 ciccio[3]=9 ciccio[4]=16
```

Supponiamo ora di voler mandare tutto l'array dal processo 0 al processo 1:

```
MPI_Send( ciccio , 5, MPI_INT, 1, 999, MPI_COMM_WORLD) ;
```

Come legge il **computer** questa funzione?

- 1 vai e metti all'inizio dell'area di memoria dove c'e' l'array `ciccio`.
- 2 prendi 5 oggetti
- 3 di tipo: intero
- 4 manda il messaggio al processo con rank =1
- 5 associa al messaggio da mandare il tag 999 (scelto da me, basta che chi riceve usi lo stesso tag). Il tag (etichetta) e' un ulteriore modo per specificare il messaggio
- 6 tutto questo deve avvenire nel communicator chiamato `MPI_COMM_WORLD`

Decay: Abbiamo scritto `ciccio` e non `&ciccio` perche' il nome dell'array senza ingressi implica il puntatore all'inizio dell'array stesso! Se avessimo voluto mandare 3 ingressi: `ciccio[2] ciccio[3] ciccio[4]` , avremmo dovuto modificare la scrittura in:

```
MPI_Send( &ciccio[2] , 3, MPI_INT, 1, 999, MPI_COMM_WORLD) ;
```

Visualizziamo invio messaggi

Cervello_uomo

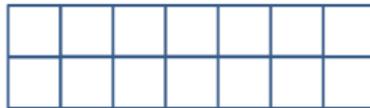


Persona =1

```

Buffer          = Cervello_uomo[2]
quantiDati     = 2
tipoDati       = cuore
rankDestinatario= 2
tag            = 1
nomeComm       = Whatsapp
  
```

Cervello_donna



Persona =2

Consigli 2

In MPI spesso si lavora con aree di memoria contigue, per cui e' sufficiente indicare il puntatore al primo elemento, il tipo e il numero degli elementi necessari. Per questo e' importante sapere come sono stoccate in memoria le informazioni. Pensiamo ad una matrice 3 x 3:

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{pmatrix} \quad (1)$$

Come vengono immagazzinati in memoria questi dati?

Pericolo: in **C** l'array a : 3×3 viene stoccato per **righe**:

$$a[0][0] \ a[0][1] \ a[0][2] \ a[1][0] \ a[1][1] \ a[1][2] \ a[2][0] \ a[2][1] \ a[2][2] \quad (2)$$

Lo stesso array in **FORTRAN** invece viene stoccato per **colonne** (occhio che in FORTRAN non si scrivono cosi' gli array 2D):

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{pmatrix} \quad (3)$$

$$a[0][0] \ a[1][0] \ a[2][0] \ a[0][1] \ a[1][1] \ a[2][1] \ a[0][2] \ a[1][2] \ a[2][2] \quad (4)$$

MPI_Recv

Questa funzione e' usata per ricevere messaggi da altri processi:

```
int MPI_Recv(
void          *nomeArray,
int          quantiDati,
MPI_Datatype tipoDati,
int          rankMittente,
int          tag,
MPI_Comm     nomeCommunicator
MPI_Status   *stat // (ignorabile con MPI_STATUS_IGNORE)
);
```

- MPI_Recv e' una funzione **blocking**: finche' non ha eseguito il proprio compito il **processo** si blocca.
- la variabile **stat**, che deve essere precedentemente e correttamente definita nel codice e contiene i possibili problemi nella ricezione (dedicheremo un pezzo di una lezione futura agli errori)
- esistono vari tipi di funzioni per spedire messaggi, per esempio MPI_Send, MPI_Ssend e MPI_Bsend... la routine MPI_Recv puo' essere usata per ricevere da ognuna di esse!

Visualizziamo la ricezione di messaggi

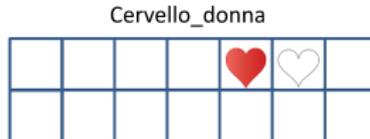


Persona =1

```

Buffer          = Cervello_uomo[2]
quantiDati     = 2
tipoDati       = cuore
rankDestinatario = 2
tag            = 1

nomeComm       = Whatsapp
  
```

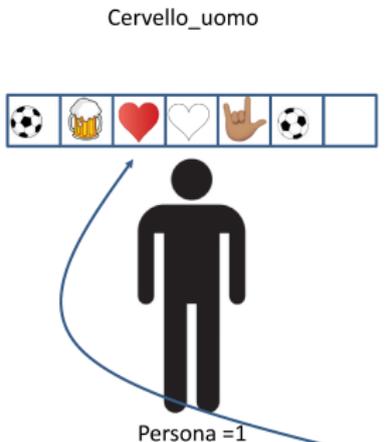


Persona =2

```

Buffer          = Cervello_donna[4]
quantiDati     = 2
tipoDati       = cuore
rankMittente   = 1
tag            = 1
Mpi_status     = ignore
nomeComm       = Whatsapp
  
```

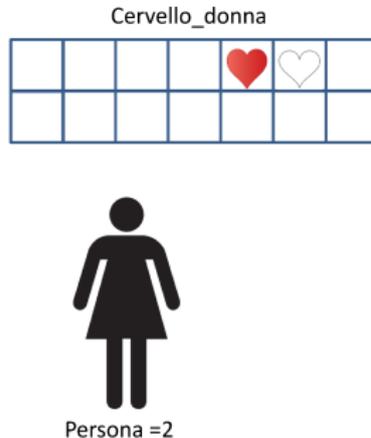
Visualizziamo la ricezione di messaggi 2



```

Buffer          = Cervello_uomo[2]
quantiDati      = 2
tipoDati        = cuore
rankDestinatario = 2
tag             = 1

nomeComm        = Whatsapp
  
```



```

Buffer          = Cervello_donna[4]
quantiDati      = 2
tipoDati        = cuore
rankMittente    = 1
tag             = 1
Mpi_status      = ignore
nomeComm        = Whatsapp
  
```

Visualizziamo la ricezione di messaggi 3

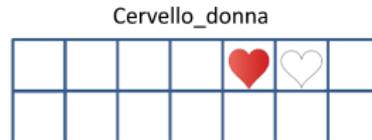


Persona =1

```

Buffer          = Cervello_uomo[2]
quantiDati     = 2
tipoDati       = cuore
rankDestinatario = 2
tag            = 1

nomeComm       = Whatsapp
  
```

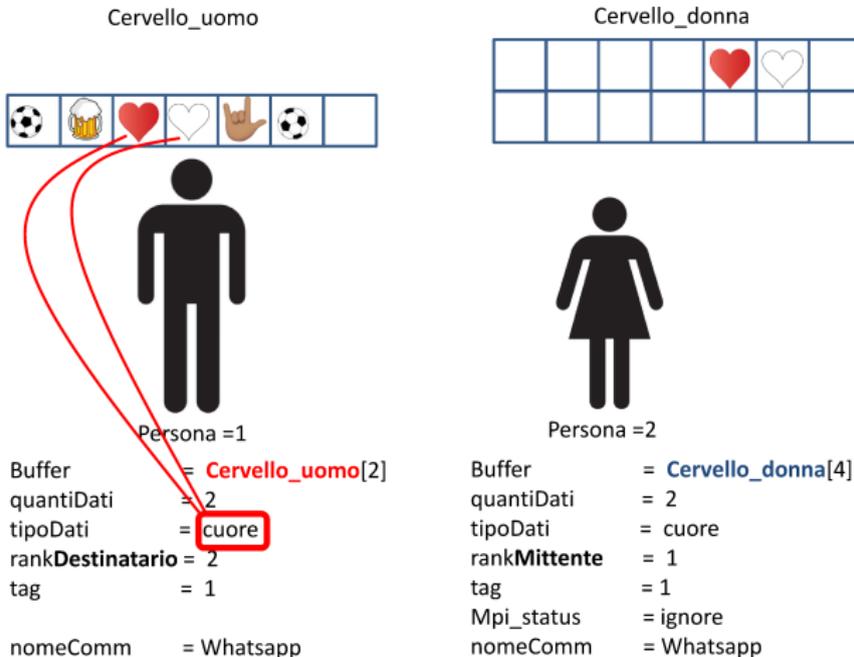


Persona =2

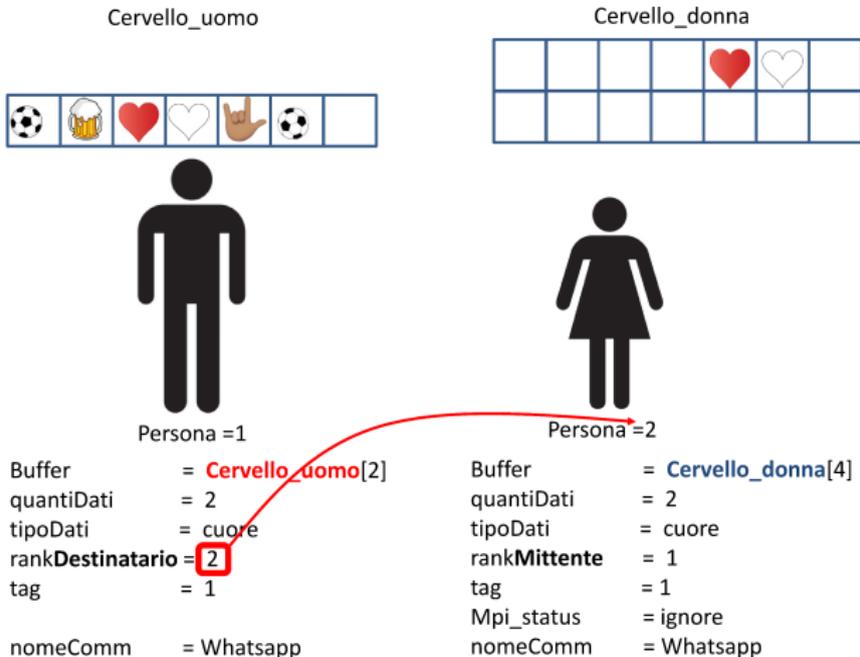
```

Buffer          = Cervello_donna[4]
quantiDati     = 2
tipoDati       = cuore
rankMittente   = 1
tag            = 1
Mpi_status     = ignore
nomeComm       = Whatsapp
  
```

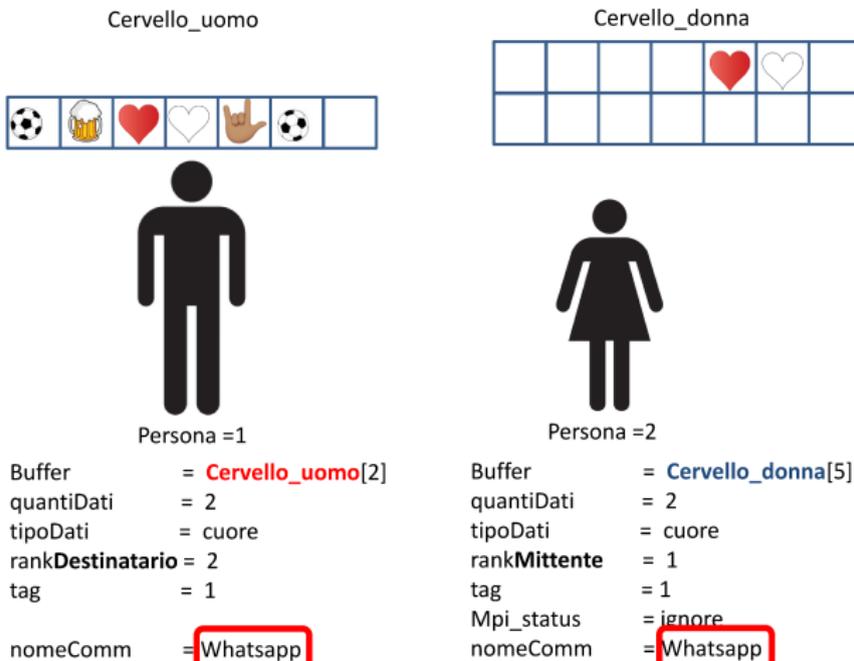
Visualizziamo la ricezione di messaggi 4



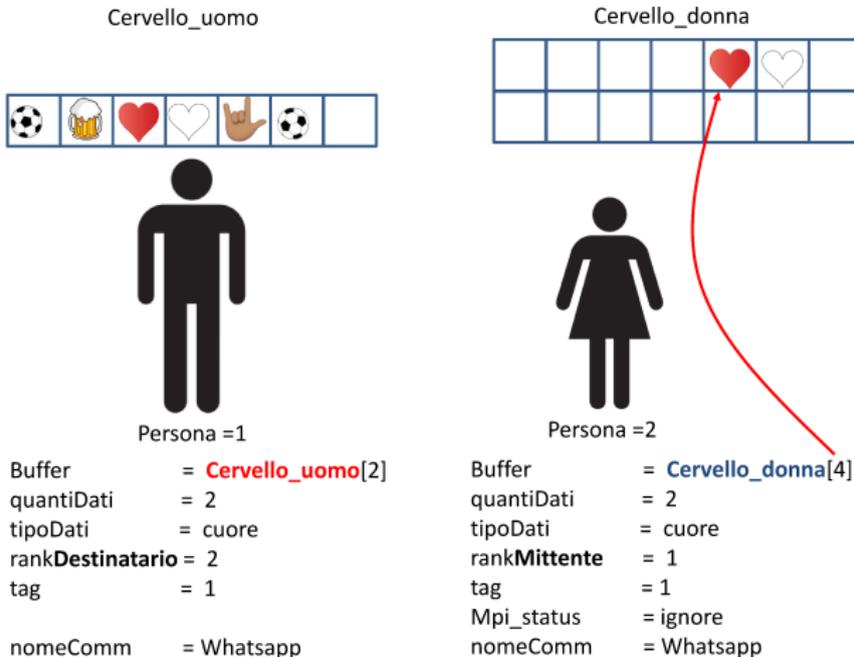
Visualizziamo la ricezione di messaggi 5



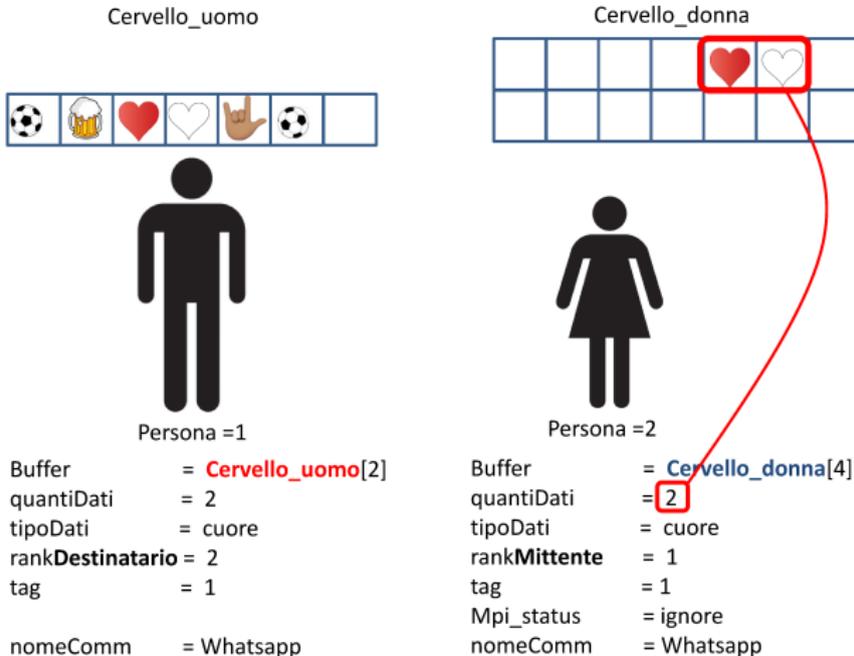
Visualizziamo la ricezione di messaggi 6



Visualizziamo la ricezione di messaggi 7



Visualizziamo la ricezione di messaggi 8



Visualizziamo la ricezione di messaggi 9

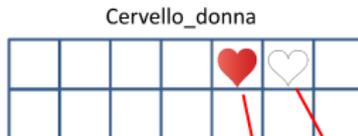


Persona =1

```

Buffer          = Cervello_uomo[2]
quantiDati      = 2
tipoDati        = cuore
rankDestinatario = 2
tag             = 1

nomeComm        = Whatsapp
  
```

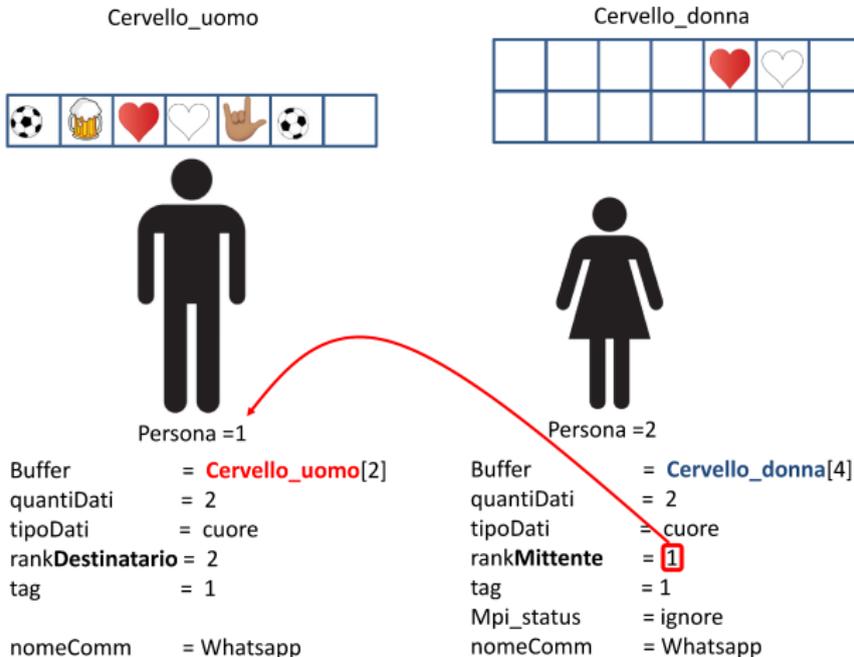


Persona =2

```

Buffer          = Cervello_donna[4]
quantiDati      = 2
tipoDati        = cuore
rankMittente    = 1
tag             = 1
Mpi_status      = ignore
nomeComm        = Whatsapp
  
```

Visualizziamo la ricezione di messaggi 10



Mandiamo un messaggio

- creiamo un array chiamato `ciccio`
- mandiamo i primi 3 ingressi di `ciccio` dal rank 0 al rank 1
- facciamo scrivere a video al rank 1 quello che ha ricevuto

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])  {
    int rank, size, i;
    int ciccio[10];                // array che voglio mandare
    MPI_Status status;            // oggetto di MPI che serve quando si riceve un mess
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)  {              // rank= 0 riempie l'array e manda il messaggio
        for (i=0; i<10; i++) ciccio[i] = i; // il rank 0 riempie l'array chiamato ciccio
    }

    if (rank == 1)  {              // rank= 1 ricevi il messaggio
        for (i=0; i<10; i++) ciccio[i] = 0; // anche il rank 1 ha un array ciccio pieno di 0,

        for (i=0; i<10; i++) {     // vediamo adesso gli ingressi di ciccio a video
            printf(" ciccio[%d] = %d \n", i, ciccio[i]);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Mandiamo un messaggio: soluzione

- creiamo un array chiamato `ciccio`
- mandiamo i primi 3 ingressi di `ciccio` dal rank 0 al rank 1
- facciamo scrivere a video al rank 1 quello che ha ricevuto

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, size, i;
    int ciccio[10]; // array che voglio mandare
    MPI_Status status; // oggetto di MPI che serve quando si riceve un mess
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) { // rank= 0 riempie l'array e manda il messaggio
        for (i=0; i<10; i++) ciccio[i] = i; // il rank 0 riempie l'array chiamato ciccio
        MPI_Send(ciccio, 3, MPI_INT, 1, 123, MPI_COMM_WORLD);
    } // manda l'array ciccio al processo 1
    if (rank == 1) { // rank= 1 ricevi il messaggio
        for (i=0; i<10; i++) ciccio[i] = 0; // anche il rank 1 ha un array ciccio pieno di 0,
        MPI_Recv(ciccio, 3, MPI_INT, 0, 123, MPI_COMM_WORLD, &status); // ricevi ciccio da 0
        for (i=0; i<10; i++) { // vediamo adesso gli ingressi di ciccio a video
            printf(" ciccio[%d] = %d \n", i, ciccio[i]);
        }
    }
    MPI_Finalize();
    return 0;
}

```

Mandiamo un messaggio... un po' piu' in la'

- come nell'esercizio precedente mandiamo un messaggio dal rank 0 al rank 1
- in questo caso pero' mandiamo ancora 3 ingressi dell'array `ciccio` ma a partire dal 4to ingresso.
- facciamo scrivere a video l'informazione ricevuta dal rank 1.

Invece che:

```
MPI_Send(ciccio , 3, MPI_INT, 1, 123, MPI_COMM_WORLD);
```

metteremo

```
MPI_Send(&ciccio[3], 3, MPI_INT, 1, 123, MPI_COMM_WORLD);
```

- Si noti che ora c'e' un `&` dinnanzi al termine `ciccio[3]`, questo perche' ci stiamo riferendo ad un ingresso particolare (`[3]`) dell'array, e quindi per avere il puntatore dobbiamo usare il `&`
- Attenzione, nell'esercizio facciamo immagazzinare le informazioni ricevute nell'array `ciccio`, se vogliamo che siano nella medesima posizione rispetto all'array di partenza dobbiamo modificare anche il `MPI_Recv`

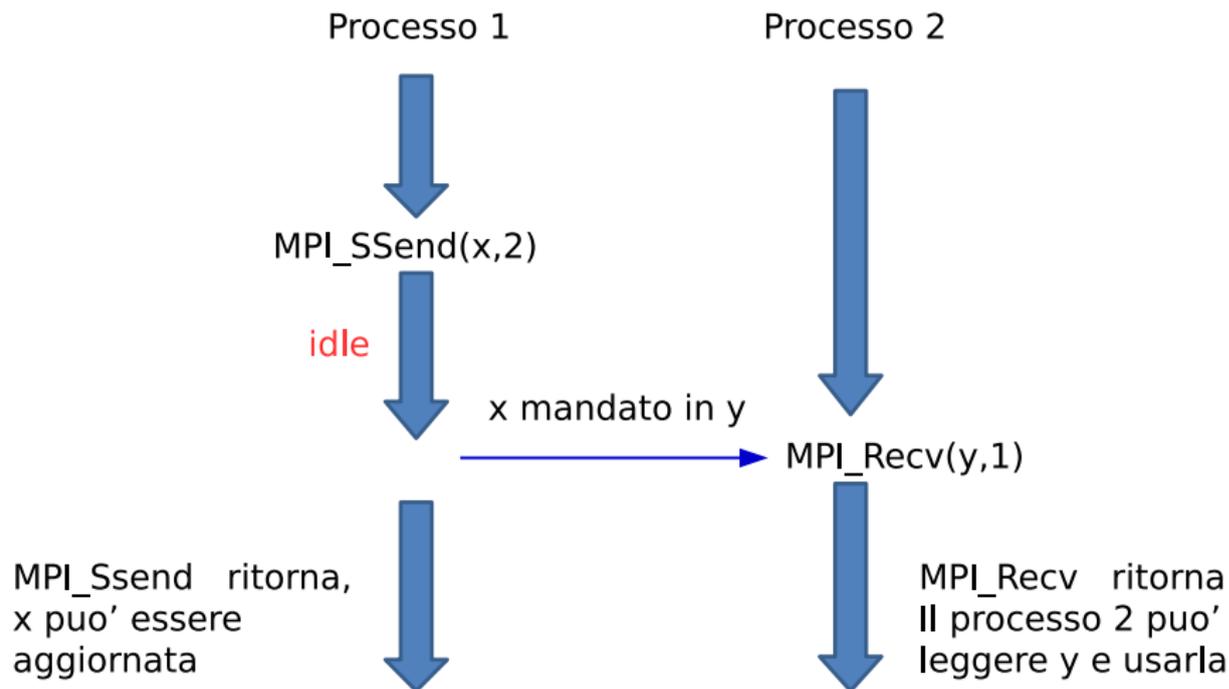
Un altro tipo di Send: MPI_Ssend

Nei corsi introduttivi a MPI, spesso si tralasciano (o si mettono alla fine) le funzioni alternative di invio, noi invece le useremo per capire meglio `MPI_Ssend()`:

MPI_Ssend: sincronizzato

- Può essere lanciato **prima** che venga lanciato il `receive` dall'altro processo (altri tipi di `send` invece necessitano che ci sia già un `receive` in ascolto)
- il `Receive` deve cominciare prima che `Ssend` spedisca qualcosa (nel frattempo il processo è **bloccato!**)
- È il modo di mandare più **sicuro** (dal punto di vista semantico) e portabile (un codice scritto con `MPI_Ssend` non dipende dall'implementazione di MPI usata)
- Non ha bisogno di *buffer* (non ben definiti...)
- Può generare un **overhead** notevole per la sincronizzazione

MPI_Ssend: sincronizzato



un altro Send ancora: MPI_Bsend()

MPI_Bsend: bufferizzato

- il send puo' essere completato prima che il `receive` cominci!!!
- bisogna allocare in modo esplicito un buffer (spazio di memoria) con `MPI_Buffer_attach`
- puo' generare errore se si manda un messaggio piu' grande del buffer
- non ha **overhead** di sincronizzazione (ma bisogna usare memoria aggiuntiva, rispetto, per esempio `Ssend`)

`MPI_Buffer_attach(buffer, size)`

- *in ingresso*: `buffer` = indirizzo iniziale (puntatore) al buffer
- *in ingresso*: `size` = dimensione del buffer, in `byte` (integer)

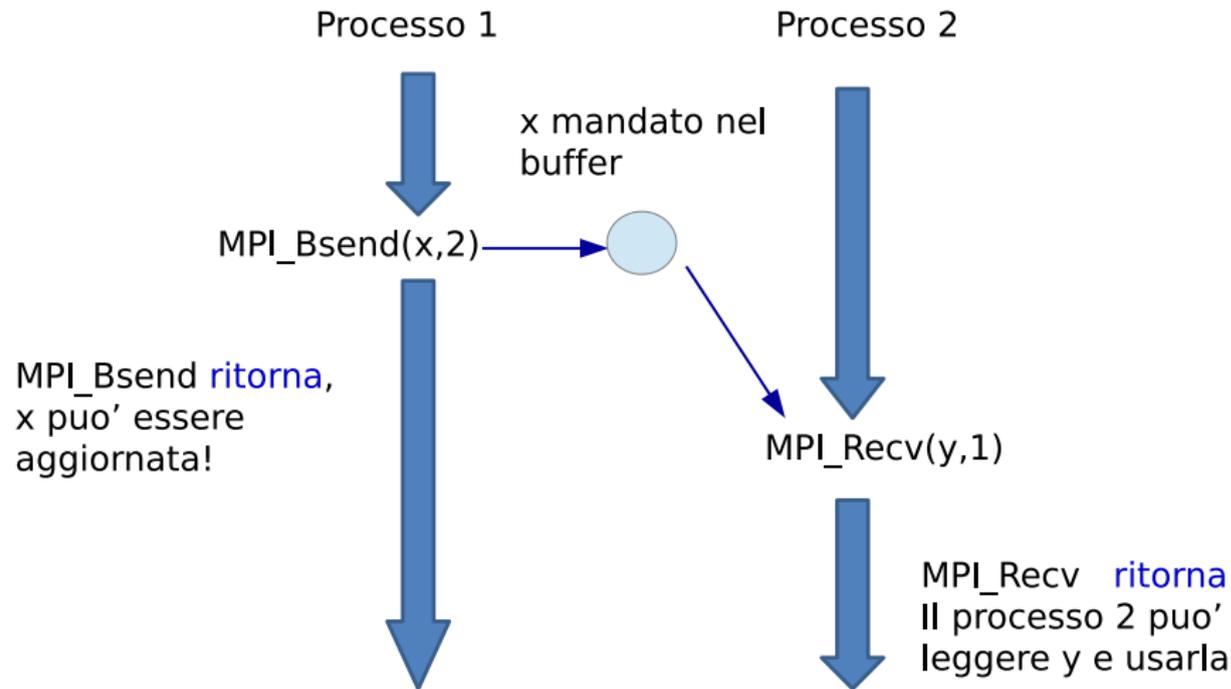
esempio `int MPI_Buffer_attach(void* buffer, int size)`

Similmente c'e' `MPI_Buffere_detach ()`

Attenzione: la `size` va calcolata correttamente:

https://www.mpich.org/static/docs/latest/www3/MPI_Buffer_attach.html

MPI_Bsend: buffering



... e quindi MPI_Send?

`MPI_Send` e' una funzione **blocking**: il nome significa che il processo si **blocca** finche' la trasmissione del messaggio non e' avvenuta correttamente (esegue il return).

NO! Non e' vero!!!

il comportamento dipende dalla implementazione di MPI, per via del **buffering**. Quello che succede e' che (in alcune implementazioni, come per esempio `openMPI`) i dati sono mandati ad un buffer.

- Il buffer e' deciso automaticamente dall'implementazione (a differenza di `MPI_Bsend`).
- Se il buffer non si riempie, si comporta come `MPI_Bsend`. Ovvero il messaggio va al buffer e l'altro processo puo' leggere con comodo dato che il messaggio e' li'.
- **Se pero' il buffer e' piu' piccolo del messaggio stesso** (per esempio un array grande) allora il buffer non si libera finche' non viene letto. Si comporta quindi come `MPI_Ssend`. Per questo si puo' incappare in un **deadlock**.

Un insieme di operazioni in questo ordine:

P0	P1
send to 1	send to 0
receive from 1	receive from 0

e' quindi considerato **unsafe**.

Usando funzioni blocking che non hanno un buffer produce sicuramente un **deadlock**! La lunghezza dei dati da mandare deve essere nota a priori, altrimenti si possono usare altre funzioni che ispezionano i dati e in questo modo si possono allocare dinamicamente.

Riassumendo: i magnifici sei

nelle lezioni precedenti abbiamo visto quelle che sono le 6 funzioni fondamentali per MPI:

- 1 `MPI_Init()`
- 2 `MPI_Finalize()`
- 3 `MPI_Comm_size()`
- 4 `MPI_Comm_rank()`
- 5 `MPI_Send()`
- 6 `MPI_Recv()`

Con queste funzioni e' possibile, in linea di principio, scrivere un qualunque codice... anche se il passaggio di informazioni da un processo all'altro, probabilmente, non sara' ottimizzato.

Un codice un po' piu' complesso (ma non molto!)

Come già accennato, scrivere un codice MPI è complicato (un ordine di grandezza più che un codice scalare). È molto difficile scrivere un codice *seriale* e, a **posteriori**, parallelizzarlo. È meglio:

- progettare il codice pensato in termini di **concorrenza** (potenziale parallelismo)
- basando il codice sulle **caratteristiche** della macchina su cui dovrà girare

Alcuni desiderata:

- vorremmo **scalabilità** con il numero di processi (spesso non sappiamo in anticipo quanti processori avremo a disposizione)
- comportamento **deterministico** (non vogliamo un risultato che sia differente ogni esecuzione)
- vorremmo un codice che sia **efficiente** (se lancio il calcolo su 10000 processi voglio che solo una minima parte del tempo venga passata in stato di **idle**)
- **dobbiamo** evitare i **deadlock**

Fare circolare i dati in un anello:

Proviamo a scrivere un codice in cui:

- 1 a partire dal processo con rank 0, un messaggio (numero intero) viene passato, in ordine, a tutti gli altri rank, e alla fine il processo con rank $n - 1$, manda il messaggio (modificato da questi passaggi) al rank 0.
- 2 dopo la ricezione, ogni processo aggiorna: messaggio = messaggio+1
- 3 alla fine del codice, ogni processo scrive il proprio rank e il messaggio nella propria memoria (per controllare)

Fare circolare i dati in un anello:

Proviamo a scrivere un codice in cui:

- 1 a partire dal processo con rank 0, un messaggio (numero intero) viene passato, in ordine, a tutti gli altri rank, e alla fine il processo con rank $n - 1$, manda il messaggio (modificato da questi passaggi) al rank 0.
- 2 dopo la ricezione, ogni processo aggiorna: messaggio = messaggio+1
- 3 alla fine del codice, ogni processo scrive il proprio rank e il messaggio nella propria memoria (per controllare)

Fare circolare i dati in un anello:

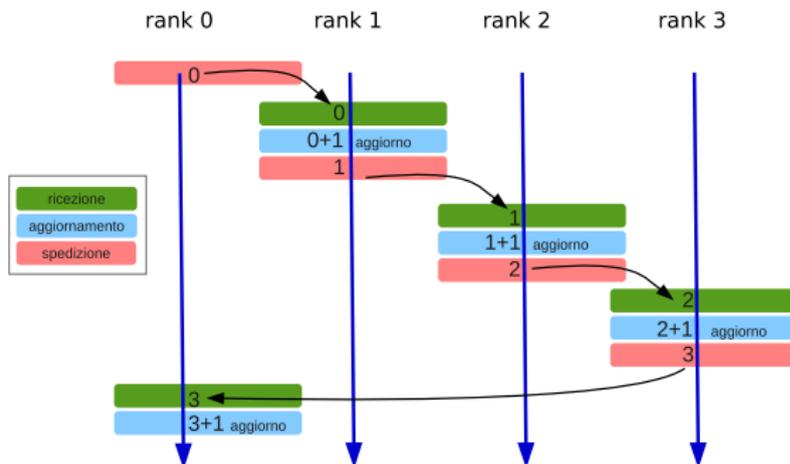
Proviamo a scrivere un codice in cui:

- 1 a partire dal processo con rank 0, un messaggio (numero intero) viene passato, in ordine, a tutti gli altri rank, e alla fine il processo con rank $n - 1$, manda il messaggio (modificato da questi passaggi) al rank 0.
- 2 dopo la ricezione, ogni processo aggiorna: messaggio = messaggio+1
- 3 alla fine del codice, ogni processo scrive il proprio rank e il messaggio nella propria memoria (per controllare)

Fare circolare i dati in un anello:

Proviamo a scrivere un codice in cui:

- 1 a partire dal processo con rank 0, un messaggio (numero intero) viene passato, in ordine, a tutti gli altri rank, e alla fine il processo con rank $n - 1$, manda il messaggio (modificato da questi passaggi) al rank 0.
- 2 dopo la ricezione, ogni processo aggiorna: $\text{messaggio} = \text{messaggio} + 1$
- 3 alla fine del codice, ogni processo scrive il proprio rank e il messaggio nella propria memoria (per controllare)



Telefono senza fili: ring

- si costruisca, utilizzando le funzioni `MPI_Send` e `MPI_Recv`, un codice secondo la logica di quanto illustrato nella diapositiva precedente.
- il codice deve poter funzionare con un numero arbitrario di processi (non solo 4...)

```
#include <mpi.h> // versione modificata da PA del codice ring.c su www.mpitutorial.com
#include <stdio.h>
int main(int argc, char** argv) {
    int world_rank;
    int world_size;
    int messaggio;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // INSERIRE QUI LE FUNZIONI MPI_Send e MPI_Recv
    // INSERIRE QUI LE FUNZIONI MPI_Send e MPI_Recv
    // INSERIRE QUI LE FUNZIONI MPI_Send e MPI_Recv

    MPI_Finalize();
}
```

Telefono senza fili: ring soluzione

```

#include <mpi.h> // versione modificata da PA del codice ring.c su www.mpitutorial.com
#include <stdio.h>
int main(int argc, char** argv) {

    int mio_rank;
    int size;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &mio_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int messaggio =0; // all'inizio messaggio =0 per tutti
    int daChi; // daChi ricevi i messaggi?
    daChi = mio_rank -1 ; // daChi ricevi? dal processo con rank precedente
    if (daChi < 0 ) daChi = daChi + size; // daChi ricevi? se sei 0 ricevi dall'ultimo rank
    if (mio_rank != 0) { // prepara tutti i processi a ricevere (tranne lo 0)
        MPI_Recv(&messaggio, 1, MPI_INT, daChi, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        messaggio= messaggio+1 ; // aggiungi 1 al messaggio
    }

    MPI_Send(&messaggio, 1, MPI_INT, (mio_rank + 1) % size, 0,
        MPI_COMM_WORLD); // ordina a tutti i processi di inviare
    if (mio_rank == 0) { // fai ricevere il processo 0 dal n-1
        MPI_Recv(&messaggio, 1, MPI_INT, daChi, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        messaggio= messaggio+1 ; // aggiungi 1 al messaggio
    }
    printf("Il processo %d ha ricevuto il messaggio %d dal processo %d\n",
        mio_rank, messaggio-1, daChi);
    MPI_Finalize();
}

```

Ring: ovvero

riassumiamo l'ordine delle chiamate del codice ring:

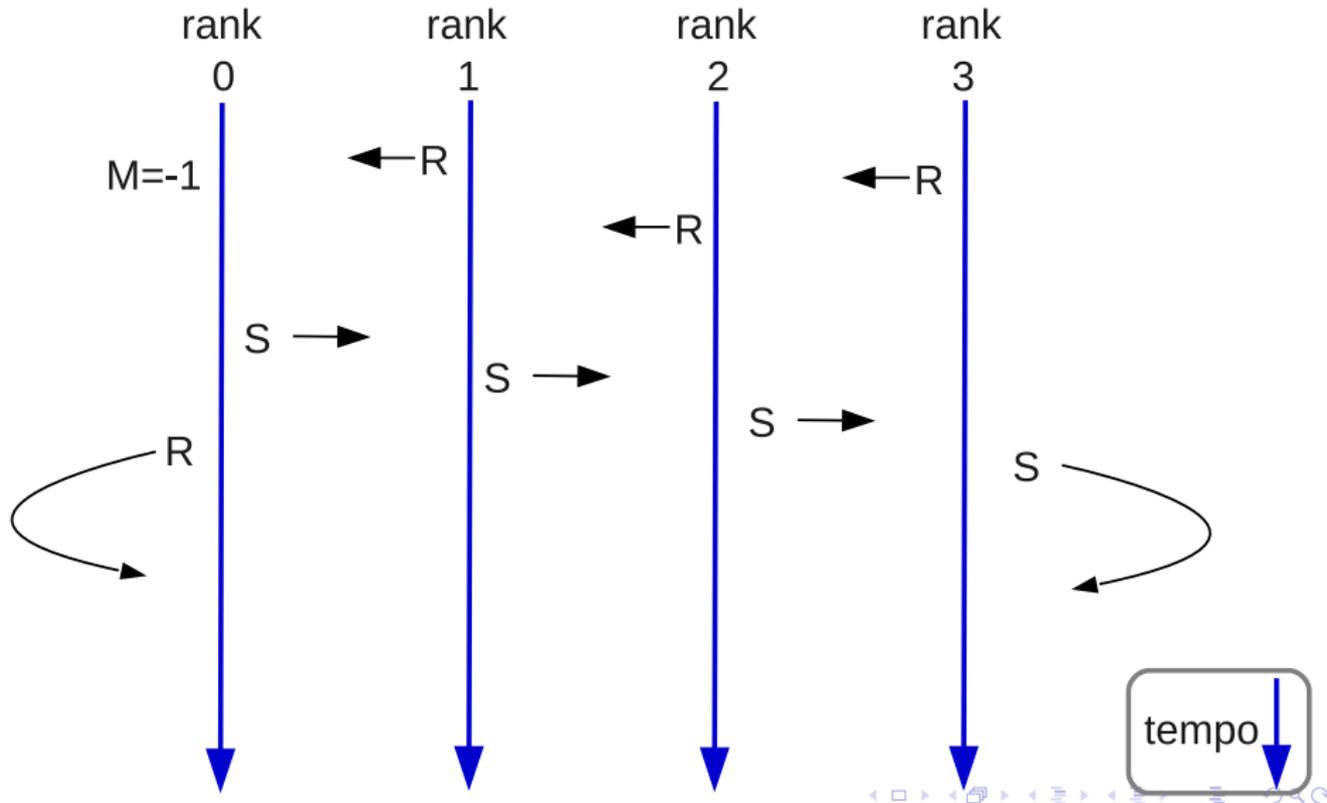
- 1 passo 1:
 - Tutti i processi (**tranne lo 0**): chiamano un `MPI_Recv` per essere pronti a ricevere
 - il processo **0** inizializza il messaggio da mandare
- 2 passo 2: Tutti i processi (incluso lo **0**) chiamano un `MPI_Send` per inviare
- 3 passo 3: il processo 0 chiama un `MPI_Recv` per ricevere

Domande:

- Cosa succede se inverte l'ordine di una di queste chiamate?
- Testiamo la dimensione del buffer definito implicitamente da `MPI_Send` nell'implementazione OpenMPI.

Esercizio: provare a modificare il ring in modo che venga fatto piu' di un giro tra tutti i processi.
C'e' rischio di **deadlock**? Come si puo' prevenire?

La tempistica di esecuzione non e' nota, per esempio:



Collective Communications

Finora abbiamo visto solo delle routine di comunicazione che si identificano come **point-to-point** (ovvero la comunicazione avviene da un singolo processo ad un singolo processo), ora invece:

- La caratteristica principale delle **collective communications** è che riguardano **molti** processi per volta.
- Se un processo chiama una funzione collettiva, deve **sempre** essere specificato in quale **communicator** ci si trova. Questo perché **tutti** gli altri processi dello **stesso communicator** **devono** chiamare la stessa funzione.
- Qualora anche **solo uno dei processi non arrivasse** a chiamarla si arriverebbe ad un **deadlock** \leftarrow **IMPORTANTE!!!!**

MPI_Barrier

Questa funzione e' pensata per creare dei **limiti**, e fare si' che il calcolo continui solo se alcuni compiti sono stati eseguiti completamente.

- Ricorda che i processi agiscono in “parallelo”, quando incontrano la barriera non significa che hanno finito di eseguire tutti i loro compiti, significa solo che non continuano ad eseguire finche' **tutti** i processi hanno chiamato la funzione `MPI_Barrier`.
- Questo implica che se, per esempio, c'e' un `if` e uno dei processi non chiama la barriera, allora tutto il codice si ferma ad aspettare quel processo. Questo e' vero per qualsiasi **collective communication** effettuata: **finche' tutti i processi non chiamano la stessa funzione tutti i processi non continuano**: sincronizzazione!
- esempio d'uso: `MPI_Barrier(MPI_COMM_WORLD)`

una Collective semplice: MPI_Barrier

Lo scopo di questo esercizio e' creare un primo codice con una collective MPI_Barrier, in cui:

- 1 tutti i **processi** diversi dal processo 0 scrivono il proprio rank
- 2 la barriera sincronizza i processi
- 3 **POI** il processo 0 scrive il proprio rank.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank;
    int size;
    MPI_Init(NULL, NULL); // Inizializza
    MPI_Comm_size(MPI_COMM_WORLD, &size); // dimensione del world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // rank del processo

    MPI_Barrier(MPI_COMM_WORLD); // Sincronizza i processi

    MPI_Finalize(); // finalizza
}
```

una Collective semplice: MPI_Barrier soluzione

```

#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank;
    int size;
    MPI_Init(NULL, NULL);           // Inizializza
    MPI_Comm_size(MPI_COMM_WORLD, &size); // dimensione del world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // rank del processo

    if (rank != 0) {
        printf("Hello world from rank %d out of %d processes\n",
            rank, size);           // scrivi il mio rank
    }
    MPI_Barrier(MPI_COMM_WORLD); //----- BARRIERA-----
    if (rank == 0) {
        printf("Hello world from rank %d out of %d processes\n",
            rank, size);           // scrivi il mio rank
    }
    MPI_Finalize(); // finalizza
}

```

Domanda: succede qualcosa di strano? chi dovrebbe scrivere il proprio nome per ultimo? succede proprio così (questa va provata su una macchina in cui possano girare davvero n processi fisici)?

con questa funzione un processo (chiamato spesso **root**), manda lo **stesso** messaggio a **tutti** i processi (del communicator).

- e' una **collective** communication
- per questo e' sincronizzata
- **tutti** i processi devono chiamare MPI_Broadcast non **SOLO** il **root** (altrimenti si genera un **deadlock**).
- Per esempio supponiamo che il root sia il processo 0, ma la funzione broadcast venga chiamata prima dal processo 1. Questo "aziona" la funzione. Il processo 1 diventa **idle** e pronto a ricevere i dati che devono essere spediti dal processo 0.
- i dati vengono inviati quando il processo **root** chiama la funzione
- i dati sono ricevuti quando i singoli processi chiamano la funzione. Se solo un processo non dovesse chiamare la funzione, tutti gli altri riceverebbero i dati e si fermerebbero ad aspettare anche l'ultimo, creando il **deadlock**.

MPI_Bcast 2

```
int MPI_Bcast (
void          *nomeArray,
int          quantiDati,
MPI_Datatype tipoDati,
int          rankRoot,
MPI_Comm     nomeCommunicator
);
```

- Chiaramente l'unico rank da conoscere e' quello del **root** dato che tutti gli altri processi sono equivalenti per il broadcast.
- come fanno a ricevere le informazioni gli altri processi? semplicemente chiamando `MPI_Bcast` che e' quindi una funzione di **INVIO** da parte del root e di **RICEZIONE** da parte di tutti i processi.
- dove vengono stoccate le informazioni? nell'oggetto definito al primo ingresso della chiamata (in questo esempio in `nomeArray`)
- ogni processo puo' chiamare la funzione usando un **differente array** dove stoccare le informazioni (occhio che questo puo' indurre in errore, in quanto si devono porre degli `if` davanti alla chiamata del `MPI_Bcast`; se anche solo uno dei processi non chiama correttamente la funzione il sistema va in **deadlock**).

Esempio di MPI_Bcast

Proviamo a fare un semplice broadcast a tutti i processi, e poi a fare scrivere a video quello che i processi hanno ricevuto.

```

#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
    int tutti, rank;
    int root; // solo per ricordare definiamo una var per il root
    int array[2]; // definiamo un array con 2 posti
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &tutti);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    root = 0; // decidiamo che root=> rank=0
    if (rank==root) array[0]=0, array[1]=1; // solo il root ha i dati all'inizio
    if (rank!=root) array[0]=3, array[1]=3; // gli altri sono diversi
    // inserire comandi MPI
    //
    MPI_Finalize();
}

```

Esempio di MPI_Bcast soluzione

Proviamo a fare un semplice broadcast a tutti i processi, e poi a fare scrivere a video quello che i processi hanno ricevuto.

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
    int tutti, rank;
    int root;
    int array[2];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &tutti);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    root = 0;
    if(rank==root) array[0]=0, array[1]=1;
    if(rank!=root) array[0]=3, array[1]=3; // gli altri sono diversi
    MPI_Bcast (array, 2, MPI_INT, root, MPI_COMM_WORLD); // tutti devono chiamarlo
    printf("%d: array[0]=%d array[1]=%d\n", rank, array[0], array[1]);
    MPI_Finalize();
}
```

Differenze tra Broadcast e Scatter

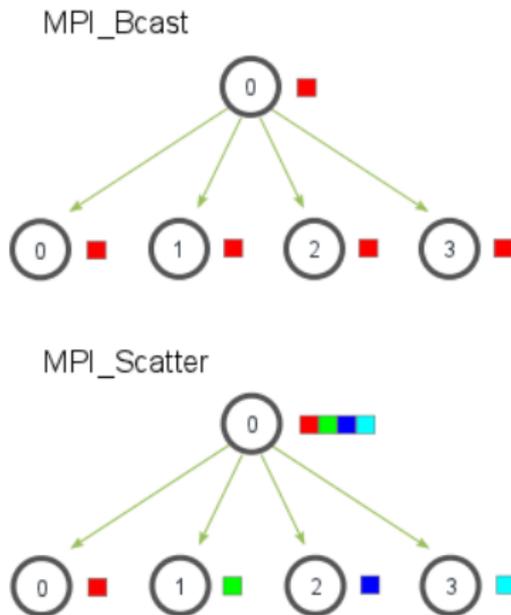


Figure: diagramma della differenza tra Broadcast e Scatter (da mpitutorial.com)

MPI_Scatter

MPI_Scatter manda dati **differenti** a processi diversi (mentre MPI_Broadcast manda gli **stessi** dati a tutti i processi).

- e' una collective communication.
- dato che e' collective e' sincronizzata.
- dato che e' sincronizzata **tutti** i processi devono chiamarla, altrimenti il sistema va in **deadlock**.
- attenzione che tutti i processi ricevono lo stesso numero di **messaggi elementari** (p.es. se vengono mandati 3 interi a tutti i processi, ogni intero puo' essere considerato come un **messaggio elementare**)
- i dati elementari sono tra loro **contigui** nell'area di memoria
- nei processi che non sono il root si puo' mettere un NULL all'interno del `send_data` (che quindi non deve essere nemmeno definito, **risparmiando memoria**).

la sintassi e' la seguente:

```

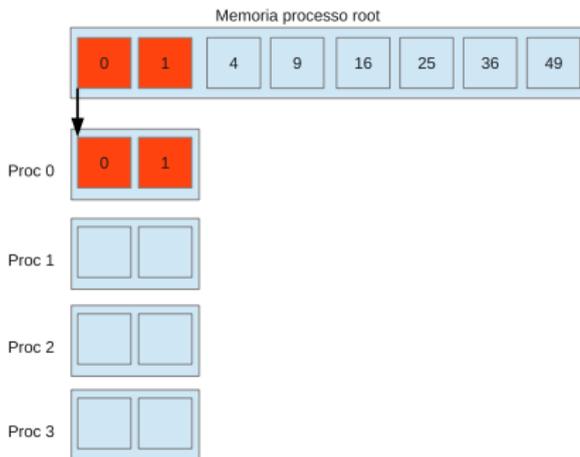
MPI_Scatter(
    void* send_data,           // array di dati che risiede nel processo root
    int send_count,          // quanti elementi mandati ad ogni processo dall'array
    MPI_Datatype send_datatype, // tipo dei dati mandati
    void* recv_data,         // indirizzo del processo che riceve i dati
    int recv_count,          // numero di dati che entrano nell'array
    MPI_Datatype recv_datatype, // tipo dei dati ricevuti
    int root,                // il rank del root
    MPI_Comm communicator)   // il communicator
  
```

MPI_Scatter 2

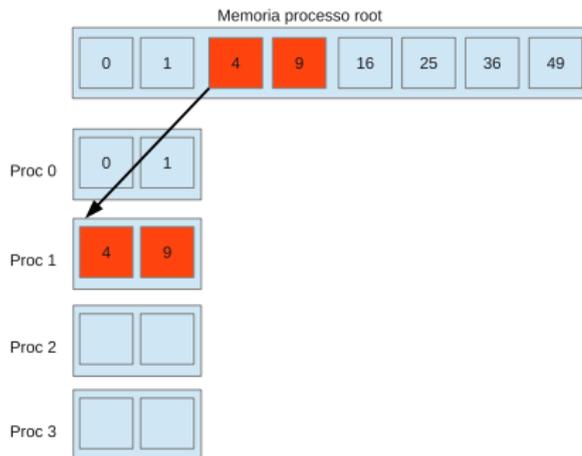
Ha senso fare un riassunto su come si deve interpretare il funzionamento di `MPI_Scatter`, per fugare eventuali dubbi, specifichiamo il **round robin**:

- Se c'è un array di `int` e `send_count=1`, allora al processo 0 viene mandato il **primo** intero dell'array, al processo 1 il **secondo** elemento dell'array, etc...
- Se `send_count=2` allora al processo 0 arrivano il **primo** e il **secondo** elemento dell'array, al processo 1 il terzo e quarto, etc... In pratica spesso:
`send_count = dim(send_data) / numero_processi` (in questo modo ad tutto un array viene equamente suddiviso tra tutti i processi).
- **Problema di flessibilità**: Se voglio inviare 3 unità elementari al processo 1 e 4 unità al processo 2 come faccio?
- **Problema di portabilità**: ad ogni processo vengono inviati **esattamente** `send_count` messaggi elementari... se `N` è il numero di processi, allora il nostro array deve contenere almeno $N \times \text{send_count}$ ingressi... altrimenti cosa succede? **deadlock**? no in questo caso il computer agisce come al solito, ergo come una capra. Continua a prendere informazione nell'area di memoria contigua...ergo non dà errore, ma invia dei messaggi **sbagliati** ai processi!

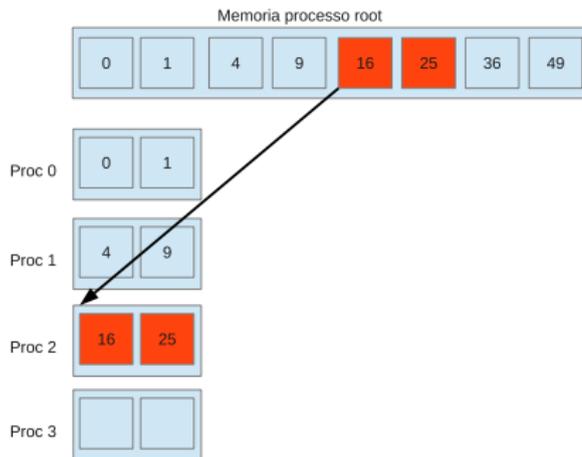
MPI_Scatter visuale



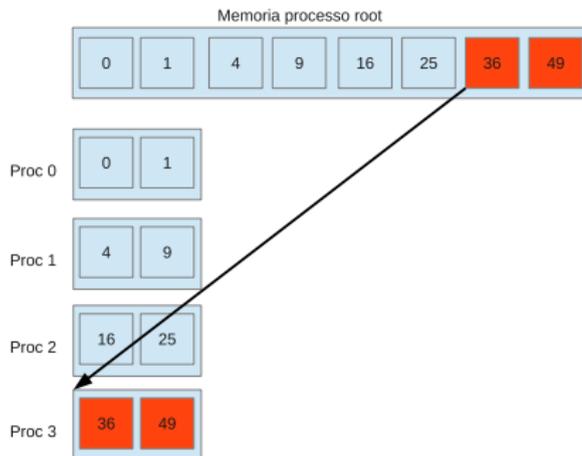
MPI_Scatter visuale



MPI_Scatter visuale



MPI_Scatter visuale



Esercizio con MPI_Scatter

- costruiamo un array con 100 elementi (chiamato per esempio vettPartenza), solo per il rank 0, popolato di valori da 0 a 99
- tutti gli altri processi devono avere a disposizione un array (chiamato per esempio vettRicezione) di 10 elementi, azzerato
- utilizziamo `MPI_Scatter` per inviare a ciascun rank 10 oggetti dell'array del rank 0
- lanciamo il codice su 10 processi
- facciamo scrivere i primi 3 elementi dei vettRicezione per ogni processo

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, i;           //
    int conto;
    int Ngrande = 100, Npiccolo = 10;
    int vettPartenza[Ngrande], vettRicezione[Npiccolo]; // vettori di partenza e ricezione

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size); //
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //

    if (rank == 0) for( i =0; i<Ngrande; i++) vettPartenza[i]=i; // interi
    if (rank != 0) for( i =0; i<Npiccolo; i++) vettRicezione[i] = 0 ; // azzerare il vettore
                    di arrivo

    //                CODICE DA INSERIRE
    //                CODICE DA INSERIRE

    MPI_Finalize();
    return 0;
}

```

Esercizio con MPI_Scatter: soluzione

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size, i;           //
    int conto;
    int Ngrande = 100, Npiccolo = 10;
    int vettPartenza[Ngrande], vettRicezione[Npiccolo];    // vettori di partenza e ricezione

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);    //
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);    //

    if (rank == 0) for( i =0; i<Ngrande; i++) vettPartenza[i]=i;    // interi
    if (rank != 0) for( i =0; i<Npiccolo; i++) vettRicezione[i] = 0 ;    // azzerà il vettore
        di arrivo

    MPI_Scatter(vettPartenza, Npiccolo, MPI_INT, vettRicezione, Npiccolo , MPI_INT, 0,
    MPI_COMM_WORLD);    // fa un broadcast

    printf("rank = %d dati ricevuti: %d, %d, %d \n", rank, vettRicezione[0],
    vettRicezione[1], vettRicezione[2]);

    MPI_Finalize();
    return 0;
}

```

- Cosa succede se lancio il codice su 11 processi?
- Cosa succede se modifico il codice mettendo Ngrande = 99 (invece che 100)?

MPI_Gather

Questa funzione e' l'inversa di Scatter, ha la medesima sintassi e struttura:

```

MPI_Gather(
    void* send_data,           // array di dati che risiede nel processo che invia
    int send_count,           // quanti elementi mandati da ogni processo al root
    MPI_Datatype send_datatype, // tipo dei dati mandati
    void* recv_data,         // buffer del processo (root) che riceve i dati
    int recv_count,          // numero di dati che entrano nell'array
    MPI_Datatype recv_datatype, // tipo dei dati ricevuti
    int root,                // indica il rank del root
    MPI_Comm communicator)   // nome del communicator
  
```

E' utile per quando si vogliono raggruppare in un unico processo dei dati calcolati in vari processi.

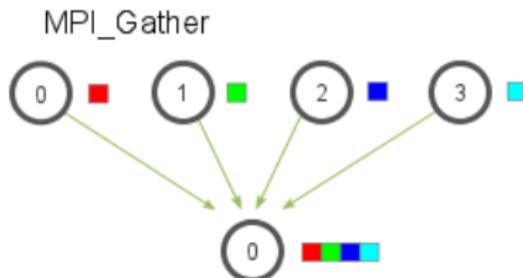


Figure: mpitutorial.com

MPI_Gather 2

- e' l'**inverso** di `MPI_Scatter`
- e' una **collective** communication
- dato che e' una collective communication e' **sincronizzata**
- dato che e' sincronizzata **TUTTI** i processi devono chiamarla!
- occhio che si possono passare argomenti diversi per ogni processo. In particolare il buffer dei dati da ricevere **serve solo per il processo root**, quindi per esempio posso definirlo solo per quel processo (in questo modo se ho 10^5 processi, risparmio un bel po' di memoria totale facendo definire solo al root l'oggetto voluto).
- Bisogna comunque inserire un argomento che indichi dove si vogliono ricevere i dati: in C si puo' assegnare il tipo NULL

MPI_Scatterv:

- I dati spediti da `MPI_Scatter` devono essere **contigui**
- C'e' un problema di **scalabilita'**, per esempio:
 - A e' un array di 10 interi
 - ci sono 3 processi
 - devo inviare 4 interi per ogni array
 - all'ultimo processo `MPI_Scatter` vuole inviare 4 interi (dal 8 all'11 incluso):
peccato che, al massimo, l'array A contiene l'ingresso 9...

La risposta ad alcuni dei problemi di `MPI_Scatter` si chiama: `MPI_Scatterv`.
Per spiegazione accurata cercare gli appunti della [Cornell](#):

<https://cvm.cac.cornell.edu/MPICc/scatterv?AspxAutoDetectCookieSupport=1>

Utilizzo Scatterv

Differenze rispetto a `MPI_Scatter`:

- si introduce un nuovo **array** che contiene il numero di messaggi elementari da inviare ad ogni processo (qui sotto lo chiamiamo `scounts`).
- si introduce un nuovo **array** (chiamato `displs` qui sotto) che contiene la posizione (nell'array di partenza) da dove prendere i dati da inviare al dato processo.

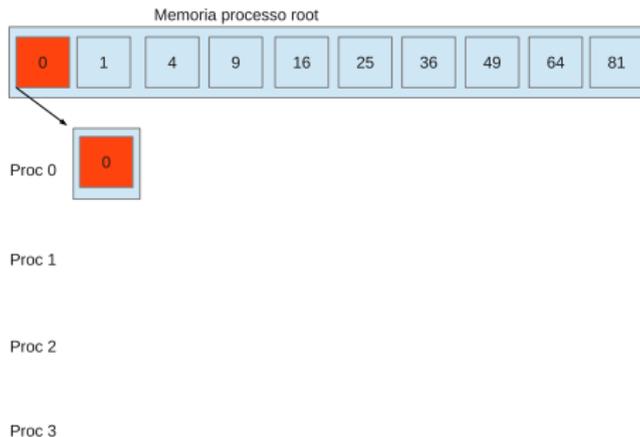
chiaramente, questi due array devono contenere un numero di ingressi uguale (o maggiore) al numero di processi coinvolti.

```
int MPI_Scatterv(
void*      sbuf, // puntatore/indirizzo dell'array (o dell'oggetto da mandare)
int        scounts[], // Array intero con il numero di elementi da mandare ad ogni processo
int        displs[], // Array che specifica lo spostamento RELATIVO a sbuf dal quale prendere
            // i dati da mandare al processo corrispondente
MPI_Datatype stype, // tipo del dato da mandare
void*      rbuf, // indirizzo dove vanno mandati i dati nel processo che riceve
int        rcount, // numero di elementi nel receive buffer e' un INTERO
MPI_Datatype rtype, // tipo dei dati da ricevere
int        root, // rank del root
MPI_Comm   comm) // communicator
```

MPI_Scatterv visuale

Supponiamo che ci siano 4 processi e:

```
counts={1,3,2,1} // numero di elementi inviati ad ogni processo  
displs = {0,7,2,5} // posizione nell'array da dove si prendono gli elementi
```

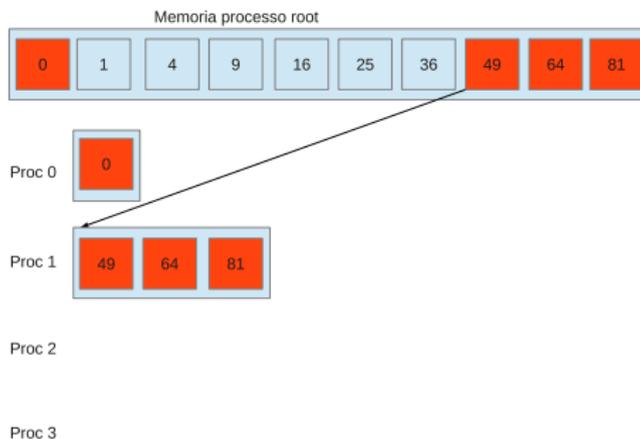


MPI_Scatterv visuale

Supponiamo che ci siano 4 processi e:

```

counts={1,3,2,1} // numero di elementi inviati ad ogni processo
displs ={0,7,2,5} // posizione nell'array da dove si prendono gli elementi
  
```

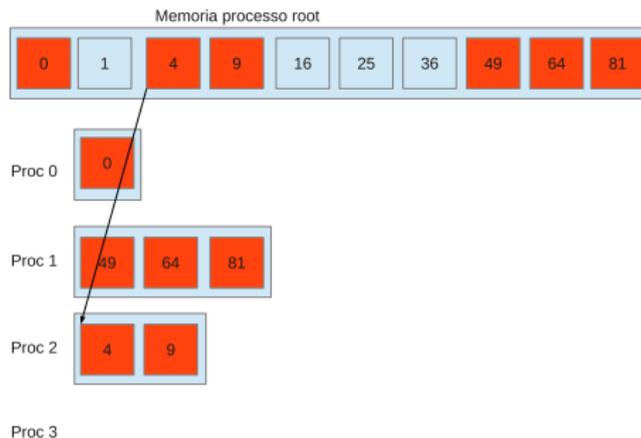


MPI_Scatterv visuale

Supponiamo che ci siano 4 processi e:

```

counts={1,3,2,1} // numero di elementi inviati ad ogni processo
displs ={0,7,2,5} // posizione nell'array da dove si prendono gli elementi
  
```

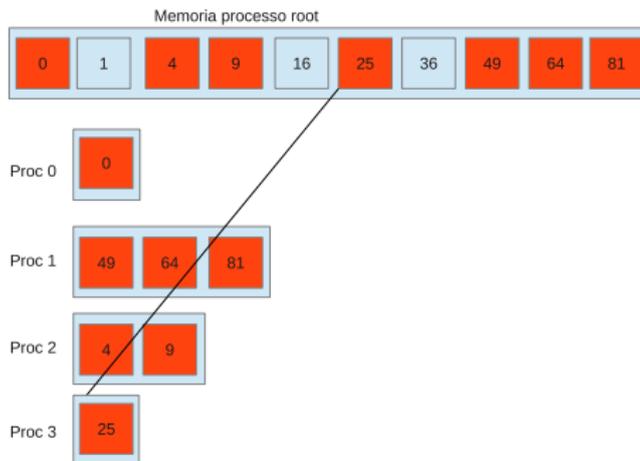


MPI_Scatterv visuale

Supponiamo che ci siano 4 processi e:

```

scounts={1,3,2,1} // numero di elementi inviati ad ogni processo
displs = {0,7,2,5} // posizione nell'array da dove si prendono gli elementi
  
```



Esempio di `MPI_Scatterv`

consideriamo una implementazione di `MPI_Scatterv()` con le seguenti caratteristiche

- viene costruito un grande array da cui inviare i dati che, in ogni ingresso, contiene il valore della posizione: `sbuf[n]=n`, con $n=0, 1, \dots, N_{\text{grande}}$
- ogni processo contiene un array piccolo (`rbuf`) dove mettere i pezzi dell'array grande.
- viene inviato un messaggio ad **ognuno** dei processi del calcolo (scalabilità)
- per ogni processo lo "spostamento" deve essere una funzione del `rank`, per esempio: `spostamento[rank]=rank%3+2` (prendo il resto della divisione per 3 del rank e aggiungo 2).
- ogni processo scrive a video quello che ha ricevuto

Esempio di MPI_Scatterv risposte

Domande:

- chi (quale processo) deve avere definito l'array da cui vengono inviati i dati `sbuf`? **root**
- chi deve avere l'array dove vengono ricevuti i dati `rbuf`? **ognuno il suo**
- chi deve avere l'array che definisce lo spostamento (displacement) `displs`? **root**
- chi deve avere l'array che definisce il numero di oggetti mandati ad ogni processo `scounts`? **root**
- perche' `rcount` (il numero di oggetti ricevuti) e' un intero e non un array? perche' e' presente in **ogni processo** ricevente

Esempio di MPI_Scatterv 2

```

int* sendCount; // ptr array con quanti oggetti elementari sono inviati ad ogni processo
int* displs; // ptr array con lo spostamento per ogni processo
int receiveCount; // quanti ingressi devo ricevere
int sbuf[Ngrande];

sendCount = (int*)malloc(size*sizeof(int));
displs = (int*)malloc(size*sizeof(int));

//
// sbuf[i]=i // quale processo ha bisogno di questo array? i=0,..,Ngrande
// sendCount[i]=i%3+2 // " " i=0,..,size-1
// displs[i]=i // " " i=0,..,size-1
// rcount // quale processo ha bisogno di questo intero?

```

Qui si ha uno spreco di memoria, tutti i processi hanno tutti gli array, tutti i processi chiamano MPI_Scatterv in questo modo:

```

MPI_Scatterv(sbuf, // puntatore al vettore di partenza (esiste solo in 0)
            sendCount, // puntatore all'array con il numero di elementi mandati per ogni processo
            displs, // puntatore all'array che contiene lo spostamento (da 0) per mandare
                elemento per ogni processo
            MPI_INT, // tipo inviato
            rbuf, // puntatore all'array di ricezione (contenuto in ogni processo)
            receiveCount, // numero di elementi da ricevere, non e' un array ma solo un intero
            MPI_INT, // tipo di ricezione
            0, // rank del root
            MPI_COMM_WORLD ); // communicator

```

Esempio di MPI_Scatterv 3

Evitiamo gli sprechi: passo gli argomenti solo a chi li usa! (ovviamente se un processo non usa un array, puo' essere utile evitare di definire quell'array per quel processo!)

```

if (rank==0) {
MPI_Scatterv(sbuf, // puntatore al vettore di partenza (esiste solo in 0)
            sendCount, // puntatore all'array con il numero di elementi mandati per ogni processo
            displs, // puntatore all'array che contiene lo spostamento per ogni processo
            MPI_INT, // tipo inviato
            rbuf, // puntatore all'array di ricezione
            receiveCount, // numero di elementi da ricevere (non e' un array!)
            MPI_INT, // tipo di ricezione
            0, // rank del root
            MPI_COMM_WORLD ); // comunicatore
}
if (rank!=0) {
MPI_Scatterv(NULL,
            NULL,
            NULL,
            NULL,
            rbuf,
            receiveCount, // numero di elementi da ricevere
            MPI_INT, // tipo di ricezione
            0, // rank del root
            MPI_COMM_WORLD); // comunicatore
}

```

MPI_Gatherv

E' la versione corrispondente a `MPI_Scatterv`, dove pero' il processo **root** riceve messaggi dagli altri **processi** all'interno del communicator

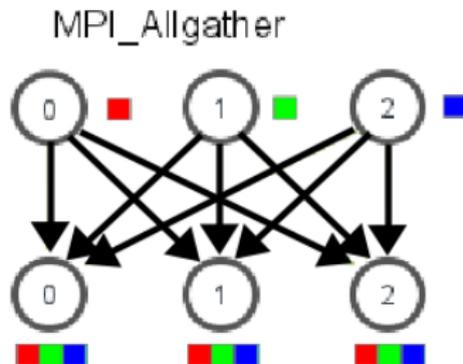
```
int MPI_Gatherv( void*      sbuf,
                 int        scount,
                 MPI_Datatype stype,
                 void*      rbuf,
                 const int   rcounts[],
                 const int   displs[],
                 MPI_Datatype rtype,
                 int         root,
                 MPI_Comm    comm )
```

In questo caso quindi: `rbuf` dovra' essere piu' grande di `sbuf` dato che deve contenere tutti i messaggi ricevuti...

MPI_Allgather

MPI_Allgather e' l'equivalente di un MPI_Gather e un MPI_Bcast insieme. La funzione prende un insieme di dati elementari **da ogni processo**, le riunisce (gather) e poi manda l'insieme di tutte le info **a tutti** i processi (equivalentemente e' come se ogni processo facesse un gather). La differenza con un MPI_Allgather e' di **ottimizzazione**, ovvero come vengono inviate le cose. Questo e' il "bello" delle funzioni MPI: fanno il lavoro sporco di ottimizzazione per mandare i dati ai vari processi (altrimenti basterebbero le poche funzioni di base come MPI_Send e MPI_Recv).

- La struttura e' la stessa di un MPI_Gather, ma **non necessita** di un argomento che indichi il **root**: non ha senso definire un root, sara' MPI a decidere dove e' piu' conveniente inviare i dati da raccogliere, tanto alla fine tutti i processi riceveranno l'informazione completa (mpitutorial.com):



MPI_Allgather: dettaglio

```
int MPI_Allgather (
    void * sendbuf , // puntatore all'array di invio
    int sendcount , // numero di oggetti elementari inviato da ogni processo
    MPI_Datatype sendtype , // tipo
    void * recvbuf , // puntatore all'array di ricezione
    int recvcnt , // numero di oggetti elementari da ricevere da ogni processo
    MPI_Datatype recvtype , // tipo
    MPI_Comm comm ) // quale communicator
```

Si noti che i dati vengono raccolti in `recvbuf` che dovrà avere abbastanza spazio per stocarli, ricordandosi che tutti i processi inviano `sendcount` dati elementari

```
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

Esercizio con allgather

Proviamo ad usare `MPI_Allgather`, facendo costruire un array piu' piccolo contenuto in ogni processo ed inviando una parte di questo array a **tutti** i processi in modo da costruire un array piu' grande presente in ogni processo.

```

#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{ int size, rank, N=5;
  int partenza[N];
  int i, ngrande, quanti = 3;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (i=0; i<N; i++)  partenza[i]=rank;           // inizializza il vettore di partenza

  ngrande=quanti*size;                               // dimensione dell'array di arrivo
  int arrivo[ngrande];                               // allocate the gathering buffer

  // codice da inserire
  // codice da inserire
  // codice da inserire

  MPI_Finalize();
}

```

Esercizio con allgather soluzione

```

#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{ int size, rank, N=5;
  int partenza[N];
  int i, ngrande, quanti = 3;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (i=0; i<N; i++)  partenza[i]=rank;           // inizializza il vettore di partenza

  ngrande=quanti*size;                               // dimensione dell'array di arrivo
  int arrivo[ngrande];                               // allocate the gathering buffer

  MPI_Allgather(partenza, quanti, MPI_INT, arrivo, quanti, MPI_INT, MPI_COMM_WORLD);

  printf("rank = %d: ", rank);                       // scrivi il rank
  for (i=0; i<ngrande; i++) printf("%d ", arrivo[i]);
  printf("\n");                                       // dopo l'ultimo vai a capo

  MPI_Finalize();
}

```

Attenzione una funzione di questo tipo puo' essere utile quando si fanno delle moltiplicazioni tra matrici.

MPI_Alltoall

`MPI_Alltoall` e' una funzione:

- collective
- equivalente ad n send e n receive (dove n e' il numero di processi) per ogni processo
- ogni processo **manda** dei messaggi **diversi** a tutti i processi
- ogni processo **riceve** dei messaggi **diversi** da ognuno dei processi

Alcune osservazioni:

- e' molto utile per alcuni algoritmi paralleli come: Fast Fourier Transform
- richiede molta banda
- Questa funzione manda messaggi contigui a processi contigui (e' poco flessibile, come si chiamera' la versione "piu' libera" che consente di inviare messaggi non contigui?)

MPI_Alltoall 2

```
int MPI_Alltoall(  
    void *sendbuf, // quanti sendbuf ci sono?  
    int sendcount, // quanti messaggi invio (a chi?)  
    MPI_Datatype sendtype, // il loro tipo  
    void *recvbuf, // dove ricevo messaggi  
    int recvcount, // quanti ne ricevo (da chi?)  
    MPI_Datatype recvtype, // tipo di ricezione  
    MPI_Comm comm) // comunicatore
```

Qualche domanda:

- se sono un processo, quanti messaggi elementari invio in tutto, se sendcount=2?
- quanti messaggi riceve ogni processo?
- dove deve essere inizializzato sendbuf? e recvbuf?

MPI_Alltoall tre

(esempio preso da http://www.math-cs.gordon.edu/courses/cps343/presentations/MPI_Collective.pdf)

Supponiamo che ci siano 4 processi, ognuno dei quali con degli array come mostrato sotto, dopo una all-to-all di questo tipo:

```
MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);
```

i dati saranno distribuiti come segue (si vedono gli MPI_Scatter?):

<i>Invio</i>									<i>Ricezione</i>							
(01	02	03	04	05	07	07	08)	rank 0	(01	02	11	12	21	22	31	32)
(11	12	13	14	15	16	17	18)	rank 1	(03	04	13	14	23	24	33	34)
(21	22	23	24	25	26	27	28)	rank 2	(05	06	15	16	25	26	35	36)
(31	32	33	34	35	36	37	38)	rank 3	(07	08	17	18	27	28	37	38)

Occhio: dallo schema di cui sopra si nota che il numero di messaggi elementari da inviare deve essere ben commisurato con il numero di processi attivi, ergo se l'array di invio contiene 8 messaggi e ci sono 4 processi allora ne posso inviare 2 (come in questo caso) o anche 1 (e quindi l'array di ricezione sarà 'piu' piccolo di quello di invio).

MPI_Alltoall quattro

Supponiamo che ci siano 4 processi, ognuno dei quali con degli array come mostrato sotto, dopo una all-to-all di questo tipo:

```
MPI_Alltoall(u, 2, MPI_INT, v, 2, MPI_INT, MPI_COMM_WORLD);
```

Qui vediamo bene il `MPI_Gather()` nascosto nell `MPI_Alltoall()`:

<i>Invio</i>									<i>Ricezione</i>							
(01	02	03	04	05	07	07	08)	<i>rank 0</i>	(01	02	11	12	21	22	31	32)
(11	12	13	14	15	16	17	18)	<i>rank 1</i>	(03	04	13	14	23	24	33	34)
(21	22	23	24	25	26	27	28)	<i>rank 2</i>	(05	06	15	16	25	26	35	36)
(31	32	33	34	35	36	37	38)	<i>rank 3</i>	(07	08	17	18	27	28	37	38)

MPI_Alltoall: equivalenza

Esercizio:

proviamo a scrivere un pezzo di codice che sia equivalente ad un `MPI_Alltoall` ma composto da solo `MPI_Send` e `MPI_Recv`

- Ogni processo deve mandare piu' messaggi. Quanti cicli servono ad ogni messaggio?
- Ogni Send (e anche receive) avra' un `sendbuf` diverso... dobbiamo spostare il punto di partenza da cui si inviano i messaggi: di quanto?
- Il tempo di esecuzione sara' il medesimo che un `MPI_Alltoall`? perche'?

```
for (i = 0, i < n; i++)
    MPI_Send(sendbuf + i * sendcount * extent(sendtype),
             sendcount, sendtype, i, ..., comm);
for (i = 0, i < n; i++)
    MPI_Recv(recvbuf + i * recvcnt * extent(recvtype),
             recvcnt, recvtype, i, ..., comm);
```

Esempio di MPI_Alltoall

proviamo a fare un codice con MPI_Alltoall

```

#include <stdio.h>
#include <mpi.h>
#define N 8 // dimensione array
int main (int argc, char *argv[])
{ int size, rank;
  int i, m, arrayInvio[N], arrayRicezione[N];
  MPI_Status status; // stato ricezione
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (i=0; i<N; i++) arrayInvio[i]=(rank+1)*10+i; // il rank 1 ha 10,11, 12.. il rank 2 ha
  20,21,...
  printf("il processo %d contiene:", rank);
  for (i=0; i<N; i++) printf("%d ", arrayInvio[i]);
  printf("\n"); // vai a capo dopo di ogni scrittura

  INSERIRE codice
  INSERIRE codice
  INSERIRE codice

  printf("il processo %d adesso ha :", rank); // non vado a capo
  for (i=0; i<N; i++) printf("%d ", arrayInvio[i]); // scrivo i valori e uno spazio
  printf("\n");

  MPI_Finalize();
}

```

Esempio di MPI_Alltoall: soluzione

```

#include <stdio.h>
#include <mpi.h>
#define N 8 // dimensione array
int main (int argc, char *argv[])
{ int size, rank;
  int i, m, arrayInvio[N], arrayRicezione[N];
  MPI_Status status; // stato ricezione
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for (i=0; i<N; i++) arrayInvio[i]=(rank+1)*10+i; // il rank 1 ha 10,11, 12.. il rank 2 ha
  20,21,...
  printf("il processo %d contiene:", rank);
  for (i=0; i<N; i++) printf("%d ", arrayInvio[i]);
  printf("\n"); // vai a capo dopo di ogni scrittura

  m = N/size; // messaggi inviati = dimensione array/numero processi
  MPI_Alltoall(arrayInvio, m, MPI_INT, // array inviato, m numero di elementi
              arrayInvio, m, MPI_INT, // posso usare lo stesso array!!!
              MPI_COMM_WORLD);

  printf("il processo %d adesso ha :", rank); // non vado a capo
  for (i=0; i<N; i++) printf("%d ", arrayInvio[i]); // scrivo i valori e uno spazio
  printf("\n");

  MPI_Finalize();
}

```

Reduce operations

Il **cuore** di un calcolo parallelo e' il fatto che i processi eseguono contemporaneamente molti calcoli,

- alla fine pero' potrei **non essere interessato** ad avere molti risultati (p.es 100 000 numeri)
- spesso devo **riassumerli** in quantita' che siano leggibili e chiare per una singola persona.

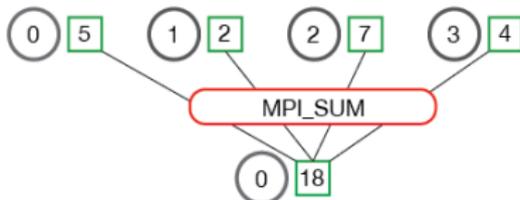
L'esempio classico e' suddividere una somma molto lunga in 100 000 somme parziali (circa 100 000 volte piu' corte). Alla fine devo sommare tra loro tutti i risultati parziali. Questo processo e' un processo di **riduzione** (riassunto) o **Reduce**. Il concetto e' banale in se', l'implementazione di un file system e un insieme di comandi che permettano di eseguire questa cosa invece non e' affatto semplice. Nel caso di MPI si ha:

```
int MPI_Reduce(
    void *sendbuf,           // array di elementi da tutti i processi
    void *recvbuf,          // array di root (sizeof(datatype)* count) risultati ridotti
    int count,              // numero di messaggi elementari mandati da OGNI processo
    MPI_Datatype datatype,  // tipo dei dati mandati dai processi
    MPI_Op op,              // quale operazione di riduzione viene fatta
    int root,               // a quale processo arrivano i dati che poi riduce
    MPI_Comm comm)         // communicator di riferimento
```

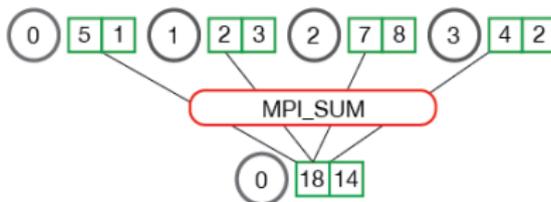
Reduce 2

Una **nota** sulla dimensione di `recvbuf` ovvero l'array che contiene i risultati della riduzione (e in qualche modo e' l'array piu' importante). La sua dimensione e' quella di `sizeof(datatype) * count` dove ricordo che `count` e' il numero di dati mandati da OGNI processo. Quindi se i processi mandano 2 dati ciascuno, l'operazione di riduzione viene effettuata su 2 insiemi di dati **separatamente**.

MPI_Reduce



MPI_Reduce



Reduce: un'osservazione

Abbiamo visto che uno degli ingressi della funzione `MPI_Reduce()` è il **rank** del **root**, dove viene inserito il risultato dell'operazione di riduzione.

- **Osservazione:** solo **alcuni tipi** di operazioni possono diventare delle reduce. Viene richiesto che l'operatore di riduzione \mathcal{O} goda della proprietà **associativa**:
 $\forall x, y, z \quad (x\mathcal{O}y)\mathcal{O}z = x\mathcal{O}(y\mathcal{O}z)$
- **Domanda:** in quale processo viene fatta l'operazione di riduzione?
- **Risposta:** la riduzione viene gestita da MPI e viene fatta durante la propagazione delle informazioni tra i processi. In questo modo la riduzione sfrutta il parallelismo, in modo implicito (il programmatore non lo controlla).
- **Esempio con la somma:**

rank	0	1	2	3
valore iniziale	5	2	7	4
valore intermedio	7	/	\	11
valore ridotto	18			

Quali riduzioni?

Quali sono le tipiche operazioni di riduzione eseguite? abbiamo visto la somma, ma c'e' anche il prodotto, oppure il massimo....

```

MPI_MAX    // Restituisce l' elemento massimo.
MPI_MIN    // Restituisce l' elemento minimo.
MPI_SUM    // Somma gli elementi.
MPI_PROD   // Moltiplica gli elementi.
MPI_LAND   // un "and" logico tra gli elementi.
MPI_LOR    // un "or" logico tra gli elementi.
MPI_BAND   // un "bitwise and" logico tra gli elementi.
MPI_BOR    // un "bitwise or" logico tra gli elementi.
MPI_MAXLOC // restituisce il massimo e il rank del processo che ha il massimo
MPI_MINLOC // restituisce il minimo e il rank del processo che ha il minimo.
  
```

E se avessi bisogno d'altri tipi di funzioni di riduzione, cosa faccio?

Uso la funzione: `MPI_OP_CREATE` per creare una handle ad una funzione scritta da me.

Attenzione che una operazione di riduzione si suppone che sia **associativa** (con delle flag si puo' indicare anche se e' **commutativa**).

Proprieta' associativa di un operatore \mathcal{O} : dati 3 elementi x, y e z su cui \mathcal{O} puo' operare (e' una operazione binaria):

$$(x\mathcal{O}y)\mathcal{O}z = z\mathcal{O}(x\mathcal{O}y) \quad (5)$$

Reduce esempio

- scrivere un codice dove ogni processo ha un array di dimensione 2 contenente, nell'ingresso 0 il proprio rank, e nell'ingresso 1 la quantità $\text{rank} \times 2$
- usare `MPI_Reduce` per ottenere la somma di tutti gli elementi 0 di ogni array e anche di tutti gli elementi 1 di ogni array

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size; //
    int vettPartenza[2], vettRicezione[2]; // vettori di partenza e ricezione
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size); //
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //

    //CODICE DA INSERIRE: azzera vettore dove si mette risultato riduzione
    //CODICE DA INSERIRE: assegna i valori da cui fare riduzione
    //MPI_Reduce ( daDoveInvio, aDoveInvio, quanti_oggetti, tipo, riduzione, rank, communicator);

    if (rank == 0 )
        printf("rank = %d Riduzioni: %d, %d \n", rank, vettRicezione[0], vettRicezione[1]);

    MPI_Finalize();
    return 0;
}

```

Reduce esempio: soluzione

```

#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size; //
    int vettPartenza[2], vettRicezione[2]; // vettori di partenza e ricezione
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size); //
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //

    if (rank == 0) {
        vettRicezione[0]=0; // azzera vett dove si fa riduzione
        vettRicezione[1]=0;
    }

    vettPartenza[0] = rank ; //
    vettPartenza[1] = 2* rank ; //

    MPI_Reduce ( vettPartenza, vettRicezione, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0 )
        printf("rank = %d Riduzioni: %d, %d \n", rank, vettRicezione[0], vettRicezione[1]);

    MPI_Finalize();
    return 0;
}

```

- Cambiare la funzione di riduzione con `MPI_MAX` e `MPI_MIN` e vedere il risultato

MPI_Sendrecv

- E' una **point to point** operation, in cui si aprono 2 “canali” uno per inviare e uno per ricevere.
- Questa funzione racchiude in se stessa sia un send e un receive
- **Attenzione:** **non** e' un send **seguito** da un receive, ma i canali procedono concorrentemente (il loro ordine non e' noto)
- l'array di partenza e di arrivo **devono** essere differenti (e possono avere una forma differente, per esempio il primo e' un array con 100 ingressi e il secondo e' un array con solo 10 ingressi).
- un messaggio inviato da una `MPI_Sendrecv` puo' essere **ricevuto** da un semplice `MPI_Recv`
- puo' essere usato per inviare un messaggio da un processo a se stesso.

MPI_Sendrecv 2

```
int MPI_Sendrecv(  
    void          *sendbuf,          // dove sono le cose da mandare  
    int           sendcount,        // quanti messaggi elementari invio  
    MPI_Datatype  sendtype,        // il loro tipo  
    int           dest,            // rank del destinatario  
    int           sendtag,         // tag associato al messaggio  
    void          *recvbuf,        // dove metto le cose in ricezione  
    int           recvcount,       // quanti messaggi elementari ricevo  
    MPI_Datatype  recvtype,       // il loro tipo  
    int           source,         // chi me li manda  
    int           recvtag,        // con che tag  
    MPI_Comm      comm,          // in che communicator agiamo?  
    MPI_Status    *status)       // lo status della ricezione
```

MPI_Sendrecv 3

Prendiamo il primo codice scritto con MPI_Send e MPI_Recv e modifichiamolo con MPI_Sendrecv

```

#include <stdio.h>
#include <mpi.h>
int main (int argc, char * argv[]) {
    int nproc, rank;
    int Narray = 2;
    float da_inviare[Narray], ricevere[Narray];
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    ricevere[0]=0, ricevere[1]=0;
    if (rank==0) {
        da_inviare[0]=2, da_inviare[1]=4;                // scrivi l'array per il processo 0

        printf("%d: ricevuto[0]=%f ricevuto[1]=%f\n", rank, ricevere[0], ricevere[1]);
    }
    MPI_Finalize();
}

```

MPI_Sendrecv: soluzione

Prendiamo il primo codice scritto con MPI_Send e MPI_Recv e modifichiamolo con MPI_Sendrecv

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char * argv[]) {
    int nproc, rank;
    int Narray = 2;
    float da_inviare[Narray], ricevere[Narray];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ricevere[0]=0, ricevere[1]=0;
    if (rank==0) {
        da_inviare[0]=2, da_inviare[1]=4;           // scrivi l'array per il processo 0
        MPI_Sendrecv (da_inviare, Narray, MPI_FLOAT, 1, 10 // 10 e' il tag INVIATO
                     , ricevere, Narray, MPI_FLOAT, 1, 11 // 11 e' il tag da RICEVERE
                     , MPI_COMM_WORLD,&status);
    }
    else if (rank==1) {
        da_inviare[0]=1, da_inviare[1]=3;           // scrivi l'array per il processo 1
        MPI_Sendrecv(da_inviare, Narray, MPI_FLOAT, 0,11 // 11 e' il tag INVIATO
                    , ricevere, Narray, MPI_FLOAT, 0,10 // 10 e' il tag da RICEVERE
                    ,MPI_COMM_WORLD,&status);
    }
    printf("%d: ricevuto[0]=%f ricevuto[1]=%f\n", rank, ricevere[0], ricevere[1]);
    MPI_Finalize();
}
```

Moltiplicazione tra Matrici: ripasso

Righe per colonne:

$$AB = C \Rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

$$c_{23} = a_{21} \cdot b_{13} + a_{22} \cdot b_{23} + a_{23} \cdot b_{33}$$

Per ottenere l'elemento c_{23} si fa il prodotto scalare tra la **seconda riga** della matrice A e la **terza colonna** della matrice B.

FORMULA per il prodotto di matrici:

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

Per il prodotto tra due matrici $N \times N$ servono:

- $N \times N$ prodotti (riga \times colonna)
- ogni prodotto riga \times colonna richiede N prodotti (e $N - 1$ somme)

Un algoritmo MPI non ottimale, per la moltiplicazione di matrici, premesse:

```

#include <mpi.h> // inventati 2 matrici, fai il prodotto usando MPI
#include <stdio.h> // questo prodotto e' molto poco ottimizzato ma utile
#include <stdlib.h> // serve per exit
#define N 16 // dimensione delle matrici

//=====
int A[ N ][ N ], B[ N ][ N ], C[ N ][ N ], D[ N ][ N ];

void costruisci_matrice(int m[ N ][ N ])
{
    int n=0, i, j;
    for (i=0; i< N ; i++)
        for (j=0; j< N ; j++)
            m[i][j] = n++; // matrice con ingressi progressivi
}

//=====
void costruisci_identita2(int m[ N ][ N ]) // costruisci matrice identita' x 2
{
    int i, j;
    for (i=0; i< N ; i++)
        for (j=0; j< N ; j++){
            m[i][j]=0;
            if (i == j) m[i][j]=2; // cosi' almeno modifica un po' la matrice...
        }
}

```

funzione per la stampa a video una matrice

questa funzione serve per controllare i risultati

```
//=====
void print_matrix(int m[ N ][ N ])    //scrivi a video
{
    int i, j = 0;
    for (i=0; i< N ; i++) {
        printf("\n\t| ");
        for (j=0; j< N ; j++)
            printf("%3d ", m[i][j]);
        printf("|");
    }
}
//=====
```

Algoritmi per prodotto matrici $N \times N$

Algoritmo *Banale*:

supponiamo di avere a disposizione un numero di processi p che divida **esattamente** il numero di righe N ; per esempio con $N = 4$ e $p = 2$ (processi); occhio che gli indici usati qui sotto non vanno bene con il C, perché qui il primo elemento è l'1 (come per gli esseri umani o con il FORTRAN).

$$A \cdot B = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

- 1 costruisco le matrici A e B all'interno del processo 0 (e inizializzo a 0 la C)
- 2 invio ad ogni processo **tutta** la matrice B (per esempio con `MPI_Bcast`)
- 3 invio ad ogni processo N/p righe di A (per esempio con `MPI_Scatter`)
- 4 faccio i prodotti scalari delle righe di A contenute in ogni processo per B
- 5 raccolgo gli elementi di matrice (con `MPI_Gather`) presenti in ogni processo in un unico rank (per esempio lo 0)
- 6 **consiglio**: definire degli indici che identifichino quale parte va inviata a quale processo.

(ricorda: il prodotto tra matrici **NON** è commutativo: $AB \neq BA$, quindi attenzione quale matrice si manda a tutti i processi)

Moltiplicazione tra matrici 2

Supponiamo di avere inizializzato correttamente il sistema.

Supponiamo che il processo 0 abbia creato A e B.

Esercizio: noti N , p , $rank$ scrivere la routine che manda a tutti i processi tutta la matrice $B[N][N]$ (contenente interi) tramite `MPI_Bcast`.

- **Ricorda:** `MPI_Bcast(*indirizzoDati, NumDati, TipoDati, root, communicator);`
- **Domanda:** chi deve chiamare questa funzione?

Costruisco A e B e invio i pezzi di matrice: esercizio

```

int main(int argc, char *argv[])
{
    int rank, P, from, to, i, j, k, numero_processi, tag=123;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // mio rank
    MPI_Comm_size(MPI_COMM_WORLD, &numero_processi); // numero di processi

    if ( N %numero_processi != 0) { // esci se N/numero_processi da' resto diverso da zero
        if (rank==0) printf("La dimensione della matrice (%d) non e' divisibile per il numero di
            processi usati (%d)\n", N, numero_processi);
        MPI_Finalize();
        exit(-1); // ricorda: serve header stdlib.h altrimenti da warning
    }

    if (rank==0) {
        printf("Calcolo il prodotto di matrici C=AB, di dimensione %dx%d\n", N,N);
        costruisci_matrice(A); // crea tutta la matrice A nel rank 0
        costruisci_identita2(B); // nella matrice B metti l'identia'
    }

    // fare un MPI_Bcast (puntatore, quanti_dati, tipo, root, communicator) della matrice B
    // a tutti i processi!

```

Costruisco A e B e invio i pezzi di matrice bcast

```

int main(int argc, char *argv[])
{
    int rank, P, from, to, i, j, k, numero_processi, tag=123;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // mio rank
    MPI_Comm_size(MPI_COMM_WORLD, &numero_processi); // numero di processi

    if ( N %numero_processi != 0) { // esci se N/numero_processi da' resto diverso da zero
        if (rank==0) printf("La dimensione della matrice (%d) non e' divisibile per il numero di
            processi usati (%d)\n", N, numero_processi);
        MPI_Finalize();
        exit(-1); // ricorda: serve header stdlib.h altrimenti da warning
    }

    if (rank==0) {
        printf("Calcolo il prodotto di matrici C=AB, di dimensione %dx%d\n", N,N);
        costruisci_matrice(A); // crea tutta la matrice A nel rank 0
        costruisci_identita2(B); // nella matrice B metti l'identia'
    }

    MPI_Bcast (B, N * N , MPI_INT, 0 , MPI_COMM_WORLD); // invia B a tutti

```

Moltiplicazione tra matrici 3

Esercizio: noti N , p , $rank$ scrivere la routine che manda dal processo 0 a tutti gli altri processi, la parte della matrice $A[N][N]$ che serve loro per calcolare il prodotto.

- **Consiglio:** usa `MPI_Scatter`
- **Ricorda:** `MPI_Scatter(*indirizzoDatiInv, NumDatiInv, TipoDati, *indirizzoDatiRicez, NumDatiRicez, root, communicator);`
- **Domanda:** chi deve chiamare questa funzione?
- **Domanda:** Come utilizziamo N , p , $rank$ perche' la chiamata alla funzione abbia effetti diversi per ogni processo?
- **Consiglio:** Puo' avere senso che tutta matrice A sia presente in ogni processo (perche'?), ma solo alcune sue parti saranno diverse da 0 nel singolo processo (quelli necessari per la sua parte del prodotto matrice matrice). Serve un **indice** (chiamiamolo `from`) che identifichi da dove la matrice A e' diversa da 0 in un determinato processo (questo numero sara' quindi una funzione del `rank` del processo)
- **Consiglio:** in un array 2D, ($A[i][j]$) in C e' possibile accedere al puntatore della riga n -esima tramite: $A[n - 1]$ (quindi inserendo un solo indice!) ← **DECAY**

Costruisco A e B e invio i pezzi di matrice

```

int main(int argc, char *argv[])
{
    int rank, P, from, to, i, j, k, numero_processi, tag=123;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // mio rank
    MPI_Comm_size(MPI_COMM_WORLD, &numero_processi); // numero di processi

    if ( N %numero_processi != 0) { // esci se N/numero_processi da' resto diverso da zero
        if (rank==0) printf("La dimensione della matrice (%d) non e' divisibile per il numero di
            processi usati (%d)\n", N, numero_processi);
        MPI_Finalize();
        exit(-1); // ricorda: serve header stdlib.h altrimenti da warning
    }

    if (rank==0) {
        printf("Calcolo il prodotto di matrici C=AB, di dimensione %dx%d\n", N,N);
        costruisci_matrice(A); // crea tutta la matrice A nel rank 0
        costruisci_identita2(B); // nella matrice B metti l'identia'
    }

    MPI_Bcast (B, N * N , MPI_INT, 0 , MPI_COMM_WORLD); // invia B a tutti

    // from(mio_rank) = indice di riga di A da cui mandare elementi al processo con mio_rank
    // to(mio_rank) = indice di riga di A fino a cui mandare edm al processo con mio_rank

    //Chiama la funzione MPI_Scatter ( puntatoreInvio, quanti, tipo, puntatorericezione, quanti,
        tipo, root, communicator) per passare pezzi di A a tutti i processi

```

Costruisco A e B e invio i pezzi di matrice soluzione

```

int main(int argc, char *argv[])
{
    int rank, P, from, to, i, j, k, numero_processi, tag=123;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // mio rank
    MPI_Comm_size(MPI_COMM_WORLD, &numero_processi); // numero di processi

    if ( N %numero_processi != 0) { // esci se N/numero_processi da' resto diverso da zero
        if (rank==0) printf("La dimensione della matrice (%d) non e' divisibile per il numero di
            processi usati (%d)\n", N, numero_processi);
        MPI_Finalize();
        exit(-1); // ricorda: serve header stdlib.h altrimenti da warning
    }

    if (rank==0) {
        printf("Calcolo il prodotto di matrici C=AB, di dimensione %dx%d\n", N,N);
        costruisci_matrice(A); // crea tutta la matrice A nel rank 0
        costruisci_identita2(B); // nella matrice B metti l'identia'
    }

    MPI_Bcast (B, N * N , MPI_INT, 0 , MPI_COMM_WORLD); // invia B a tutti
    from = rank * N /numero_processi; //
    to = (rank+1) * N /numero_processi; //
    MPI_Scatter (A, N * N /numero_processi, MPI_INT, &A[from], N * N /numero_processi, MPI_INT,
        0, MPI_COMM_WORLD)

```

Moltiplicazione tra matrici 4

Esercizio: scrivere la routine che calcola il prodotto scalare delle righe di A contenute nel processo e tutte le colonne di B :

- eseguiamo il prodotto scalare (ogni processo fa dei prodotti scalari differenti)
- Il prodotto scalare in un dato processo va dalla riga `from` alla riga finale che verrà indicata con un nuovo indice chiamato `to`
- **Ricorda:** abbiamo definito la prima riga presente in un dato processo come:
`from = rank * N / size;`
- **Soluzione:** la riga finale nel processo con `rank` sarà data da:
`to = (rank+1) * N / size;`
- a questo punto devo costruire un ciclo che gira su tutti i processi usando `from` e `to`:
- ```
for (i=from; i<to; i++) // cicla sulle righe di A contenute in rank
 for (j=0; j< N ; j++) { // cicla sulle colonne di B
 C[i][j]=0; // azzerla la C
 for (k=0; k< N ; k++)
 C[i][j] += A[i][k]*B[k][j]; // prodotto scalare
 }
```
- **Osservazione:** Tutti i processi hanno una matrice  $C$  in memoria. La maggior parte di questa matrice  $C$  è però piena di zeri, e solo una parte contiene gli elementi della matrice prodotto. Solo questa parte andrà poi inviata al processo `root` quando vorremo ricostruire tutta la  $C$ .

## Moltiplicazione tra matrici 5

- **Esercizio:** dobbiamo raccogliere tutte le parti della matrice prodotto  $C$  (che sono state calcolate nei vari processi) e metterle nel processo 0.
- **Ricorda:** `MPI_Gather(*indirizzoDatiInv, NumDatiInv, TipoDati, *indirizzoDatiRicez, NumDatiRicez, root, communicator);`
- **Attento:** i pezzi della matrice  $C$  contenuta in ogni processo sono già al punto giusto di dove andranno messi nella matrice  $C$  finale (il resto della matrice  $C$  contenuto in ogni processo è uguale a zero e non voglio rispedirlo con il `MPI_Gather`)
- **Consiglio:** il numero di "messaggi elementari da inviare tramite il gather deve essere esattamente lo stesso dei messaggi che erano stati sparsi tramite il `MPI_Scatter` qualche slide fa...
- **Soluzione:** `MPI_Gather (C[from], N*N/P, MPI_INT, C, N*N/P, MPI_INT, 0, MPI_COMM_WORLD);`

# Moltiplicazione tra matrici 6

- Riassumendo il cuore del codice:

---

```

MPI_Bcast (B, N * N , MPI_INT, 0 , MPI_COMM_WORLD);
MPI_Scatter (A, N * N /P, MPI_INT, A[from],
N * N /P, MPI_INT, 0, MPI_COMM_WORLD);
for (i=from; i<to; i++)
 for (j=0; j< N ; j++) {
 C[i][j]=0;
 for (k=0; k< N ; k++) C[i][j] += A[i][k]*B[k][j];
 }
MPI_Gather (C[from], N * N /P, MPI_INT, C,
N * N /P, MPI_INT, 0, MPI_COMM_WORLD);

```

---

- **Problema:** Qualora volessi generalizzare il calcolo di matrici al caso in cui il rapporto  $p$  (numero di processi) non sia un divisore di  $N$  numero di righe, quali funzioni dovrei modificare?
- **Soluzione:** i candidati sono chiaramente `MPI_Scatterv` e `MPI_Gatherv`!

## Moltiplicazione tra matrici 7

Cerchiamo di vedere i pregi e i difetti di questa implementazione:

- **Pregi:** implementazione semplice (!?)
- **Difetti:**
  - Ogni processo allocca tutta  $A$  nella propria memoria ma solo  $N/p$  parti di  $A$  sono utili
  - Ogni processo contiene tutta  $B$
  - Ogni processo allocca tutta  $C$  ma ci sono solo  $N/p$  parti di  $C$  utili
  - Se il numero di **processi**  $p$  e' maggiore di quello delle righe  $N$ , i processi in piu' non vengono utilizzati!
- **Soluzione:** Algoritmo di **Cannon** per la moltiplicazione tra matrici (per questo pero', sara' utile avere qualche nozione sulla suddivisione dei **communicator**).
- **Domanda:** allocare tutta la matrice  $B$  e tutta la matrice  $C$  in ognuno dei processi e' uno spreco notevole, per quale motivo e' stato fatto?
- **Risposta:** Facendo in questo modo si hanno i pezzi di matrice in una posizione "conveniente", per esempio quando si fa il `MPI_Gather`, non si fa fatica ad indicare da dove prendere i pezzi di  $C$  contenuti in ogni processo e dove devono finire nel root: e' gia' tutto ordinato!

# Cannon Matrix Multiplication

Un algoritmo intelligente che cerca di *riempire* il meno possibile la memoria dei singoli processi e che sfrutta la comunicazione per far circolare l'informazione tra un processo e l'altro.

- Il **vantaggio** di Cannon e' **solo** una minore occupazione della memoria per il singolo processo, al costo di un maggiore overhead di trasmissione dati.
- L'**idea** di questo algoritmo e' di sfruttare la **commutativita'** e **associativita'** della somma.

L'algoritmo in breve:

- Consideriamo matrici quadrate  $A, B, C: n \times n$ .
- Per gli elementi di matrice di  $A$  ( $B$  e  $C$ ) useremo le **minuscole**  $a_{ij}$  ( $b_{ij}, c_{ij}$ ).
- $C = AB$  dove  $c_{ij} = \sum_k a_{ik} b_{kj}$
- supponiamo che ci siano  $p$  processi
- supponiamo di suddividere la matrice  $A$  in  $p$  sottomatrici quadrate
- per semplicita' supponiamo che  $p$  sia un intero quadrato (p.es.  $p = 9 = 3 \times 3$ )

In questo modo possiamo creare una **griglia logica** di sottomatrici. Questa griglia contiene  $\sqrt{p} \times \sqrt{p}$  matrici, con queste caratteristiche:

- $A_{ij}$  e' una sottomatrice  $A$  dove  $i, j = 1, 2, \dots, \sqrt{p}$  ed ha dimensione  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$  (in tutto le  $p$  sottomatrici ricostruiscono tutta  $A$ ).
- $B_{ij}$  e' l'analogo ottenuto dalla matrice  $B$ , dove  $i, j = 1, 2, \dots, \sqrt{p}$  con dimensione  $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

# Cannon: le matrici

Per esempio:

- sia  $A$  una matrice  $9 \times 9$
- supponiamo che il numero di processi sia  $p = 9$  ( $3 \times 3$ ).
- suddividiamo  $A$  in 9 sottomatrici chiamate  $A_{ij}$  (vorremmo mandare ad ogni processo solo una piccola parte di tutta la matrice).

$$A = \begin{pmatrix}
 \begin{array}{ccc|ccc|ccc}
 a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} \\
 a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} \\
 a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} \\
 \hline
 a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} \\
 a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} \\
 a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} \\
 \hline
 a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} \\
 a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} \\
 a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8}
 \end{array}
 & = &
 \begin{pmatrix}
 A_{00} & A_{01} & A_{02} \\
 A_{10} & A_{11} & A_{12} \\
 A_{20} & A_{21} & A_{22}
 \end{pmatrix}
 \end{pmatrix}$$

In questo caso le sottomatrici  $A_{ij}$  hanno dimensione  $3 \times 3$  ( $\sqrt{p} = 3$ ,  $n = 9$ ,  $n/\sqrt{p} = 3$ ).

## Matrici per ogni processo

- Di solito i processi vengono pensati come uno dietro l'altro (per via del `rank` che va da 0 a  $p - 1$ ).
- sarebbe **piu' comodo** pensare ai processi in termini di coordinate  $i, j$  all'interno della **griglia logica** che abbiamo appena formato per suddividere la matrice globale  $A$ :

|                          |                                           |                                |
|--------------------------|-------------------------------------------|--------------------------------|
| <b>Nome</b> del processo | che cosa deve <b>contenere</b> all'inizio | cosa vogliamo <b>calcolare</b> |
| $P_{ij}$                 | $A_{ij} \ B_{ij}$                         | $C_{ij}$                       |

Quindi (al passo 0 dell'algoritmo Cannon) ad ogni processo  $P_{ij}(i, j = 0, 2)$  mandiamo un pezzo di  $A$  e un pezzo di  $B$ , nel modo seguente:

$$\begin{pmatrix} P_{00} & P_{01} & P_{02} \\ P_{10} & P_{11} & P_{12} \\ P_{20} & P_{21} & P_{22} \end{pmatrix} \Leftrightarrow \begin{pmatrix} A_{00}, B_{00} & A_{01}, B_{01} & A_{02}, B_{02} \\ A_{10}, B_{10} & A_{11}, B_{11} & A_{12}, B_{12} \\ A_{20}, B_{20} & A_{21}, B_{21} & A_{22}, B_{22} \end{pmatrix} \Rightarrow ? \begin{pmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$$

si noti che in questo modo, ogni processo ha bisogno di occupare solo  $1/9$  ( $=1/p$  dove  $p$ =numero di processi) della memoria necessaria per contenere  $A$  e  $B$ .

**Problema:** Per calcolare  $C_{ij}$  non e' sufficiente avere le sottomatrici  $A_{ij}$  e  $B_{ij}$  !!!

- Sono necessarie tutte le  $A_{ik} B_{kj}$  con  $k = 0, \dots, \sqrt{p} - 1$

Con una implementazione "semplice" sarebbe necessario avere  $2\sqrt{p}$  sottomatrici per calcolare  $C_{ij}$ , pero' l'occupazione di memoria dei singoli processi  $2\sqrt{p}$  volte **maggiore**... Per superare questo problema possiamo notare la seguente formula:

**contention-free formula** (formula senza contenziosi):

$$C_{ij} = \sum_{k=0}^{\sqrt{p}} A_{i;(i+j+k) \bmod(\sqrt{p})} B_{(i+j+k) \bmod(\sqrt{p});j} \quad (6)$$

Nonostante sembri una formula complicata, ci dice semplicemente che possiamo cominciare la sommatoria dove vogliamo e attaccare i pezzi man mano (tanto la somma e' **commutativa e associativa**).

## Cannon 2

Per esempio prendiamo il processo  $P_{12}$  nel quale (all'inizio) abbiamo inserito  $A_{12}$  e  $B_{12}$ . Il nostro obiettivo è però quello di calcolare tutta la sottomatrice  $C_{12}$  (all'interno del processo  $P_{12}$ ). La formula per ottenerlo è:

$$C_{ij} = \sum_{k=0}^{\sqrt{p}} A_{ik} \cdot B_{kj} \quad (7)$$

(ricordiamo che è la somma di prodotti di sottomatrici). Nel nostro caso diventa:

$$\begin{aligned}
 C_{ij} &= A_{i0} \cdot B_{0j} && + A_{i1} \cdot B_{1j} && + A_{i2} \cdot B_{2j} \\
 &&& \text{mentre all'inizio è:} && \\
 C_{ij} &= A_{ij} \cdot B_{ij}
 \end{aligned} \quad (8)$$

Per esempio per il processo  $P_{12}$ , vorremmo calcolare la corrispondente parte di matrice prodotto:

$$C_{12} = A_{10} \cdot B_{02} + A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \quad (9)$$

all'inizio le sottomatrici presenti nella memoria di  $P_{12}$  sono solo 2:

$$A_{12} \quad B_{12} \quad (10)$$

Sfortunatamente questa coppia di matrici non serve per calcolare  $C_{12}$  (le coppie di sottomatrici  $A$  e  $B$  devono essere tali che l'indice di colonna di  $A$  sia identico all'indice di riga di  $B$ ). **IDEA** con la [comunicazione](#) proviamo a spostare le sottomatrici da un processo all'altro (tipo cubo di **Rubik**) per ottenere le sottomatrici volute.

## Ancora Cannon? Allineamento iniziale

Se facciamo uno **shift** delle sottomatrici all'interno della matrice logica, possiamo cambiare uno degli indici, per esempio per cambiare il secondo indice si fa uno shift **laterale**:

$$A_{ij} \xleftarrow{m \text{ passi}} A_{i(j+m)\sqrt{p}} \quad (11)$$

oppure, se vogliamo cambiare il valore del primo indice, facciamo uno shift **verticale**

$$B_{ij} \xrightarrow{l \text{ passi}} B_{(i+l)\sqrt{p},j} \quad (12)$$

(ricorda che la matrice logica ha dimensioni  $\sqrt{p} \times \sqrt{p}$  e lo shift delle sottomatrici e' **ciclico**: uscendo da un lato si rientra dal lato opposto)

A questo punto dobbiamo **solo(?)** trovare dei valori per  $l$  e  $m$  tali che gli indici colonna di tutte le sottomatrici A e gli indici di riga delle sottomatrici di B siano le stesse:

$$(j + m)\sqrt{p} = (i + l)\sqrt{p} \quad (13)$$

Come devono essere  $m$  e  $l$ ?

Questa equazione si risolve per esempio  $m = i$  e  $l = j$ :

$$(j + i)\sqrt{p} = (i + j)\sqrt{p} \quad (14)$$

(il modulo  $\sqrt{p}$  serve solo ad assicurare che non si prendano degli indici che al di fuori dei valori sensati)

Questo ci riporta alla (6) Ovvero se le sottomatrici hanno degli indici che seguono questa formula allora mi servono per fare il prodotto scalare ed ottenere  $C_{ij}$

## Ancora Cannon 2?

Il primo passo di Cannon associa al processo  $P_{ij}$  le matrici in una **griglia logica**:

$$A_{ij} \quad B_{ij} \quad (15)$$

Pero' a noi servono coppie di matrici nella forma (indici **interni** uguali):

$$A_{ix} \quad B_{xj} \quad (16)$$

Cambiare il valore della **seconda** coordinata delle sottomatrici  $A$  e' equivalente a *shiftarle* lungo la direzione **sinistra/destra** nella griglia logica:

$$A_{ij} \xleftarrow{m \text{ passi}} A_{i(j+m)\sqrt{p}} \quad (17)$$

Cambiare il valore della **prima** coordinata delle sottomatrici  $B$  equivale a *shiftarle* in **alto/basso** nella griglia logica:

$$B_{ij} \xrightarrow{l \text{ passi}} B_{(i+l)\sqrt{p}j} \quad (18)$$

A questo punto dobbiamo trovare un algoritmo per shiftare sia le  $A$  che le  $B$  in modo che i valori degli indici **interni** coincidano:

Vogliamo ora sapere il numero di passi  $m$  da spostare a sinistra le sottomatrici  $A$  e il numero di passi  $l$  da spostare verso l'alto le sottomatrici  $B$  perche' gli indici **interni** coincidano:

$$m = j + k \Rightarrow A_{i:(j+i+k)\sqrt{p}} \quad (19)$$

$$l = i + k \Rightarrow B_{(i+j+k)\sqrt{p}j} \quad (20)$$

# Cannon visualizzato

Guardare il video <https://www.youtube.com/watch?v=sB-Dh4DsOy0>

Per esempio, scegliamo  $k = 0$ , la sottomatrice  $A_{12}$  va dunque rimpiazzata con  $A_{1(2+1)_2} = A_{10}$ . Ma questo vale per **tutte le altre sotto matrici** della stessa riga, si sta semplicemente eseguendo uno **shift a sinistra** (ciclico, ovvero uscendo da destra si ritorna a sinistra).

La formula 6 (contention-free formula) equivale a prendere e fare uno shift a sinistra di tanti passi quanti sono i numeri di riga delle sottomatrici  $A$ .

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \leftarrow \\ \leftarrow\leftarrow \end{pmatrix} \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{11} & A_{12} & A_{10} \\ A_{22} & A_{20} & A_{21} \end{pmatrix}$$

Similmente e' uno shift verso l'alto di tanti passi quanti sono i numeri di colonna (0,1 e 2) per le sottomatrici  $B$ .

$$\begin{pmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} \uparrow & \uparrow\uparrow \end{pmatrix} \begin{pmatrix} B_{00} & B_{11} & B_{22} \\ B_{10} & B_{21} & B_{02} \\ B_{20} & B_{01} & B_{12} \end{pmatrix}$$

Vediamo ora in ogni processo quali sottomatrici sono presenti, per esempio nel processo  $P_{12}$  si hanno:  $A_{10}$  e  $B_{02}$ :

**WOW** queste mi servivano proprio per calcolare la sottomatrice  $C_{12}$ !

Al passo successivo, devo solo ricordare che mi servono ancora delle sottomatrici della forma  $A_{ik}B_{kj}$ , devo solo cambiare il valore di  $k \rightarrow k + 1$ , come si fa? **Risposta:** basta fare uno shift a sinistra per tutte le sottomatrici  $A$  e uno in alto per tutte le sottomatrici  $B$ !

**Dopo quanti passi finisco?**

**Risposta:** Semplice basta che io abbia fatto  $\sqrt{p}$  moltiplicazioni tra sottomatrici e sono sicuro di averle fatte tutte!

# Quale logica?

- 1 Creo  $A$  e  $B$  all'interno del rank 0
- 2 Mando  $A_{ij}$  e  $B_{ij}$  al processo  $P_{ij}$
- 3 faccio il primo insieme di shift a sinistra di sottomatrici  $A_{ij}$  che dipende dall'indice di riga  $i$
- 4 faccio il primo insieme di shift verso l'alto di sottomatrici  $B_{ij}$  che dipende dall'indice di colonna  $j$
- 5 faccio i prodotti scalari delle sottomatrici contenute nel processo e sommo alla  $C_{ij}$  già presente nel processo  $P_{ij}$
- 6 se ho fatto  $\sqrt{p}$  prodotti tra sottomatrici, finisco
- 7 faccio uno shift a sinistra delle sottomatrici  $A_{nm}$  di un posto
- 8 faccio uno shift in su delle sottomatrici  $B_{mn}$  di un posto
- 9 ritorno al punto 5

## Suddividere il Communicator

A questo punto e' necessaria una digressione su come riuscire a gestire una griglia logica di processori. Questo richiede di manipolare i **communicator**.

`MPI_COMM_WORLD` e' il comunicatore generale che contiene **tutti** i processi, pero' e' possibile suddividerlo, per esempio, se si vogliono fare delle azioni collettive solo su un insieme ristretto di processi.

- quando si suddivide un communicator, il contenitore maggiore continua ad esistere (si creano dei sotto communicator).
- `MPI_Comm_split` crea i nuovi sub communicators
- `MPI_Comm_free (&nome_nuovo_communicator)`; libera la memoria del nuovo communicator. E' **sempre** bene liberare i communicator quando non servono piu'.
- `MPI_Comm_dup (MPI_Comm old_comm, MPI_Comm *newcomm)` crea una copia di un communicator. **Puo' essere utile copiare `MPI_COMM_WORLD` e lavorare con la copia.**  
 ([https://www.quora.com/MPI\\_Comm\\_dup-why-would-we-ever-need-to-duplicate-the-communicator](https://www.quora.com/MPI_Comm_dup-why-would-we-ever-need-to-duplicate-the-communicator))

**ATTENZIONE:** in MPI tutte le risorse sono in numero limitato, e' per questo che si rimuovono i communicator che non servono, altrimenti si puo' arrivare a saturare il sistema e ad avere un errore.

# MPI\_Comm\_split

una funzione per creare nuovi communicator e' la seguente:

---

```

MPI_Comm_split (
 MPI_Comm vecchio_communicator, // communicator di partenza da suddividere
 int colore, // proc. con stesso colore vanno in stesso communicator
 int key, // determina il rank nel nuovo communicator
 MPI_Comm* nuovo_communicator) // nome del nuovo sotto-communicator creato

```

---

- questa funzione puo' creare fino a `size` (numero processi del communicator di partenza) **nuovi** communicator (ma in questo e' un caso estremo perche' in ogni communicator ci sarebbe solo un processo)
- **Attenzione** e' una **collective** call; tutti i processi all'interno di un **communicator** devono chiamarla (visto che possono esistere vari communicator, il communicator (di partenza) non deve essere necessariamente `MPI_COMM_WORLD`)!
- Processi a cui viene assegnato lo stesso **colore** (in realta' e' un numero intero...) finiscono nello stesso communicator (quindi in pratica il **colore** sara' una funzione del `rank` (scelta dal programmatore), per esempio `colore=rank/2`).
- Si puo' lasciare il **colore** indeterminato passando `MPI_UNDEFINED`, in questo modo il processo con questo parametro resta solo nel communicator di partenza.
- La **key** e' un intero, associato al processo.

## Attenzione:

Quando faccio uno split mi ritrovo con vari sotto-communicator... ma nella funzione inserisco un solo nome (nell'esempio `nuovo_communicator`), come distinguo tra loro i **nuovi** communicator? come faccio a lavorare con il communicator che ha un determinato colore? **Risposta:** Ricordiamoci che ogni processo (dato il proprio rank originale) puo' calcolare il proprio **colore**. Quindi processi con **colore** differente non possono comunicare.

## MPI\_Comm\_split: esempio

- Usare `MPI_Comm_split ( *comm, colore, key, *nuovo_communicator)` e suddividere il communicator in 3 parti
- assegnare un "nuovo rank =0" al rank 2
- fare scrivere ad ogni processo il proprio **rank originale**, il proprio **nuovo rank**, e il **colore**
- inviare un messaggio nel communicator con **colore 0**, dal rank 2 al rank 1 (i rank si riferiscono al nuovo comunicatore)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
 MPI_Init(NULL, NULL);
 int dato=0;
 int vecchio_rank, size;
 int nuovo_rank, new_size;
 MPI_Comm nuovoComm; // nome di un nuovo communicator
 MPI_Comm_rank (MPI_COMM_WORLD, &vecchio_rank);
 MPI_Comm_size (MPI_COMM_WORLD, &size);

 printf("Vecchio rank: %d, nuovo rank: %d, colore: %d, dato=%d \n",
 vecchio_rank, nuovo_rank, colore, dato);
 MPI_Comm_free (&nuovoComm);
 MPI_Finalize();
}
```

## MPI\_Comm\_split: esempio

```

#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
 MPI_Init(NULL, NULL);
 int dato=0;
 int vecchio_rank, size;
 int nuovo_rank, nuova_size;
 MPI_Comm nuovoComm; // nome di un nuovo communicator
 MPI_Comm_rank(MPI_COMM_WORLD, &vecchio_rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 int colore = vecchio_rank / 3; // determina il colore di ogni processo per poi suddividere
 MPI_Comm_split(MPI_COMM_WORLD, colore, vecchio_rank, &nuovoComm); // dividi communicator
 if(vecchio_rank==2) MPI_Comm_split(MPI_COMM_WORLD, colore, -1, &nuovoComm); // metto key bassa
 MPI_Comm_rank(nuovoComm, &nuovo_rank); // nuovo rank
 MPI_Comm_size(nuovoComm, &nuova_size); // nuova size
 if (nuovo_rank == 2 && colore == 0) {
 dato=34;
 MPI_Send(&dato, 1, MPI_INT, 1, 33, nuovoComm); }
 if (nuovo_rank == 1 && colore == 0) {
 MPI_Recv(&dato, 1, MPI_INT, 2, 33, nuovoComm, MPI_STATUS_IGNORE);
 dato=dato+1; }
 printf("Vecchio rank: %d, nuovo rank: %d, nuova size: %d colore: %d, dato=%d \n",
 vecchio_rank, nuovo_rank, nuova_size, colore, dato);
 MPI_Comm_free(&nuovoComm);
 MPI_Finalize();
}

```

Attenzione cosi' com'e' va in deadlock: perche'?

## Coordinate Cartesiane:

Come descritto precedentemente ha senso cercare di superare una visione in cui tutti i processi sono uno dietro l'altro ed associare loro una **griglia virtuale** (in alcuni casi questa può essere associata alla reale distanza fisica dei processi ma è estremamente difficile ottenere dei risultati in questo senso)

Vogliamo quindi che si abbiano delle coordinate virtuali associate al `rank`:

$$\text{rank} \longrightarrow (i_x, i_y) \quad (21)$$

- Con questa nuova **topologia** i processi all'interno della griglia possono avere 4 vicini virtuali (e non più solo 2, ovvero quello di sinistra e quello di destra).
- È utile che si possa dichiarare se le direzioni della griglia virtuale devono essere **cicliche**, ovvero se "uscendo" a destra si rientra a sinistra (e viceversa).

MPI fornisce delle funzioni che gestiscono questo tipo di riordino automaticamente, consentono di controllare la griglia (andando anche a strutture **3D**).

## Coordinate Cartesiane 2:

### MPI\_Cart\_create()

- Cosa fa questa funzione?
- La **topologia** aiuta notevolmente nel caso in cui si debbano fare delle operazioni di **shift**
- ATTENZIONE: e' una **Collective** communication
- PERICOLO: se il numero di processi della griglia e' **inferiore** a quelli lanciati, allora ad alcuni processi verra' assegnato un MPI\_COMM\_NULL
- **Attenzione**: se il numero di processi della griglia e' **superiore** al numero di processi lanciati, la griglia non puo' essere completata e si ha sicuramente un errore!

---

```

MPI_Cart_create(
 MPI_Comm comm_old, //communicator di partenza
 int ndims, // dimensioni di suddivisione, se voglio 2D metto 2, 3D metto 3...
 int *dims, // array (p.es. se ndims=2) posso avere array[3][4] ho una griglia 3x4,
 int *periods, // array logico di dimensione ndims, indica periodicit  per ogni dim
 int reorder, // se falso allora nuovi rank = vecchi, altrimenti riordina
 MPI_Comm *comm_cart); // nome del nuovo comunicatore

```

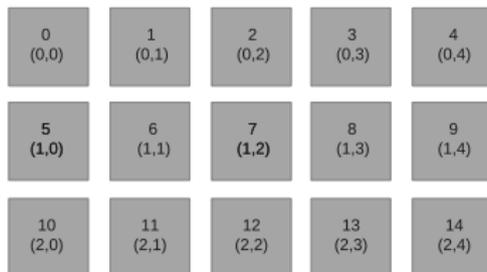
---

**Osservazione:** a meno di avere particolari ragioni, viene consigliato di lasciare fare il reordering dei rank, in questo modo risulta piu' facile gestire i processi. Questo perche' i nuovi rank vengono assegnati con un algoritmo che in qualche modo segue la topologia, e aiutano nella costruzione del programma, essendo piu' intuitivi.

## Visualizziamo le coordinate

Consideriamo un caso con 15 processi, in cui la griglia virtuale ha:

- `ndims = 2`
- `dims[0] = 3` e `dim[1] = 5`:



( si noti l'effetto ottico...)

## Muoversi nelle coordinate

Esiste una funzione che, dato il nostro rank, ci dice quali sono le coordinate all'interno della griglia logica:

---

```
int MPI_Cart_coords(MPI_Comm comm, // input
 int rank, // input rank del processo nel comunicatore comm
 int maxdims, // input lunghezza vettore coord
 int coords[]) // output array dimensione ndims
```

---

Per l'operazione inversa, ovvero date le coordinate trovare il rank si usa invece:

---

```
int MPI_Cart_rank(MPI_Comm comm,
 int coords[], // input coordinate
 int *rank) // output rank
```

---

## Esempio MPI\_Cart\_create

Cerchiamo di usare `MPI_Cart_create`: costruiamo un nuovo communicator con coordinate cartesiane. La griglia sia composta da 5 righe e 3 colonne, sia periodica rispetto alle righe ma NON periodica rispetto alle colonne. Facciamo scrivere il rank originale, le coordinate e il nuovo rank.

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[]) {
 int rank;
 MPI_Comm griglia;
 int dim[2],period[2],riordina;
 int coord[2],nuovo_rank;
 # define TRUE 0; // occhio che in C non ci sono le Boolean
 # define FALSE 1; // vanno definite apposta

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 // MPI_Cart_create(comm_partenza,numero_dimensioni, arraydim,arrPeriodicita,riordino,&
 // nuovo_communicator);
 // MPI_Cart_coords(nome_communicator,mio_rank,numero_dimensioni,coordinate);

 MPI_Finalize();
}
```

## Soluzione di MPI\_Cart\_create

Cerchiamo di usare `MPI_Cart_create`: costruiamo un nuovo comunicator con coordinate cartesiane. La griglia sia composta da 5 righe e 3 colonne, sia periodica rispetto alle righe ma NON periodica rispetto alle colonne. Facciamo scrivere il rank originale, le coordinate e il nuovo rank.

---

```

#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[]) {
 int rank;
 MPI_Comm griglia;
 int dim[2],period[2],riordina;
 int coord[2],nuovo_rank;
 # define TRUE 0; // occhio che in C non ci sono le Boolean
 # define FALSE 1; // vanno definite apposta

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 dim[0]=5; dim[1]=3; // numero di righe (5) e numero di colonne (3) (tot=15)
 period[0]=TRUE; period[1]=FALSE; // periodo x vero, y falso
 riordina=TRUE; // associa nuovi rank
 MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,riordina,&griglia); // crea coord cartesiane
 MPI_Comm_rank(griglia, &nuovo_rank); // controllo il nuovo rank
 MPI_Cart_coords(griglia,nuovo_rank,2,coord); // cerco le coordinate associate al nuovo rank
 MPI_Cart_rank(griglia,coord,&nuovo_rank);
 printf("Il processo %2d, va alla posizione (%d,%d) con nuovo rank %2d\n"
 , rank, coord[0],coord[1],nuovo_rank);

 MPI_Finalize();
}

```

---

## Muoversi nelle coordinate 2

In una griglia logica spesso si fanno delle operazioni di "shift". Ovvero i dati passano in modo ciclico tra i processi. Supponiamo, per esempio, di voler spostare un dato di 2 passi alla nostra sinistra (ricordate **Cannon**):

TUTTI i processi vogliono fare questa operazione (ergo, per esempio, io mando i miei dati a 2 passi a sinistra, ma anche il mio vicino di due passi a destra lo fa...). Vediamo ora una funzione che aiuta a fare gli shift: `MPI_Cart_shift()`

- **Attenzione:** questa funzione **NON** e' una collective.
- **Attenzione:** `MPI_Cart_shift` non muove **NULLA!** serve solo per conoscere il rank di partenza e di arrivo dei dati.
- voglio che i dati posseduti dal processo chiamante vadano in un altro processo alla sua **sinistra** (o su, giu', destra...)
- allo stesso tempo mi aspetto che al processo chiamante arrivino dei dati dalla sua **destra** (o giu', su, sinistra...).
- Per inviare (e ricevere) dei dati mi serve il **rank** del sender o receiver (**non** si usano le coordinate!) ho quindi bisogno di una funzione che mi aiuti a conoscere, per esempio il rank del processo due passi alla mia sinistra!
- Intuitivamente questa funzione e' pensata per aiutare un `MPI_Sendrecv`, quindi ogni processo invia dei dati e li riceve, con questa funzione sappiamo da chi e a chi.

---

```
MPI_Cart_shift(MPI_Comm comm,
 int direction, // 1 direzione dx/sx; 0 direzione su giu'
 int disp, // di quanti passi mi muovo?
 int *rank_source, // parametro in USCITA, indica DA dove ricevo
 int *rank_dest) // parametro in USCITA, indica A dove invio
```

---

Nota, che non si inserisce il rank del chiamante... perche'? (e' semplice)

## Muoversi nelle coordinate 3

Ora che conosciamo i rank di partenza e di arrivo, vorremmo una funzione sola che usi **poca memoria** e permetta di ricevere e di “far girare” informazioni.... (ok conosciamo già `MPI_Sendrecv` facciamo un ulteriore passo in avanti):

---

```

MPI_Sendrecv_replace(void *buf, // area di memoria dove vanno e vengono i dati
 int count, // quanti dati mandati
 MPI_Datatype datatype, // tipo dei dati
 int dest, // rank del destinatario
 int sendtag, // tag di invio
 int source, // rank di dove arrivano i dati
 int recvtag, // tag di ricezione
 MPI_Comm comm, // comunicatore dove ci troviamo
 MPI_Status *status)

```

---

Consecutio temporum:

- 1 Prima i dati vengono inviati
- 2 la stessa area di memoria viene adibita ai dati da ricevere
- 3 quando i dati sono stati inviati, vengono ricevuti (da qualunque tipo di funzione di invio, non solo un `Sendrecv`...)

# Cominciamo a costruire Cannon

Struttura del codice (consigliata):

- 1 **Buona Pratica**: suddividiamo il codice in funzioni
- 2 inizializziamo il codice, con MPI
- 3 facciamo in modo che il rank=0 **costruisca** e **mandi** le sottomatrici  $A_{ij}$  e  $B_{ij}$  ai processi  $P_{ij}$
- 4 e' utile avere una funzione che faccia il *normale* prodotto tra matrici
- 5 serve una funzione che faccia gli spostamenti dell'algoritmo Cannon

# Funzione trasformazione: mandare le sottomatrici ai processi

- che funzione possiamo usare per mandare dal rank 0 le sottomatrici ad ogni processo?
- **consiglio** `MPI_Scatter`
- **Problema** `MPI_Scatter` manda dati **contigui**, mentre le sottomatrici non lo sono (vedi matrice sotto).
- ricorda che, in C, un array bidimensionale e' memorizzato mettendo le righe una dietro l'altra in fila. Quindi **de facto** e' identico ad un array 1D.
- il problema e' che nemmeno in questo caso i dati da mandare sono contigui!
- per visualizzare le matrici e' bene che siano in forma 2D
- scriviamo quindi le matrici che vogliamo visualizzare in 2D e poi costruiamo una funzione che trasformi le matrici 2D in un array 1D in cui le sottomatrici sono formate da dati contigui

Matrice A (in forma utile per l'essere umano):

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \rightarrow \begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix}$$

Riordino la matrice A in modo che sia pronta per essere mandata ad ogni processo sotto forma di righe una dietro l'altra:

`a00 a01 a10 a11 a02 a03 a12 a13 a20 a21 a30 a31 a22 a23 a32 a33`

# trasformare una matrice quadrata in un array 1D

Supponiamo che  $A$  sia una matrice  $N \times N$ , e che sia suddivisibile in matrici quadrate  $A_{ij}$  di dimensione  $N_{sot} \times N_{sot}$ , e supponiamo di voler mettere tutti i dati in un array 1D dove i dati della prima sottomatrice vengono per primi, quelli della seconda vengono per secondi e così' via. Sia  $pn = N/N_{sot}$  il numero di sottomatrici in una riga (o colonna).

Proviamo a pensare ad un algoritmo che riorganizzi una matrice in modo che i dati contigui siano esattamente equivalenti alle sottomatrici da mandare.

- dobbiamo identificare ogni sottomatrice (per esempio con 2 coordinate, della "griglia" di sottomatrici), che possiamo chiamare  $ix$  e  $iy$
- servono due indici per muoverci all'interno di una sottomatrice, chiamate  $i$  e  $j$
- riordiniamo ora gli elementi dalla matrice  $A$  alla matrice  $1A$  (linea a...)
- la nuova matrice  $1A$  verrebbe costruita tramite un indice che dipende dai 4 indici succitati:  $ix, iy, i, j$

Per esempio:

$$ix = 1, iy = 2 \Rightarrow A_{12} \quad i = 0, j = 1 \Rightarrow (A_{12})_{01} = a_{3,7} \quad (22)$$

$$A = \begin{pmatrix}
 a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} \\
 a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} \\
 a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} \\
 a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} \\
 a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} \\
 a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} \\
 a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} \\
 a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} \\
 a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8}
 \end{pmatrix} =$$

$$= \begin{pmatrix}
 A_{00} & A_{01} & A_{02} \\
 A_{10} & A_{11} & A_{12} \\
 A_{20} & A_{21} & A_{22}
 \end{pmatrix} \Rightarrow A_{12} = \begin{pmatrix}
 a_{3,6} & a_{3,7} & a_{3,8} \\
 a_{4,6} & a_{4,7} & a_{4,8} \\
 a_{5,6} & a_{5,7} & a_{5,8}
 \end{pmatrix}$$

## soluzione alla trasformazione

In questa implementazione servono 4 cicli for:

---

```

conta = 0;
for (ix=0; ix< N/pn; ix++) { // i identifica la sottomatrice A_{ix,iy}
 for (iy=0; iy< N/pn; iy++) { // j identifica la sottomatrice A_{ix,iy}
 for (i=0; i<pn; i++) { // data A_{ix,iy} mi muovo in essa A_{ix,iy}_{ij}
 for (j=0; j< pn; j++) { // data A_{ix,iy} mi muovo in essa A_{ix,iy}_{ij}
 lA[conta] = A[i+ix*pn][j+iy*pn] ;
 conta = conta+1 ; } } } }

```

---

## un main di Cannon

- Prima manda le sottomatrici di A e B, in modo banale, a tutti i processi.
- Aspetta che tutti i processi abbiano ricevuto (barriera)
- chiama la funzione che implementa Cannon
- Aspetta che tutti i processi abbiano calcolato
- raccogli tutti i risultati.

---

```
// manda ad ogni processo una sottomatrice a,
MPI_Scatter(A, Nsot*Nsot, MPI_INT, a, Nsot*Nsot, MPI_INT, root, MPI_COMM_WORLD);
MPI_Scatter(B, Nsot*Nsot, MPI_INT, b, Nsot*Nsot, MPI_INT, root, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD); // aspetta che tutti i processi abbiano ricevuto

cannon(a, b, c, Nsot); // chiama Cannon
MPI_Barrier(MPI_COMM_WORLD); // aspetta che esegua ogni processo

// raccogli tutte le sottomatrici c in C nel ROOT (occhio che e' collective)
MPI_Gather(c, Nsot*Nsot, MPI_INT, C, Nsot*Nsot, MPI_INT, root, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

---

# Vediamo Cannon:

## Dichiarazioni iniziali

```

int nproc, rank, rank_griglia;
int dim[2], periodicità[2], coord[2];
int rankDx, rankSx, rankSu, rankGiu;
int sorgente, destinazione;

MPI_Status status; // per comunicazione
MPI_Comm griglia; // nome nuovo communicator

MPI_Comm_size(MPI_COMM_WORLD, &nproc); // la dimensione del communicator serve anche qui
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // trova il mio rank

```

## Parte Prima: costruiamo un nuovo communicator con una topologia di griglia.

```

dim[0] = n; // dimensione della griglia logica n x n
dim[1] = n; // dimensione della griglia logica n x n
periodicità[1] = 1; // definisco cicliche sull'asse x e y
periodicità[0] = 1; // " "

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, periodicità, 1, &griglia); // crea nuovo comm
MPI_Comm_rank (griglia, &rank_griglia); // trova il mio nuovo rank
MPI_Cart_coords(griglia, rank_griglia, 2, coord); // trova le mie coordinate
// =====

```

## continuiamo a vedere Cannon:

Facciamo l'allineamento

---

```
// trova i rank per gli shift a SINISTRA (coord[0]) e sposta le sottomatrici A
MPI_Cart_shift(griglia, 1, -coord[0], & sorgente, & destinazione); //
MPI_Sendrecv_replace(a, n*n, MPI_INT, destinazione, 1, sorgente, 1, griglia, & status);
// =====
// trova i rank per gli shift in ALTO (coord[1]) e sposta le sottomatrici B
MPI_Cart_shift(griglia, 0, -coord[1], & sorgente, & destinazione); //
MPI_Sendrecv_replace(b, n*n, MPI_INT, destinazione, 1, sorgente, 1, griglia, & status);
// =====
MPI_Barrier(griglia); // aspetta che tutti i processi abbiano fatto il loro dovere
```

---

**Moltiplicazioni** a questo punto dobbiamo: cominciare a fare moltiplicazioni tra le sottomatrici *a* e *b* presenti in ogni singolo processo, spostare a sx/alto e rifare le moltiplicazioni:

---

```
// trova le coordinate dei vicini di destra, sinistra, su e giu'
MPI_Cart_shift(griglia, 1, -1, & rankDx, & rankSx); // trova rank dei miei vicini di dx e sx
MPI_Cart_shift(griglia, 0, -1, & rankGiu, & rankSu); // trova rank dei miei vicini su e giu
// =====

for(i =0;i<dim[0];i++) // cicla sul numero di sottomatrici in una riga logica
{
moltiplica(a, b, c, n); // fai prodotto delle sottomatrici all'interno del processo
MPI_Sendrecv_replace(a, n*n, MPI_INT, rankSx, 1, rankDx, 1, griglia, & status); // sposta
(contention free formula)
MPI_Sendrecv_replace(b, n*n, MPI_INT, rankSu, 1, rankGiu, 1, griglia, & status); // sposta
(contention free formula)
}
MPI_Comm_free(&griglia); // libera risorse
```

---

# Costo dell'invio messaggi nel parallel computing

Alcuni termini:

- **Startup Time** ( $t_s$ ) e' il costo per gestire il messaggio ai nodi di invio e ricezione. In questo costo sono inclusi, il fatto di preparare il messaggio (mettendolo nell'*envelope*), il tempo di esecuzione dell'algoritmo di *routing*, il tempo per stabilire un'interfaccia tra il nodo locale e il router.
- **Per-word-transfer-time** ( $t_w$ ) e' il tempo che necessita una *parola* ad essere trasferita da un processo all'altro. Se la banda del sistema consente di spedire  $r$  parole al secondo  $t_w = \frac{1}{r}$ .
- **Per-hop-time** ( $t_h$ ) se un messaggio deve passare lungo un percorso (e quindi attraversare vari nodi) e' il tempo necessario per attraversare un singolo nodo.

## Cannon: prestazioni 2

Ricordiamo che:

- ci sono  $\sqrt{p} \times \sqrt{p}$  sottomatrici  $A_{ij}$  (e anche  $B_{ij}$ )
- ognuna di queste sottomatrici  $A_{ij}$  o  $B_{ij}$  contiene  $\frac{n}{p^{1/2}} \times \frac{n}{p^{1/2}} = \frac{n^2}{p}$  valori

Il costo totale di Cannon e' quindi dato dal costo dei seguenti passi (ognuno fatto in parallelo da tutti i processi):

- l'**allineamento** iniziale:  $\left(t_s + t_w \frac{n^2}{p}\right)$  (viene eseguito solo all'inizio, 2 volte: una per le  $A_{ij}$  e una per le  $B_{ij}$ )
- le **moltiplicazioni** di due sottomatrici (di dimensione  $\frac{n}{p^{1/2}}$ ) richiedono ognuna  $\frac{n^3}{p^{3/2}}$ , sono  $\sqrt{p}$ , per un totale di:  $\frac{n^3}{p}$
- ogni **shift** successivo richiede:  $\left(t_s + t_w \frac{n^2}{p}\right)$  (vengono eseguiti  $\sqrt{p}$  shift per le  $A_{ij}$  e altrettante per le  $B_{ij}$ )

Il tempo parallelo che necessita per moltiplicare 2 matrici  $n \times n$ , tramite l'algoritmo Cannon, con  $p$  processi scala quindi come:

$$T_p = \frac{n^3}{p} + 2\sqrt{p} \left(t_s + t_w \frac{n^2}{p}\right) + 2 \left(t_s + t_w \frac{n^2}{p}\right) \quad (23)$$

# Algoritmo DNS 1

**Attenzione** cambiamo un po' la notazione (per seguire l'immagine della diapositiva successiva).

- **NON** suddividiamo le matrici  $A$ ,  $B$  e  $C$  in sottomatrici  $A_{ij}$  e  $B_{ij}$ .
- Prodotto di matrici:  $C = AB$  dove  $A$ ,  $B$  e  $C$  sono matrici  $n \times n$ :

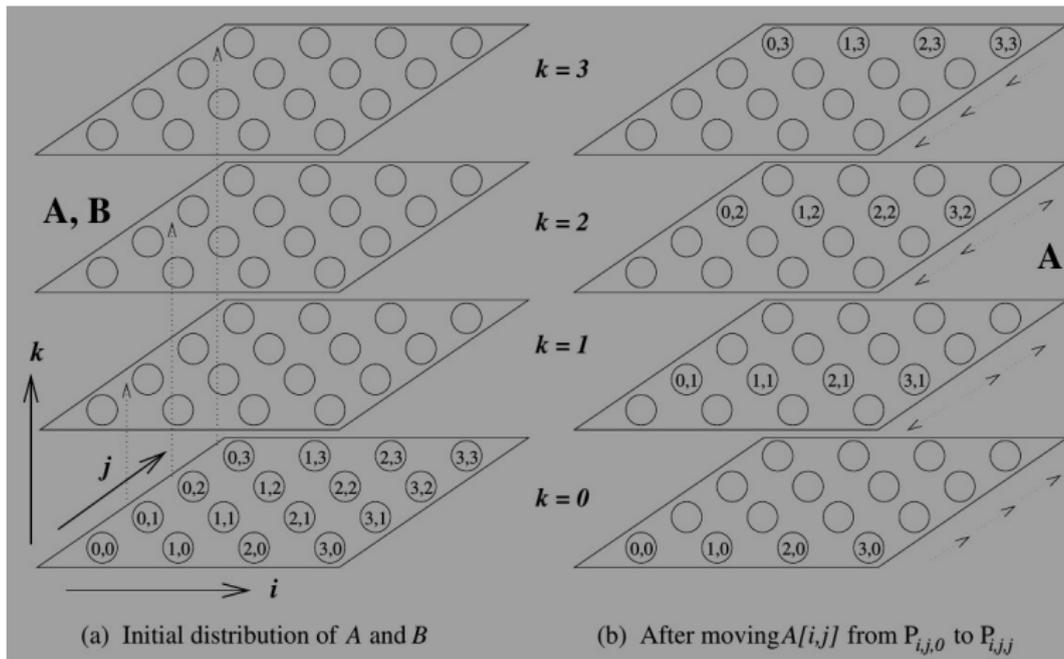
L'algoritmo:

- autori Dekel, Nassimi and Sahni (DNS...)
- il singolo elemento di matrice di  $C$  si ottiene  $C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$  (che richiede  $n$  prodotti per ogni elemento), ed essendoci  $n^2$  di questi elementi di matrice, implica che per calcolare la matrice prodotto servono ordine di  $n^2 \times n = n^3$  prodotti.
- Se ci fossero a disposizione  $n^3$  processi potrei fare calcolare **un prodotto** ad ognuno di essi. Ha senso quindi cominciare a pensare ad una griglia non piu' 2D ma 3D.
- le somme richiedono un tempo che cresce come  $\log n$  e puo' essere eseguito in ogni singolo processo.
- Servono  $n \times n \times n$  processi per implementare questo algoritmo

# Algoritmo DNS 2

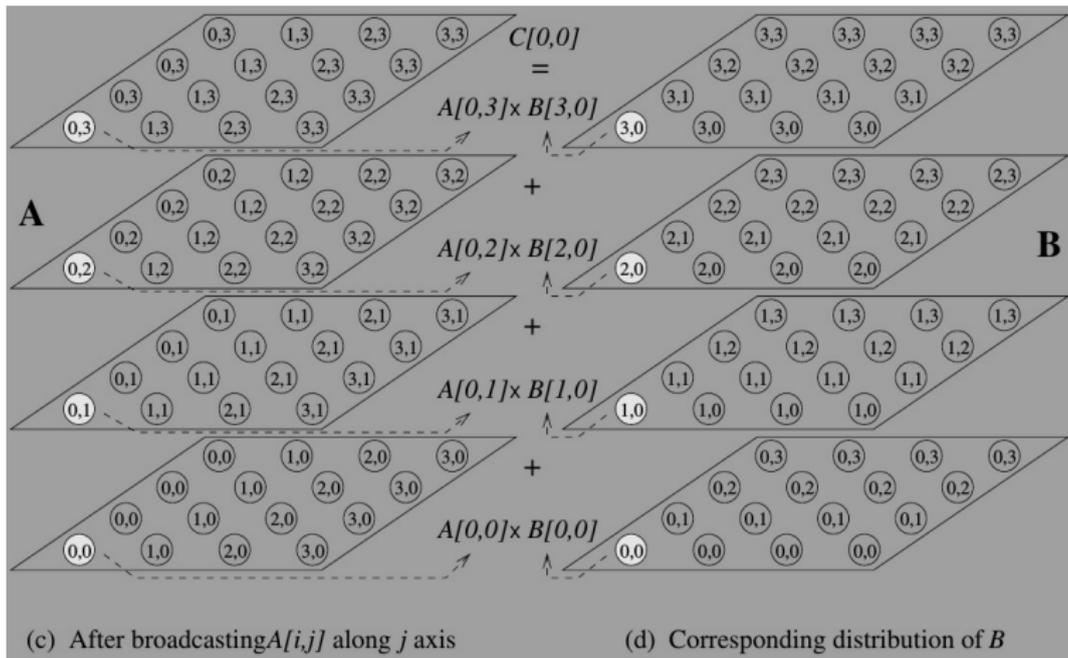
Come vengono distribuiti di dati:

- metto gli  $n \times n$  e.d.m. nello strato  $k = 0$
- sposto la riga  $k$ -esima in alto di  $k$  passi
- faccio broadcast a tutti gli elementi della medesima colonna



# Algoritmo DNS 3

- ogni processo fa una moltiplicazione (p.es.  $A_{02}B_{21}$ , che serve per l'edm  $C_{01}$ )
- per ogni  $i, j$  facciamo le somme su  $k$  (magari con un `MP_I_Reduce...`)



# Algoritmo DNS 4

Se la dimensione delle matrici  $A$  e  $B$  comincia a crescere (p.es.  $n = 10^5$ ), appare difficile poter trovare un supercomputer con  $n^3 = 10^{15}$  processori. Ha senso quindi, invece che inviare il singolo elemento di matrice ad ogni processo, inviare una sottomatrice.

- Sia  $p$  il numero di processi disponibili e  $p = q^3$  (la radice cubica di  $p$  e' un numero intero  $q$  inoltre  $n/q$  e' un intero)
- Le matrici  $A$  e  $B$  sono partizionate in blocchi di dimensione  $(n/q) \times (n/q)$ .
- Ogni matrice puo' quindi essere considerata come una griglia 2D di dimensione  $q \times q$  di sottomatrici.
- l'algoritmo, in questo caso e' identico al precedente, basta fare attenzione al fatto che le moltiplicazioni (somme) tra numeri diventano moltiplicazioni(somme) tra matrici.

## Algoritmo DNS 4 bis

L'efficienza in questo caso e':

- Il primo passo (e' un processo di comunicazione) effettuato per sia per  $A$  che per  $B$ , richiede  $t_s + t_w \frac{n^2}{q}$  tempo per ogni matrice.
- i **broadcast** (nello stesso strato) richiedono per ogni sottomatrice :  $2(t_s \log q + t_w \frac{n^2}{q} \log q)$  (e vanno effettuati sia per  $A$  che per  $B$ )
- il tempo del processo di riduzione (**somma**) richiede:  $t_s \log q + t_w \frac{n^2}{q} \log q$
- la **moltiplicazione** di due sottomatrici (di dimensione  $n/q \times n/q$ ) richiede  $\frac{n^3}{q^3}$
- in totale quindi il tempo parallelo e' circa:

$$T_p = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{3/2}} \log p \quad (24)$$

per confronto, Cannon invece era:

$$T_p = \frac{n^3}{p} + 2\sqrt{p} \left( t_s + t_w \frac{n^2}{p} \right) + 2 \left( t_s + t_w \frac{n^2}{p} \right)$$

per una bella presentazione:

[http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap8\\_slides.ppt](http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap8_slides.ppt)

## Funzioni Non-Blocking:

<https://cvw.cac.cornell.edu/mpip2p> ← Spiegazioni chiare sulle funzioni non-blocking.

- Per aggiungere maggiore flessibilità per i programmatori ( possibilmente evitare deadlock o ridurre le latenze) e' stata introdotta (in MPI 3.0) una nuova classe di funzioni MPI chiamate **non-blocking**.
- Queste funzioni **restituiscono subito** il controllo al programma che le ha chiamate, sta' quindi al programmatore l'assicurarsi la correttezza semantica dei messaggi spediti e ricevuti.
- **Svantaggio**: un codice con funzioni **non-blocking** e' piu' difficile!
- si devono utilizzare delle funzioni che controllino cosa sta' succedendo in modo da completare correttamente il passaggio di informazioni.

Alcune osservazioni:

- una funzione **non-blocking** puo' essere completata da una **blocking** (e viceversa)
- una **non-blocking**, rispetto ad una **blocking** ha un parametro aggiuntivo di tipo: `MPI_Request`. Questa e' una handle ad un cosiddetto oggetto **opaco** (la cui forma e dimensione non e' visibile all'utente) che identifica lo stato dell'operazione di comunicazione: serve per *coniugare* l'operazione che comincia la comunicazione con quella che la termina (occhio a non confondere questo oggetto con `MPI_Status`)
- `MPI_Request` deve essere **completato** prima che i (send/receive) buffer vengano modificati o acceduti (altrimenti si ha un comportamento non predicibile).

## Funzioni di completamento

Come si controlla una funzione **non-blocking**?

Una funzione non-blocking deve essere **completata**. Per completamento si intende che la funzione e' libera di ritornare e riscrivere il buffer... **indipendentemente** dal fatto che il messaggio sia stato ricevuto! Il messaggio in se' passa in secondo piano, il programmatore "completa" le **non-blocking** con le Request

Se si vuole che il proprio codice sia conforme agli standard bisogna **SEMPRE** usare una funzione di completamento dopo una **non-blocking**. Per completare si puo' usare:

- MPI\_Wait
- MPI\_Test
- MPI\_Request\_free

Lo standard consente infatti di postporre la trasmissione dei dati fino ad una wait/test.

Terminologia delle Request:

- una request e' **null** se ha come valore `MPI_REQUEST_NULL`
- se la request non e' associata a nessuna comunicazione in uscita e' considerata **inactive**
- una request e' considerata **active** se non e' ne **null** ne **inactive**

Possibile **problema**: il numero di handle alle request e' limitato, se **non** si usano delle funzioni di completamento, si puo' esaurire questo numero e far crashare il codice!

## *Isend e Irecv*

La sintassi di `MPI_Isend` e' praticamente identica a quella di `MPI_Send` (anche il nome non e' molto diverso...)

```
int MPI_Isend (void *buf ,
 int count ,
 MPI_Data type datatype ,
 int dest ,
 int tag ,
 MPI_Comm comm,
 MPI_Request *request) // parametro in output
```

La variabile `request` diventa **attiva** solo quando la funzione `MPI_Isend` e' stata chiamata (e a quel punto puo' quindi essere usata per attivare una `MPI_Irecv`).

La sintassi di `MPI_Irecv` e':

```
int MPI_Irecv(void* buf,
 int count,
 MPI_Datatype datatype,
 int source,
 int tag,
 MPI_Comm comm,
 MPI_Request *request)
```

# MPI\_Wait()

---

```
int MPI_Wait(MPI_Request *request, // input
 MPI_Status *status) // output
```

---

- Questa funzione (di tipo **blocking**) *ferma temporaneamente* il **processo** finché l'operazione identificata dalla `request` è completata.
- La `MPI_Request` dopo la chiamata viene deallocata, e la corrispondente handle viene settata su `MPI_REQUEST_NULL`
- In questo senso la coppia:  
`MPI_Isend + MPI_Wait` (una di seguito all'altra) è equivalente ad un `MPI_Send` (anche dal punto di vista delle tempistiche)
- Il vantaggio è dovuto al fatto che il **bravo programmatore** può far fare delle cose al codice tra il `MPI_Isend` e il `MPI_Wait`.
- Per esempio si mette la `MPI_Wait` solo **prima** che il buffer da mandare debba essere riutilizzato.
- Il risultato della funzione è restituito dalla variabile `status`

# MPI\_Test()

Un altro modo di controllare una funzione **non-blocking** e' tramite la funzione:

```
int MPI_Test(MPI_Request *request, //input
 int *flag, //output vera se l'operazione
 //specificata da request e' completata
 MPI_Status *status) //output
```

A differenza da `MPI_Wait` questa funzione **non** ferma il processo: restituisce subito una `flag` che indica se la (`request`) e' completata (nel caso in cui la `request` sia stata prodotta da un `MPI_Isend` quando il messaggio e' stato inviato correttamente). Se la funzione e' **completa** allora il valore della handle della `request` e' settato su `MPI_REQUEST_NULL` (e la `request` stessa viene dellocata).

Altrimenti se la `flag = falsa`, il valore di `request` rimane lo stesso.

Spesso, si usa `MPI_Test` in un loop:

- si controlla se la comunicazione e' completata
- si fa qualcosa d'altro nel frattempo
- si ricontrolla, se la `flag` non e' true, si continua a fare altro
- cosi' via...

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request);
do {
 MPI_Test(request, flag, status);
} while (flag != 1);
```

## *consigli utili*

Per un non-blocking **send**:

- NON si deve fare update
- NON si deve riallocare
- NON si deve liberare il buffer

tra il send e il completamento (`MPI_Wait`, `MPI_Test`, `MPI_Request_free`).

**Leggere** il buffer, invece, non causa problemi!

Per un non-blocking **receive**

- NON si deve accedere
- NON si deve riallocare
- NON si deve liberare il buffer

tra la chiamata della funzione e il suo completamento (`MPI_Wait`, `MPI_Test`, `MPI_Request_free`).

# Non-blocking ring

Partire dal codice [ring](#) solito, crearne uno nuovo con le funzioni non-blocking, usando (a scelta) `MPI_Wait` o `MPI_Test`.

```

MPI_Request request=MPI_REQUEST_NULL;
MPI_Status status;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numero_processi);

// ===== FIRMA DELLE FUNZIONI =====
// MPI_Wait (puntatoreRequest, puntatoreStatus)
// MPI_Irecv (puntatoreMessaggio, quantiMessaggi, tipo,
// rank_mittente, tag, comunicatore, &puntatore arequest);
if (rank != 0) // tutti si mettono in ascolto tranne lo 0
{
 // usare funzione di ricezione non blocking
printf("Il processo %d ha ricevuto il messaggio %d dal processo %d\n",rank,messaggio,rank-1);
 messaggio = messaggio +1;
}
// tutti si cominciano ad inviare (anche lo 0)
// usare funzione non blocking di invio

if (rank == 0) // solo lo zero ascolta dall'ultimo rank
{
 // usare funzione non blocking di ricezione
printf("Il processo %d ha ricevuto il messaggio %d dal processo %d\n",rank,messaggio,
 numero_processi-1);
}

```

## Non-blocking ring: soluzione

Partire dal codice [ring](#) solito, crearne uno nuovo con le funzioni non-blocking, usando (a scelta) `MPI_Wait` o `MPI_Test`.

```

MPI_Request request=MPI_REQUEST_NULL;
MPI_Status status;
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numero_processi);

if (rank != 0) // tutti si mettono in ascolto tranne lo 0
{
 MPI_Irecv(&messaggio, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait (&request, &status);
 printf("Il processo %d ha ricevuto il messaggio %d dal processo %d\n", rank, messaggio,rank-1);
 messaggio = messaggio +1;
}

// tutti si cominciano ad inviare (anche lo 0)
MPI_Isend(&messaggio, 1, MPI_INT, (rank + 1) % numero_processi, 0,MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

if (rank == 0) // solo lo zero ascolta dall'ultimo rank
{
 MPI_Irecv(&messaggio, 1, MPI_INT, numero_processi - 1, 0, MPI_COMM_WORLD, &request);
 MPI_Wait (&request, &status);
 printf("Il processo %d ha ricevuto il messaggio %d dal processo %d\n", rank, messaggio,
 numero_processi-1);
}

```

# *Fine di MPI*

Abbiamo visto:

- cos'è MPI e cosa sono i communicator
- le 6 funzioni principali
- alcune funzioni collective
- alcune funzioni point-to-point avanzate
- come costruire e suddividere i communicator
- moltiplicazione di matrici: algoritmo Cannon
- funzioni non-blocking

**Non** abbiamo visto:

- La gestione degli errori delle funzioni
- parallel I/O
- intEr-communicators
- .... molto altro!