Esercitazioni di Calcolo Parallelo

Paolo Avogadro

DISCo, Università di Milano-Bicocca
U14, Id&aLab T36
paolo.avogadro@unimib.it
Aula Lezione T014,
edificio U14

- Martedi' 15:30-17:30
- Mercoledi' 10:30-12:30

nota per me stesso

compila questa presentazione con pdflatex, altrimenti ci sono errori.

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 portatili, con cui poter fare gli esercizi ed interfacciarsi con il server. Qualora non tutti avessero a disposizione un portatile si possono scrivere i codici in collaborazione con altri studenti.
Numero di lezioni 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

Introduzione a MPI

MPI = Message Passing Interface.

Cos'e'?

- E' 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) gestire lo scambio di messaggi tra vari "centri di calcolo".
- diverse implementazioni prensentano lievi differenze, che pero' possono creare grossi problemi, ovvero lo stesso codice con due implementazioni differenti puo' non funzionare.
- I codici MPI possono essere del tipo SPMD: Single Program Multiple Data (noi ci concentreremo su questo paradigma)
- e' 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

- Puo' 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 e' molto generico, esempio possono essere i vari core di un laptop, o computer veri e propri di una rete locale o meno, o i nodi di un supercomputer).
- In qualche modo ha senso pensare che ognuno dei "computer" conivolti abbia a disposizione la propria memoria, ed esegua una istanza del codice: questa istanza verra' 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 che tante piu' sono le unita' di calcolo coinvolte, tanto piu' la comunicazione puo' 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".

ATTTENZIONE: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**
- 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++, ovvero oltre ai comandi del linguaggio usato, per esempio il C, vengono aggiunte delle funzioni dello standard MPI (e altro). In questo corso ci limiteremo ad esempi di 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 (attenzione 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

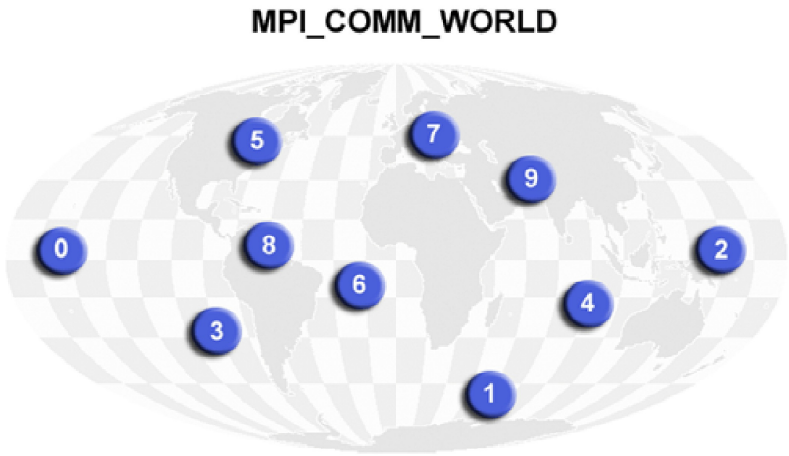


Figure: rappresentazione grafica del “mondo comune”.

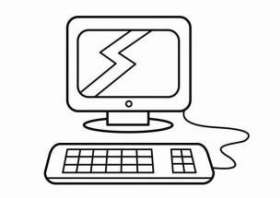
Il *Communicator* e’ una delle strutture piu’ importanti per MPI.
In un certo senso e’ l’universo in cui vivono i vari processi e attraverso a cui comunicano.

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*; per esempio l’identificativo del communicator stesso. 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).

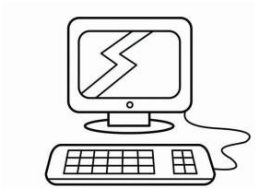
Size e Rank

Dimensione del communicator: SIZE = 5

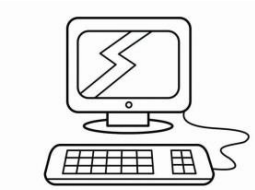
Rank = 0



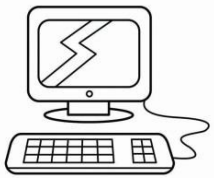
Rank = 1



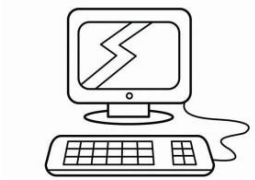
Rank = 3



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** se appartengono allo stesso communicator

un po' di gergo

1. **process** (o processo) un'istanza di esecuzione di un programma (attenzione non e' un nome *standard* e' solo il piu' diffuso, pero' alle volte viene chiamato **task**, **clone**, etc... peccato che taluni di questi termini abbiano altri significati specifici, per questo nel corso useremo solo **process** mentre **task** sara' un'altro tipo di cosa). Ha un indirizzo specifico. E' differente da altre istanze di esecuzione nel senso che accede a risorse differenti. E' il codice che gira su una singola cpu o nodo. I proccessi possono essere visti in `/proc/`.
2. **funzione**/ **routine** / **subroutine** / **procedura** sono tutti sinonimi, in C si hanno solo funzioni, mentre in FORTRAN le subroutine sono essenzialmente delle funzioni con tipo associato `void` (ergo non restituiscono un valore, come invece fanno le `function`).
3. **Wall time** e' il tempo dall'inizio del calcolo della prima cpu alla fine dell'ultima cpu.
4. **concurrency** e' il potenziale parallelismo. Per esempio un programma puo' essere scritto per dare luogo ad un calcolo parallelo, ma essere eseguito su una macchina che non ha risorse multiple, in questo caso vi e' concurrency ma non parallelismo.
5. **competition** se un processo richiede un uso esclusivo di una risorsa (per esempio un array).
6. **asyncronous message passing**: se l'ordine di invio dei messaggi NON e' obbligato ad essere lo stesso dell'ordine di ricezione
7. **syncronous message passing**: se l'ordine di invio e ricezione devono essere identici.
8. **MPI handles** sono gli strumenti (spesso sono i nomi) per maneggiare oggetti MPI, come i processi, i messaggi, i gruppi, etc...
9. **MPI Datatype** sono i tipi usati dalle funzioni MPI (che avranno spesso dei corrispondenti nei tipi di FORTRAN, C etc, ma possono essere dei nuovi tipi, propri di MPI.

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:

- installre 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`, (opprure: `sudo yum install openmpi-devel.i686`) probabilmente con una versione piu' recente 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 ti 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
```

nella versione installata sul mio pc locale di `openMPI`, invece che `module add` si deve usare `module load`

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.

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) perche' in questo modo si ha maggiore portabilita'.
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 fare 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, va messo invece: `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.

```c
#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.

```c
#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 sara' 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?

MPI_Comm_rank

questa funzione viene specificata nel seguente modo:

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

tipo — nome communcator — tipo — puntatore al nome variabile dove metto il rank

- Serve a determinare il **rank** del processo chimante all'interno del communicator chiamato `comm`, l'informazione viene messa nella variabile `miorank` (nome di esempio ma sceglibile a piacere).
- 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 anche come il suo indirizzo quando si spediscono o ricevono messaggi da esso.
- 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.

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 `comm`).
- il secondo argomento della funzione (`nproc`) e' una variabile di output che conterra' la grandezza (**size**) del communicator stesso. Occhio che qui si passa un **puntatore**.
- 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(" bla bla %d", rank);`

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

MPI_Init(&argc, &argv);



MPI_Finalize();
return 0;
}
```

diversi processi-diversi dati:

- MPI si occupa di gestire il **trasferimento** di messaggi ed informazioni tra processi
- Quando si lancia un codice MPI, per un supercomputer, si dovra' scrivere uno **script** che indica quale codice viene lanciato, quali sono i file di input etc...
- in generale ci sara' uno scheduler che si occupa di gestire le aree di memoria assegnate ad ogni processo, e' pero' possibile utilizzare alcune funzioni MPI per fare del parallel-IO, ovvero si puo' gestire una notevole mole di lavoro di scrittura e lettura in modo parallelo.
- Nei casi piu' semplici, come quelli visti da noi, si sfrutta il **rank** per decidere che un particolare processo debba accedere ad un file particolare e poi si puo' fare circolare l'informazione raccolta tramite le normali funzioni di message passing... che vedremo nelle prossime slide.

Spedire Messaggi

Un messaggio e' composto di:
envelope (busta): con sorgente (rank del sender), destinazione (rank del processo receiver), communicator (nome del dove comunicano i processi), tag (intero non negativo usato per identificare il messaggio, se il sender ne manda piu' di uno per volta.
body (corpo):

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

In qualunque **point-to-point** communication ci sono alcune cose **OBBLIGATORIE**:

- il sender deve dichiarare ESPLICITAMENTE il rank del *receiver* (numero identificativo del processo)
- il receiver deve essere dichiarato ESPLICITAMENTE il rank del *sender*

La lunghezza del messaggio deve essere controllata indicando *esplicitamente* il numero di oggetti inviati e il numero di oggetti che si aspetta di 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.

Gergo 2

- **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 (nei tutorial che si trovano in rete), 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.

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 mettiti 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`

Nota: 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:

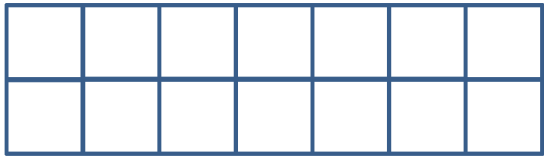
```
MPI_Send( &ciccio[2] , 3, MPI_INT, 1, 999, MPI_COMM_WORLD) ;
```

Visualizziamo invio messaggi

Cervello_uomo

Persona =1

Cervello_donna



Persona =2

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

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:

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{pmatrix} \tag{1}$$

Come vengono immagazzinati in memoria questi dati?
Pericolo: in C l'array `a`: 3×3 viene stoccato per **righe**:

$$a[0][0] \quad a[0][1] \quad a[0][2] \quad a[1][0] \quad a[1][1] \quad a[1][2] \quad a[2][0] \quad a[2][1] \quad a[2][2] \tag{2}$$

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

$$\begin{pmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{pmatrix} \tag{3}$$

$$a[0][0] \quad a[1][0] \quad a[2][0] \quad a[0][1] \quad a[1][1] \quad a[2][1] \quad a[0][2] \quad a[1][2] \quad a[2][2] \tag{4}$$

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 finisce di funzionare 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

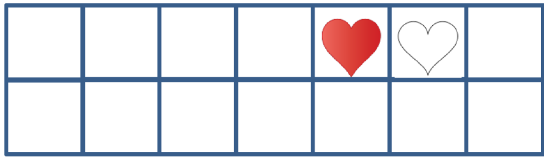
Cervello_uomo

Persona =1

Buffer = **Cervello_uomo**[2]
quantiDati = 2
tipoDati = cuore
rank**Destinatario** = 2
tag = 1

nomeComm = Whatsapp

Cervello_donna



Persona =2

Buffer = **Cervello_donna**[5]
quantiDati = 2
tipoDati = cuore
rank**Mittente** = 1
tag = 1
Mpi_status = ignore
nomeComm = Whatsapp

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... un po' piu' in la'

- come nell'esercizio precendente 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`

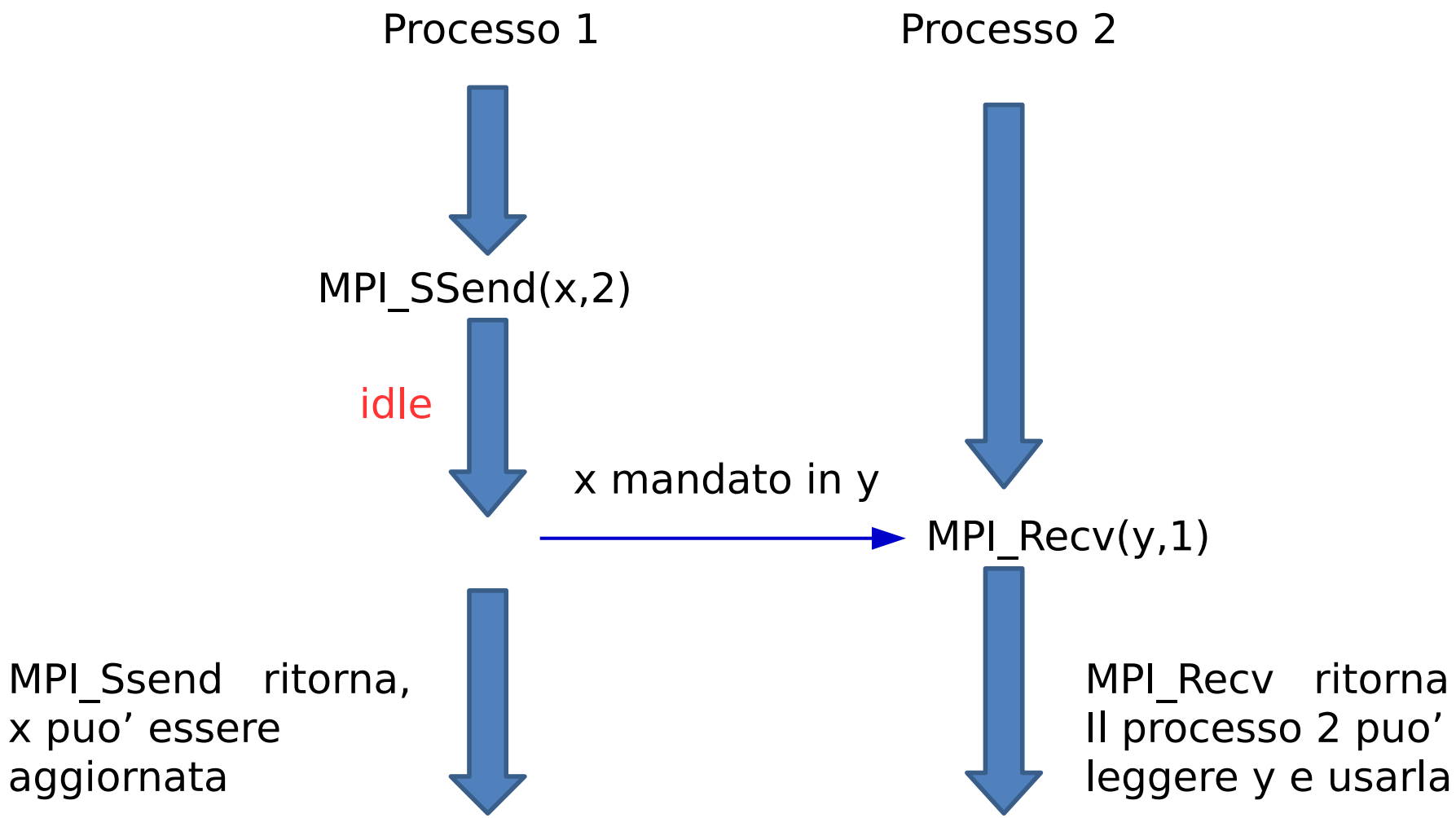
un altro tipo di Send

UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

MPI_Ssend: sincronizzato

- Puo' cominciare ad operare **prima** che venga lanciato il `receive` dall'altro processo (altri tipi di send invece necessitano che ci sia gia' un receive in ascolto)
- il `Receive` deve cominciare prima che `Ssend` spedisca qualcosa
- E' il modo di mandare piu' **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
- Puo' genereare un overhead notevole per la sincronizzazione

MPI_Ssend: sincronizzato



un altro Send ancora...

MPI_Bsend: bufferizzato

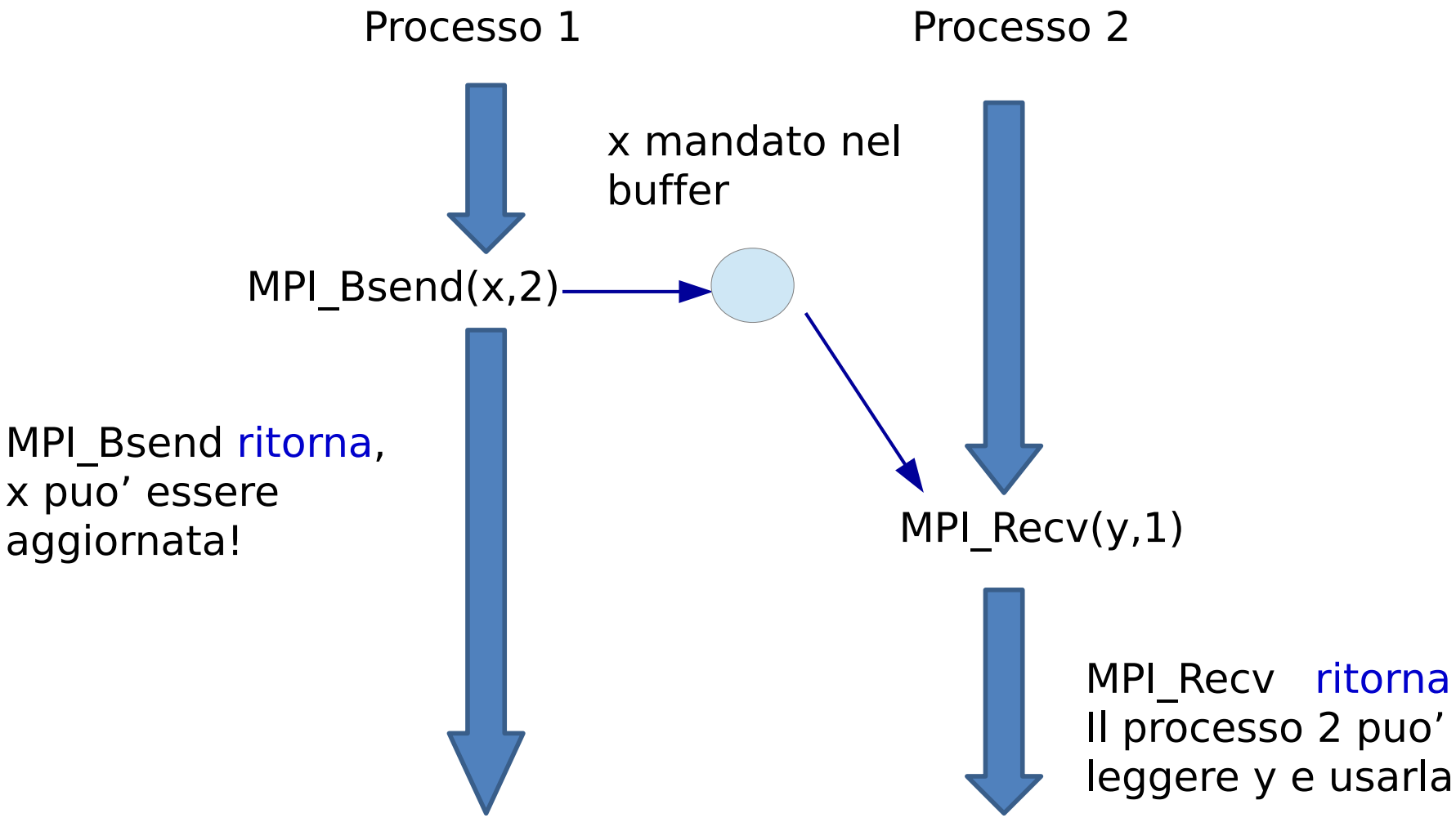
- il send puo' essere completato prima che il `receive` cominci!!!
- bisogna alloccare in modo esplicito un buffer (spazio di memoria) con `MPI_Buffer_attach`
- puo' generare errore se si manda un messaggio piu' grande del buffer
- non ha overhead di sincronizzazione (ma bisogna usare memoria aggiuntiva,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

MPI_Bsend: buffering



... e quindi MPI_Send?

`MPI_Send` e' una funzione chiamata **blocking**: il codice non continua finche' la funzione non ritorna correttamente. Questo significa che finche' il processo di invio del messaggio non e' completato il sistema si blocca....

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 alloccare 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_world()`
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!)

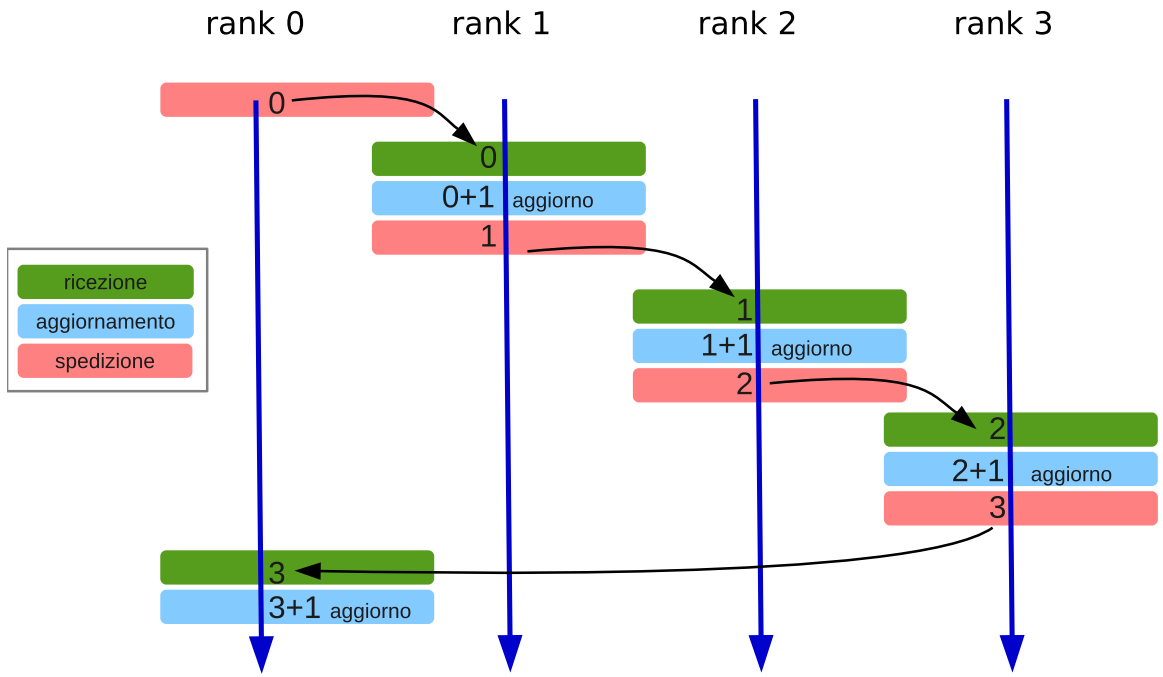
Come gia' accennato scrivere un codice MPI e' complicato (un ordine di grandezza piu' che un codice scalare). E' molto difficile scrivere un codice *seriale* e, a **posteriori**, parallelizzarlo. E' meglio progettare il codice direttamente perche' funzioni in parallelo, e ancora meglio conoscendo bene le caratteristiche della macchina su cui dovra' girare. Queste sono alcune delle ragioni:

- vorremmo scalabilita' 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)



Telefono senza fili: ring

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

```c
#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();
}
```

Ring: ovvero

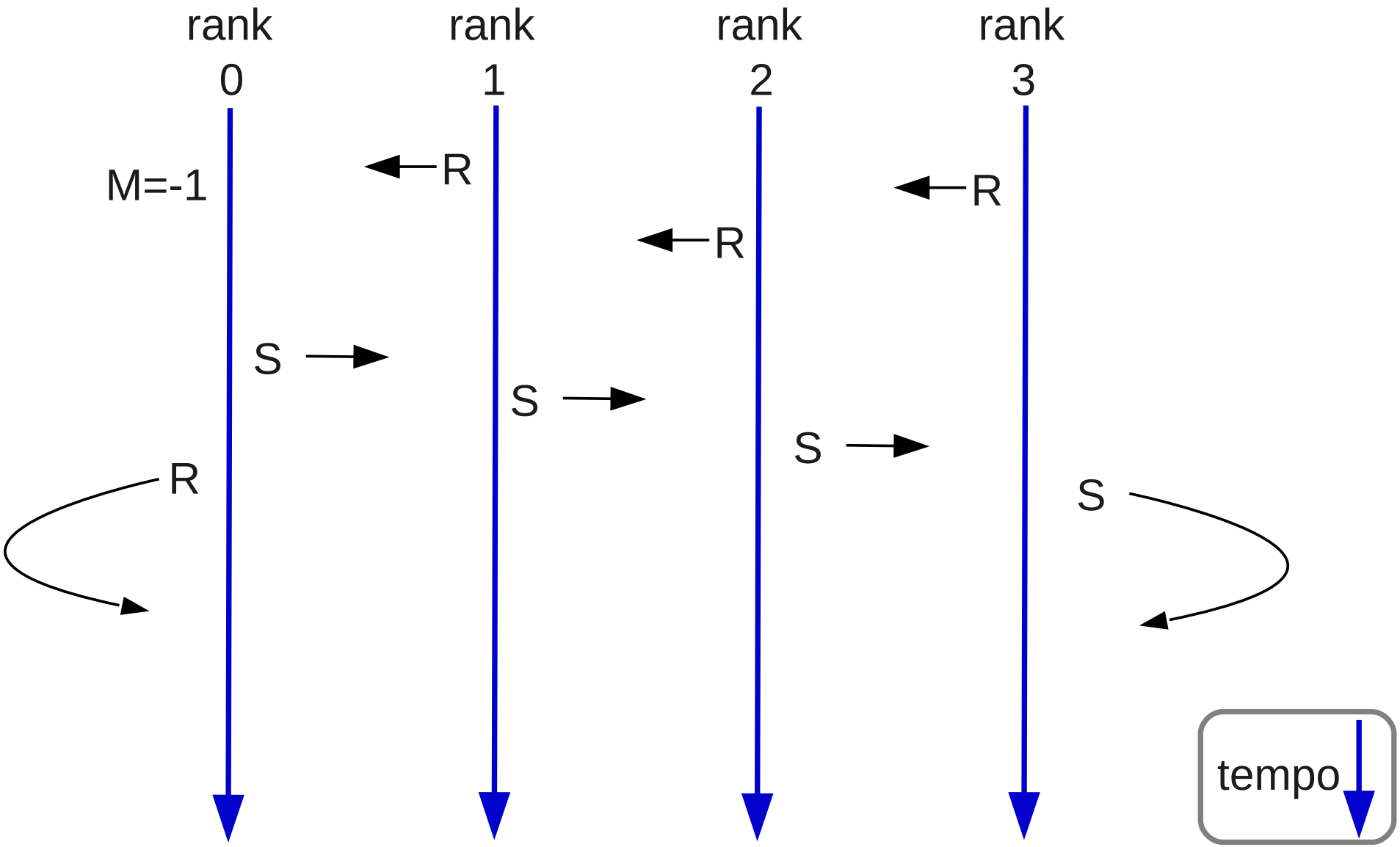
riassumiamo l'ordine delle chiamate del codice ring:

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 inverto l'ordine di una di queste chiamate?
- Testiamo la dimensione del buffer definito implicitamente da `MPI_Send` nell implementazione openMPI.

La tempistica di esecuzione:



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 e' che riguardano molti processi per volta.
- Se un processo chiama una funzione collettiva, deve sempre essere specificato in quale communicator ci si trova. Questo perche' **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** $\Longleftarrow$ IMPORTANTE!!!!!

MPI_Barrier

Questa funzione e' pensata per crare 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' quello di fare un primo codice con una collective `MPI_Barrier`, in cui:

- tutti i processi diversi dal processo 0 scrivono il proprio rank
- la barriera sincronizza i processi
- **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_Finalize();  // finalizza
}
```

MPI_Bcast

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 degli altri 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 modo di fare puo' indurre in errore in quanto si devono porre degli `if` davanti alla chiamata del `MPI_Bcast` e 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


 MPI_Finalize();
}
```

Differenze tra Broadcast e Scatter

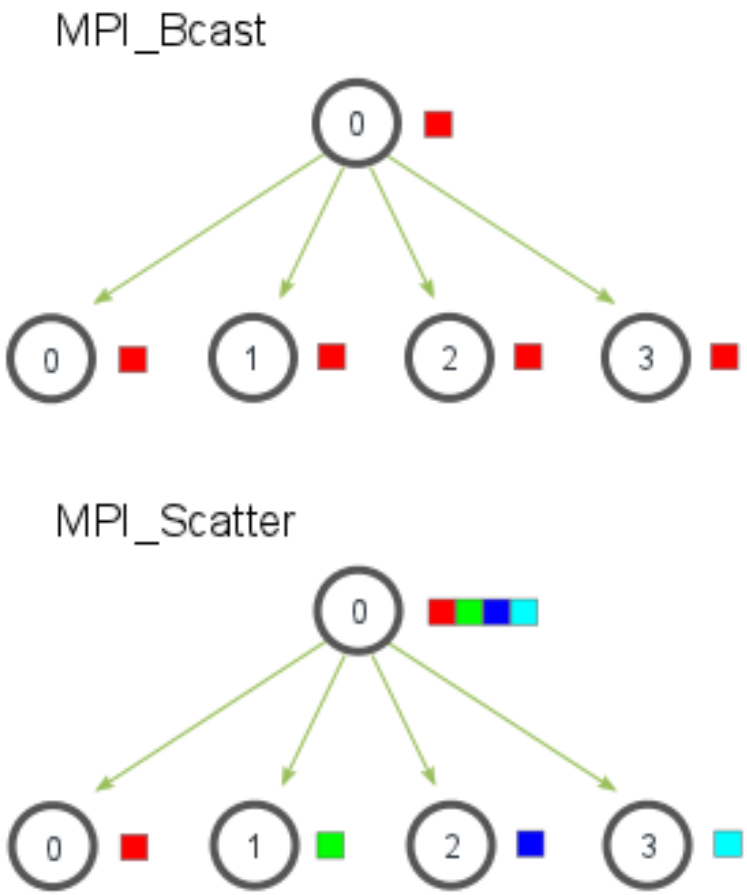


Figure: diagramma della differenza tra Broadcast e Scatter (da `mpitutorial.com`)

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_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)              //
```

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

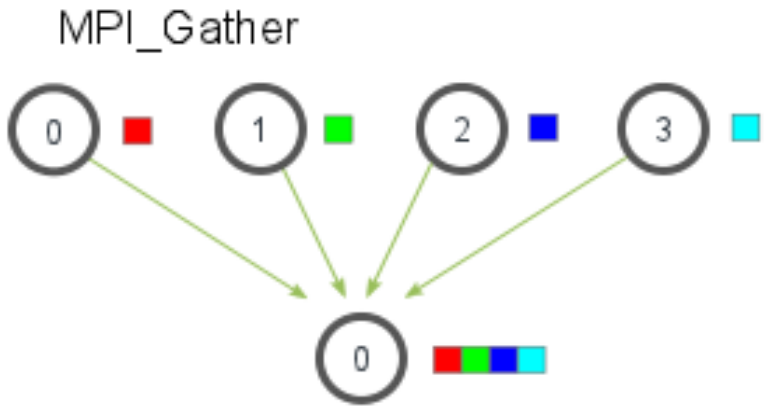


Figure: mpitutorial.com

MPI_Gather 2

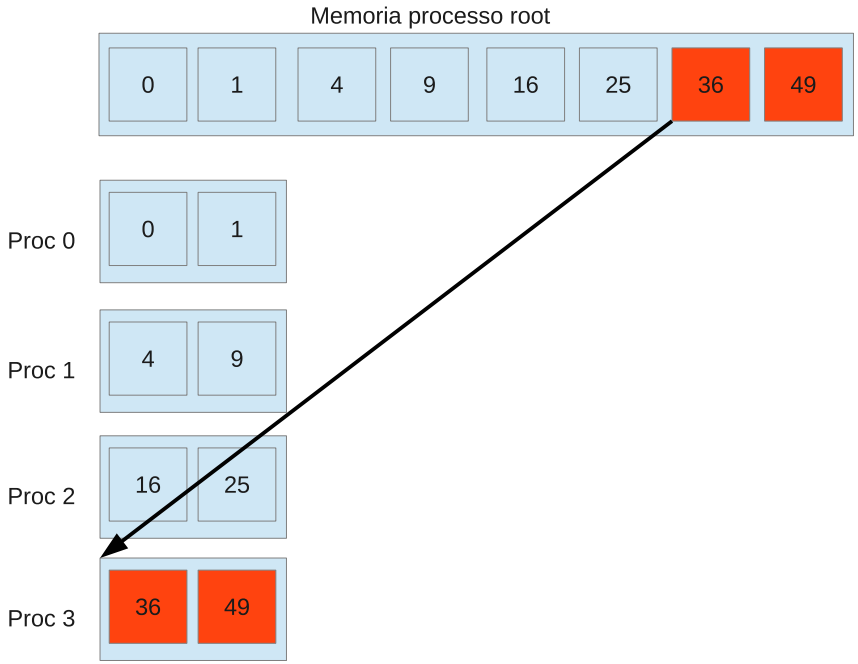
- e' l'inverso di scatter
- e' una collective communication
- dato che e' una collective communication e' sincronizzata
- dato che e' sincronizzata **TUTTI** i processi devono chiamarla
- occhio che i parametri che vengono inseriti sono diversi per ogni processo in particolare il buffer dei dati da ricevere 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).
- dato che bisogna definire dove arrivano i dati, in C si puo' assegnare il tipo NULL

MPI_Scatter 2

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

- Se c'e' 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 di solito il numero `send_count` e' equivalente al numero di elementi dell'array `send_data` diviso il numero dei processi (in questo modo ad ogni processo vengono assegnati lo stesso numero di dati nel seguito si vede come fare per numero dati non divisibile esattamente per il numero dei processi).
- Problema di flessibilita': Se voglio inviare 3 unita' elementari al processo 1 e 4 unita' al processo 2 come faccio?
- Problema di portabilita': ad ogni processo vengono inviati **esattamente** `send_count` messaggi elementari... se N e' il numero di processi, allora il nostro array deve contenere almeno N $\times$ `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 da' errore, ma invia dei messaggi sbagliati ai processi!

MPI_Scatter


Memoria processo root
0
1
4
9
16
25
36
49
Proc 0
0
1
Proc 1
4
9
Proc 2
16
25
Proc 3
36
49

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

```c
#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 ;   // azzera il vettore
          di arrivo

    //                    CODICE DA INSERIRE
    //                    CODICE DA INSERIRE

    MPI_Finalize();
    return 0;
}
```

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://cvw.cac.cornell.edu/MPIcc/scatterv?AspxAutoDetectCookieSupport=1`

Utilizzo Scatterv

In particolare:

- 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.

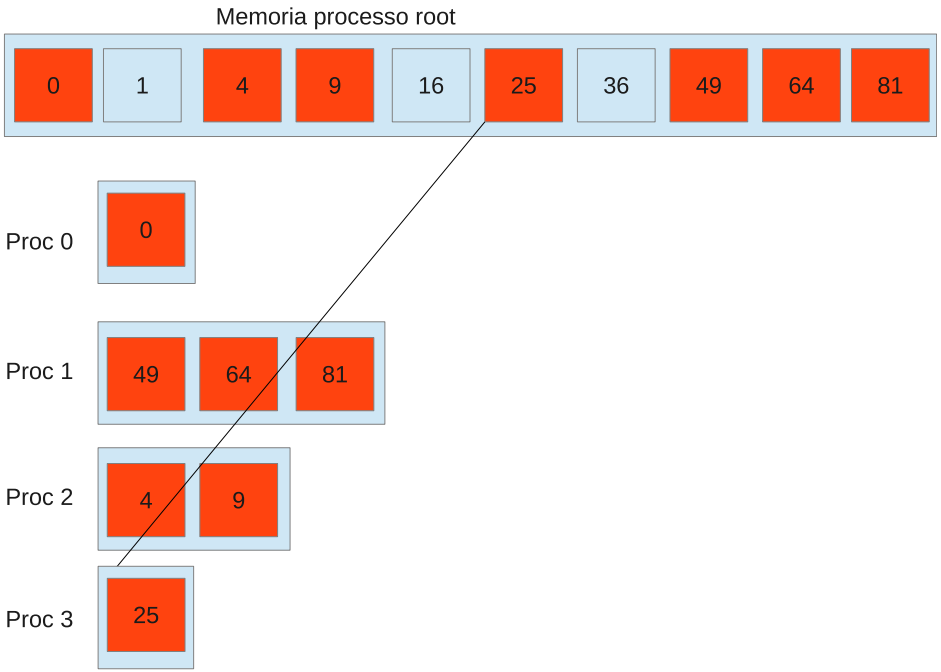
chiaramente, questi due array devono contenere un numero di ingressi uguale (o maggiore) al numero di processi coinvolti.

```
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:

```
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,...,Ngrande`
- ogni processo contiene un array piccolo (`rbuf`) dove inviare pezzi dell'array grande.
- viene inviato un messaggio ad **ognuno** dei processi del calcolo (scalabilita')
- 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

Domande:

- chi (quale processo) deve avere definito l'array da cui vengono inviati i dati `sbuf`?
- chi deve avere l'array dove vengono ricevuti i dati `rbuf`?
- chi deve avere l'array che definisce lo spostamento (displacement) `displs`?
- chi deve avere l'array che definisce il numero di oggetti mandati ad ogni processo `scounts`?
- perche' `rcount` (il numero di oggetti ricevuti) e' un intero e non un array?
- un processo che deve ricevere, come fa a conoscere il numero di oggetti che deve ricevere (quali soluzioni si possono adottare)?

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?
```

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  );  // commiunicator
```

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

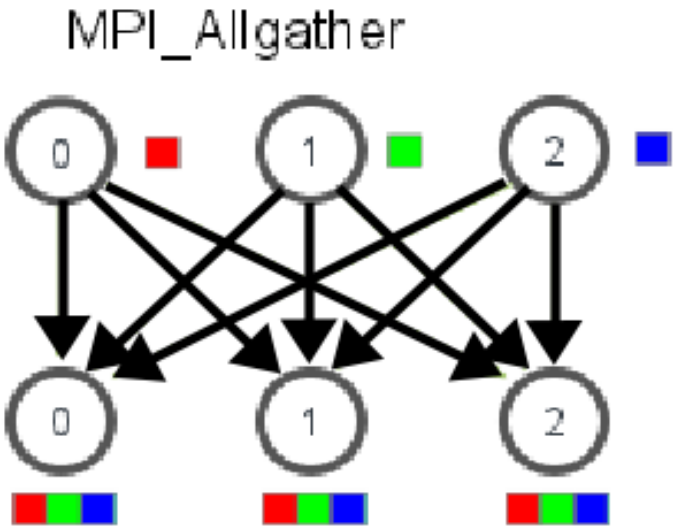
```
int MPI_Gatherv( void*         sbuf,
                 int          scount,
         MPI_Datatype         stype,
                void*         rbuf,
           const int    rcounts[],
           const int     displs[],
         MPI_Datatype         rtype,
                 int          root,
            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, ovvero prende un insieme di info **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 MPI ovvero fanno il lavoro sporco di ottimizzazione per mandare i dati ai vari processi (altrimenti basterebbero le poche funzioni di base come send e receive).

- 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.



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 recvcount , // 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 dovra' avere abbastanza spazio per stoccarli,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 queto 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;          // initializza 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();
}
```

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 3

(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:

(10	11	12	13	14	15	16	17)	*rank* 0	(10	11	20	21	30	31	40	41)
(20	21	22	23	24	25	26	27)	*rank* 1	(12	13	22	23	32	33	42	43)
(30	31	32	33	34	35	36	37)	*rank* 2	(14	15	24	25	34	35	44	45)
(40	41	42	43	44	45	46	47)	*rank* 3	(16	17	26	27	36	37	46	47)

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 partenza 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 arrivo sara' piu' piccolo di quello di partenza).

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 * recvcount * extent(recvtype),
         recvcount, 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();
}
```

Reduce operations

Il cuore di un calcolo parallelo e' dato dal fatto che vengono eseguiti contemporaneamente molti calcoli,

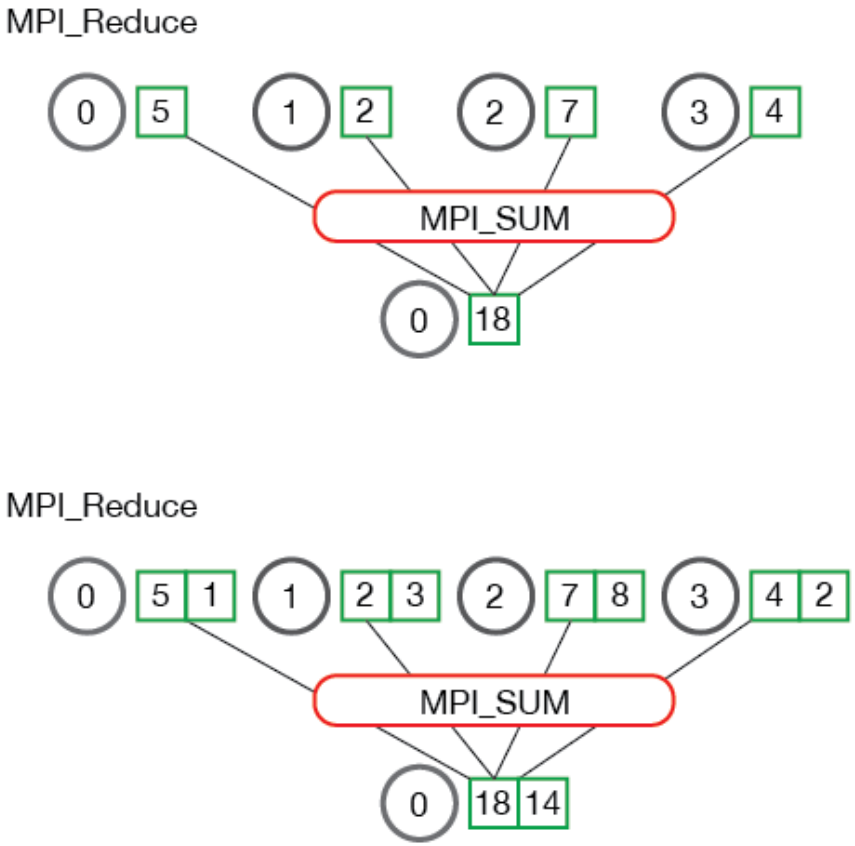
- alla fine pero' potrei non essere interessato ad avere p.es 100 000 numeri
- devo in qualche modo riassumerli in quantita' che siano leggibili e chiare per una singola persona.

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 risultati 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**.



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 massimo.
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 `hadle` 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' associtiva di un operatore O: dati 3 elementi a, b e c su cui O puo' operare (e' una operazione binaria):

$$(aOb)Oc = aO(bOc) \tag{5}$$

Reduce esempio

- scrivere un codice dove ogni processo ha un array di dimensione 2 contentente, nell'ingresso 0 il proprio rank, e nell'ingresso 1 la quantita' rank*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;
}
```

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 in parallelo al momento della chiamata
- l'oggetto 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();
}
```

Moltiplicazione tra Matrici: ripasso

Righe per colonne:

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

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

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:

$$c_{ij} = \sum_{\mathbf{k}} a_{i\mathbf{k}} b_{\mathbf{k}j}$$

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:

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

```c
#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, perche' qui il primo elemento e' 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** e' 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]` (contentente 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!
```

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: Ha senso che la matrice *A* sia presente in ogni processo, ma solo dei 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!)

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 processso 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
```

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 verra' 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` sara' 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;             // azzera 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* e' pero' piena di zeri, e solo una parte contiene gli elementi della matrice prodotto. Solo questa parte andra' 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 gia' al punto giusto di dove andranno messi nella matrice *C* finale (il resto della matrice *C* contenuto in ogni processo e' 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: alloccare 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 in modo da per passare 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 $i, j = 1, 2, \ldots, \sqrt{p}$ 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 $i, j = 1, 2, \ldots, \sqrt{p}$ con dimensione $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$

Cannon: le matrici

Per esempio:

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

$$A = \left(\begin{array}{ccc|ccc|ccc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} \\ \hline a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} \end{array}\right) =$$

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

In questo caso le sottomatrici A_{ij} hanno dimensione 3x3 ($\sqrt{p} = 3$, $n = 9$, $n/\sqrt{p} = 3$).

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:

Nome del processo	che cosa deve **contenere** all'inizio	cosa vogliamo **calcolare**
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.

Cannon

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 $k = 0, ..., \sqrt{p} - 1$

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):

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

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 obbiettivo e' pero' quello di calcolare tutta la sottomatrice C_{12} (all'interno del processo P_{12}). La formula per ottenerlo e':

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

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

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

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

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

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

$$C_{12} = \quad A_{12} \quad B_{12} \tag{10}$$

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 a quello immediatamente vicino e vediamo se, con un po' di fortuna (similmente di cubo di Rubik) otteniamo le sottomatrici volute.

Ancora Cannon?

Facciamo uno shift delle sottomatrici all'interno della matrice logica di m posizioni:

$$A_{ij} \overset{\text{m passi}}{\longleftarrow} A_{i(j+m)_{\sqrt{p}}} \tag{11}$$

$$B_{ij} \overset{\text{l passi}}{\uparrow} B_{(i+l)_{\sqrt{p}},j} \tag{12}$$

(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 delle sottomatrici di A e gli indici di riga delle sottomatrici di B siano le stesse:

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

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

$$(j+i+k)_{\sqrt{p}} = (i+j+k)_{\sqrt{p}} \tag{14}$$

(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**:

$$A_{ij} \quad B_{ij} \tag{15}$$

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

$$A_{i\mathbf{x}} \quad B_{\mathbf{x}j} \tag{16}$$

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

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

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

$$B_{ij} \overset{\text{l passi}}{\uparrow} B_{(i+l)_{\sqrt{p}}} \tag{18}$$

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:

$$m = j + k \Rightarrow A_{i;(j+i+k)_{\sqrt{p}}} \tag{19}$$

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

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.

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

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!
Quando 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} gia' 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 communcator 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.

ATTENZIONE: in MPI tutte le risorse sono in numero limitato, e' per questo che si rimuovono i comunicatori 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 `nproc` (numero processi) 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. Il processo con la **key** piu' piccola all'interno di un sotto-communicator avra' rank 0, poi rank 1, etc... a due processi con la stessa key viene assegnato il `nuovo_rank` minore a quello che ha il `vecchio_rank` minore.

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 dal communicator, con colore 0, dal rank 2 al rank 1

```c
#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();
}
```

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 puo' essere associata alla reale distanza fisica dei processi ma e' estremamente difficile ottenere dei risultati in questo senso)
Vogliamo quindi che si abbiano delle coordinate virtuali associate al `rank`:

$$\texttt{rank} \longrightarrow (i_x, i_y) \tag{21}$$

- Con questa nuova topologia i processi all'interno della griglia possono avere 4 vicini virtuali (e non piu' solo 2, ovvero quello di sinistra e quello di destra).
- E' utile che si possa dichiarare se le direzioni della griglia virtuale devono essere cicliche, ovvero se "uscendo" a destra si rientra a sinstra (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? crea un nuovo communicator, che ha anche una topologia!
- 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` (questo secondo `http://mpi.deino.net/mpi_functions/MPI_Cart_create.html` ma con i miei esperimenti, il sistema va semplicemente in crash)
- 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!

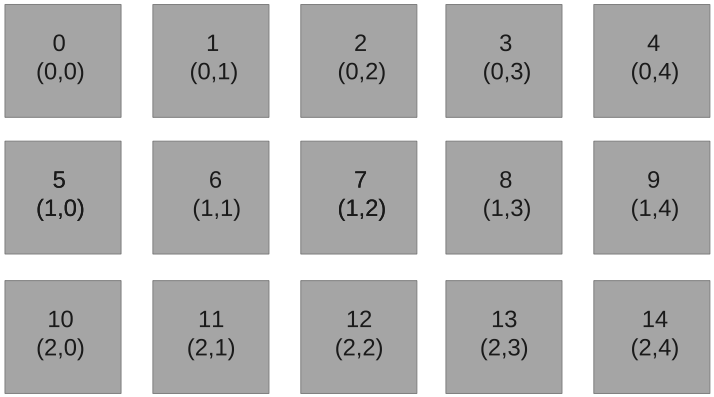
```
MPI_Cart_create(
  MPI_Comm comm_old,//communicator di partenza
  int ndims,        // dimensioni di suddivisione, se voglio 2D metto 2, 3D metto 3...
  int *dims,        // array (p.es. se ndims=2) posso avere array[3][4] ho una griglia 3x4,
  int *periods,     // array logico di dimensione ndims,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

Consiglio: 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();
}
```

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 sapere 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.
- Per inviare (e ricevere) dei dati mi serve il **rank** del sendero receiver (non si usano le cooridnate!) 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" infomazioni.... (ok conosciamo gia' `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)
```

Consecuzio temporum:

1. Prima i dati vengono inviati
2. la stessa area di memoria viene usata per i 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

Supponiamo di avere inizializzato correttamente il sistema.
Struttura del codice (consigliata):

1. inizializziamo il codice, con MPI
2. facciamo in modo che il rank=0 costruisca e mandi le sottomatrici A_{ij} e B_{ij} ai processi P_{ij}
3. e' utile avere una funzione che faccia il *normale* prodotto tra matrici
4. suddividiamo il codice in funzioni
5. serve una funzione che faccia l'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 un array bidimensionale in C 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):

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

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

$$a_{00} \quad a_{01} \quad a_{10} \quad a_{11} \quad a_{02} \quad a_{03} \quad a_{12} \quad a_{13} \quad a_{20} \quad a_{21} \quad a_{30} \quad a_{31} \quad a_{22} \quad a_{23} \quad a_{32} \quad a_{33}$$

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 cosi' 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 `lA` (linea a...)
- la nuova matrice `lA` verra' costruita tramite un indice che dipende dai 4 indici succitati: `ix,iy,i,j`

Per esempio:

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

$$A = \left(\begin{array}{ccc|ccc|ccc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} & a_{0,8} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} & a_{1,8} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} & a_{2,8} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} \\ \hline a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} \\ a_{8,0} & a_{8,1} & a_{8,2} & a_{8,3} & a_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} \end{array}\right) =$$

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

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
- chiama 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], periodicita[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
    periodicita[1] = 1;                             // definisco cicliche sull'asse x e y
    periodicita[0] = 1;                             //                    "

    MPI_Cart_create(MPI_COMM_WORLD, 2, dim, periodicita,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: coinciare 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++)  //
    {
    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

- 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

- l'allineamento iniziale: $2\left(t_s + t_w \frac{n^2}{p}\right)$
- gli $\sqrt{p}$ shift di un posto, successivi richiedono ognuno: $t_s + t_w \frac{n^2}{p}$
-
- le $\sqrt{p}$ moltplicazioni delle sottomatrici (di dimensione $n/\sqrt{p} \times n/\sqrt{p}$) richiedono ognuna $\frac{n^3}{p^{3/2}}$, per un totale di: $\frac{n^3}{p}$

Il tempo parallelo che necessita per moltiplicare 2 matrici $n \times n$, tramite l'algoritmo Cannon, con p processi scala quindi in questo modo, in funzione di n e p:

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

Algoritmo DNS 1

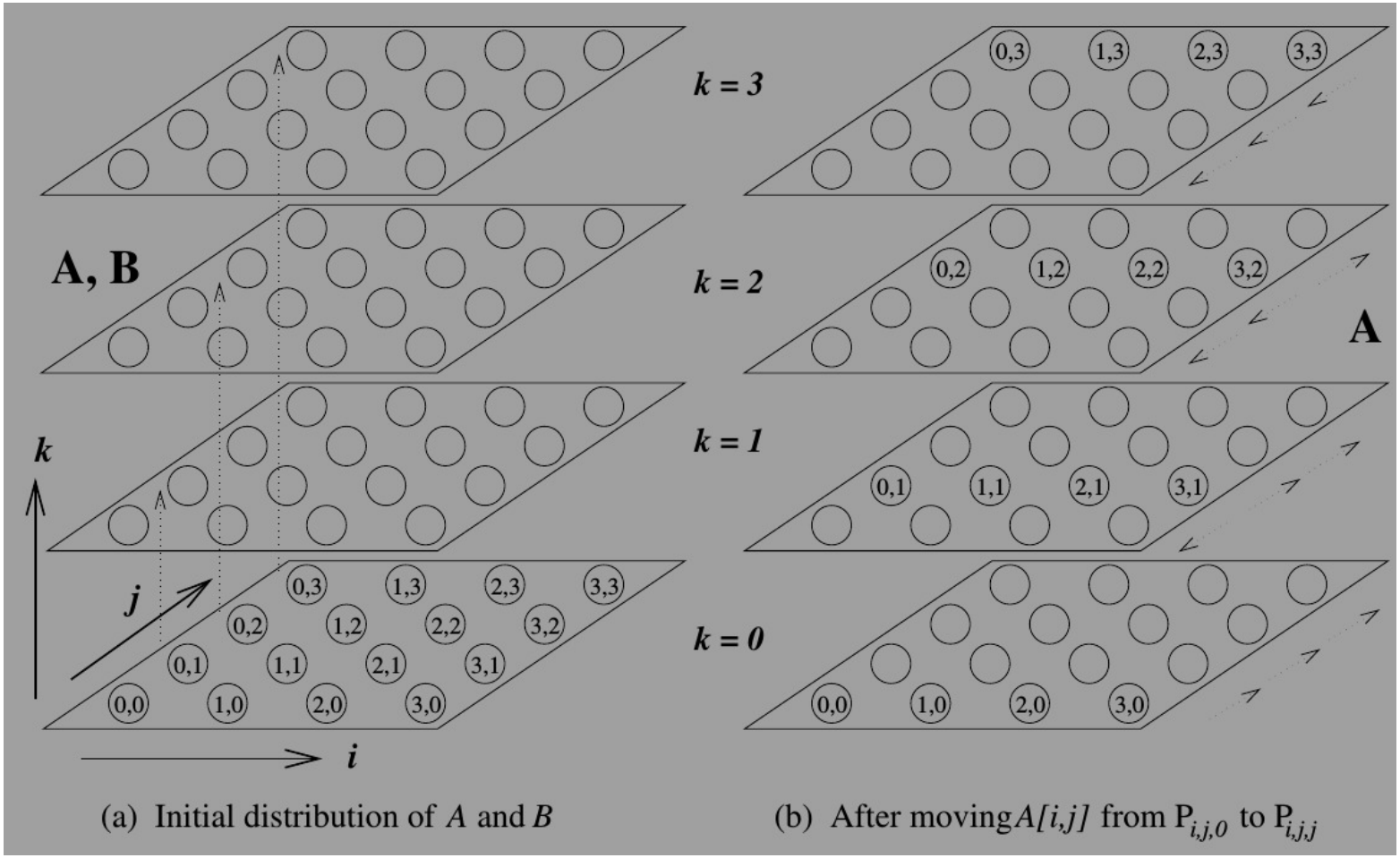
Prodotto di matrici: $C = AB$ dove A, B e C sono matrici $n \times n$:

- autori Dekel, Nassimi and Sahni (DNS...)
- ricordando che l'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 distirbuiti 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

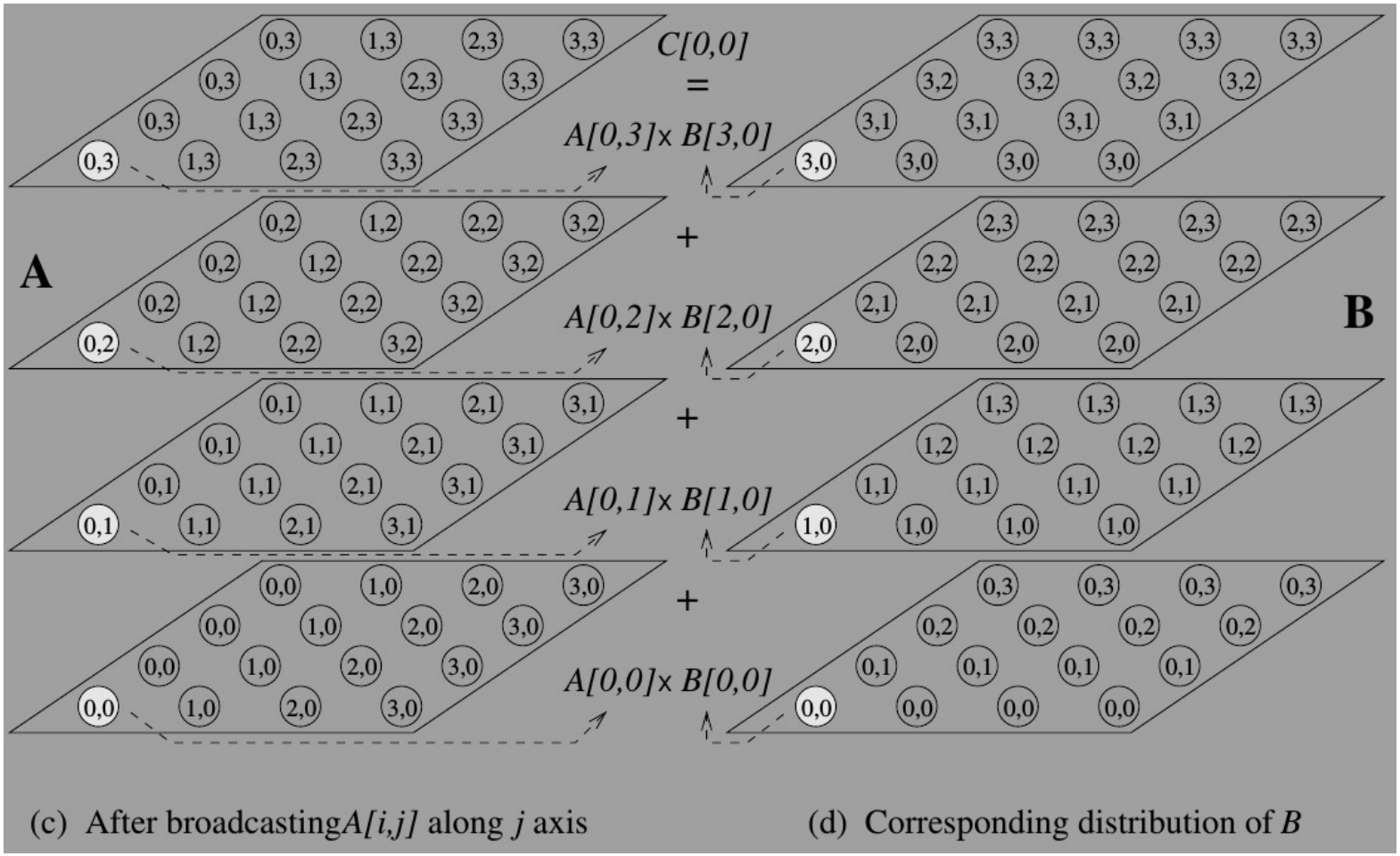


(a) Initial distribution of A and B

(b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

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 `MPI_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 signolo 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 precendente, basta fare attenzione al fatto che le moltiplicazioni (somme) tra numeri diventano moltiplicazioni(somme) tra matrici.

Algoritmo DNS 4

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:

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

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`

Funzioni Non-Blocking:

`https://cvw.cac.cornell.edu/mpip2p` ⇐Spiegazioni chiare sulle nonblocking.

- Per aggiungere maggiore flessibilita' 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 operazioni di comunicazione e serve per *coniugare* l'operazione che comincia la comiunicazione con quella che la termina.
- `MPI_Request` deve essere **completato** prima che 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 come, per esempio:

- `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)
```

consigli utili

Per un non-blocking send:

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

tra il send e il completamento (`MPI_Wait` o `MPI_Test`).
Leggere il buffer, invece, non causa problemi!

Per un non-blocking receive

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

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

MPI_Wait()

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

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

A differenza da `MPI_Wait` questa funzione non si ferma: ritorna subito un `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 e il valore della handle della `request` e' settato su `MPI_REQUEST_NULL` (e la request stessa viene delloccata). Altrimenti se la flag = falsa e 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);
```

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 bloking 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);
   }
```

Fine di MPI

Abbiamo visto:

- cos'e' 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!

Note e suggerimenti per chi e' alle prime armi

- Stai attento, dentro un costrutto MPI non sei in un loop!!! se hai un indice che serve per iterare e metti una cosa tipo:

  ```
  i= i+1
  ```

  occhio che non si aggiorna, in quanto ogni processo becca i nel momento in cui e' stato chiamato il processo stesso. Alla fine del calcolo, per ogni processo `i=1`! (se era inizializzato a 0)
- Problema: ho provato a fare scrivere delle cose dopo `MPI_Finalize` e mi veniva ripetuta tante volte quanti sono i processi!
  Anche se finalizzo, TUTTO lo stesso codice gira su tutti i processi e quindi mi stampa una volta per ogni processo. Quindi quando uno scrive un codice per 10^5 processi devo prestare particolare attenzione a fare scrivere le cose volute a SOLO un processo (che posso chiamare root). In genere meglio non fare nulla dopo un finalize.
- **Attento** se hai una clausola `if` dove c'e' un receive, per esempio, e poi scrivi quello che hai ricevuto ATTENTO, magari nel frattempo, tra la **ricezione** e la **scrittura** altri processi hanno fatto qualcosa e rischi che l'ordine di scrittura sia sbagliato. In generale la scrittura a video con MPI puo' essere insidiosa, nel senso che da luogo spesso a comportamenti non deterministici.
- **Occhio** meglio non chiamare mai una funzione `MPI_blablabla` altrimenti rischi di confonderla con lo standard MPI (anzi nello standard suggeriscono di non farlo!).

Gli errori

- La maggior parte delle funzioni MPI include un parametro di return/errore
- Peccato che, secondo lo standard MPI, il comportamento di default e' di abortire la funzione in caso di errore (Quindi in teoria l'unico messaggio che si ottiene e' da standard e' `MPI_SUCCESS` (zero).
- Lo standard fornisce anche una metodologia per fare un override dell'abort, e quindi il tipo di errori mostrati dipende dall'implementazione.
- Per maggiori dettagli si puo' consultare la sezione *error handling* della *MPI Standard documentation* presso `http://www.mpi-forum.org/docs/`

eseguire in una rete locale

Nel caso si voglia usare il protocollo MPI su computer in una rete domestica e' necessario definire una variabile d'ambiente chiamata `MPI_HOSTS`, che punti all `host_file`, quest'ultimo e' un file in cui c'e' una lista dei processori/computer che possono eseguire il calcolo.
Se tutto e' eseguito correttamente si lancia con:

```
mpirun -np 4 -f host_file ./mpi_hello_world.x
```

quindi, dopo aver specificato il numero dei processi, si indica il file con i nomi dei computer che li faranno girare e l'eseguibile che e' stato compilato.
Attenzione bisgna fare: `export MPI_HOSTS=hosts_file`
Esempio di host_file:

```
>>> cat host_file
cetus1
cetus2
cetus3
cetus4
```

dove cetus1 e' il nome di una macchina etc (e' proprio il nome usato per le chiamate ssh). Se cetus1 e' una macchina dual core e voglio che i processi vengano **spawn**ati sui vari core prima dipassare alle altre macchien devo fare:

```
>>> cat host_file
cetus1:2
cetus2:2
cetus3:2
cetus4:2
```

quindi in questo modo prima gira sui 2 processori del cetus1, poi passa al cetus2 etc...

Dove sono?

provare a scrivere un codice in cui si fa scrivere a video il nome dell'host di ogni processo usando la funzione C `gethostname`

```c
#include <stdio.h>
#include <unistd.h>
#include <limits.h>  // serve per definire PATH_MAX (lunghezza massima)
#include "mpi.h"
int main(int argc, char** argv) {
int me, nproc;
char hostname[PATH_MAX];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
if(gethostname(hostname, PATH_MAX) == 0){
printf("Sono il processo %d di %d  localizzato @%s\n",
me, nproc, hostname);
} else {
printf("Sono un clone perso %d su un totale di %d !\n",
me, nproc);
}
MPI_Finalize();
return 0;
}
```