Sviluppo di software parallelo con il sistema MPI

Sommario

- Cluster Beowulf
- Message Passing
- Installazione di MPI
- Routine di base di MPI
- Calcolo della somma di n numeri
- Gruppi di processi e topologie virtuali
- Gestione di file in parallelo
- Prodotto tra 2 matrici con la strategia BMR
- ScaLapack

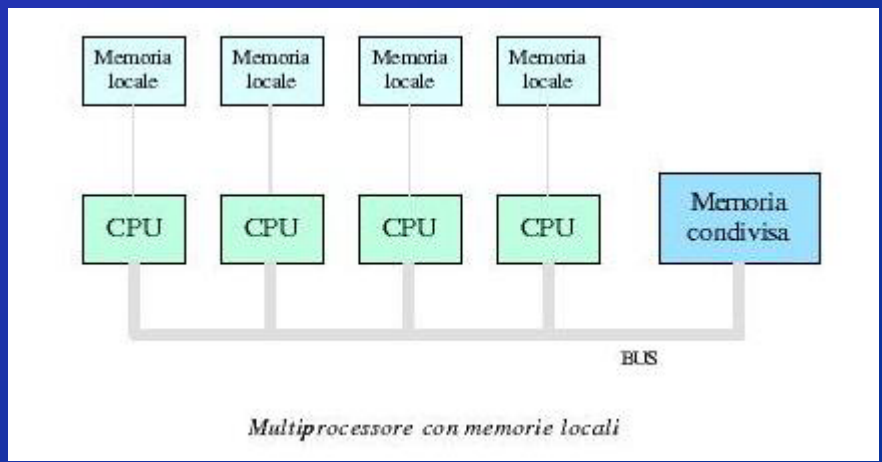
Architetture parallele

CALCOLATORI PARALLELI

- **SISD**
- **SIMD**
- **MIMD**

Multiple Instruction Multiple Data

Istruzioni e dati sono indipendenti su ogni CPU



Multiprocessore con memorie locali

Cluster Beowulf

con hardware facilmente reperibile

Hardware:

1. server, unità di elaborazione che mette a disposizione una o più risorse per altre unità (client) in un cluster
2. client, generica macchina del cluster
3. switch, concentratore

server

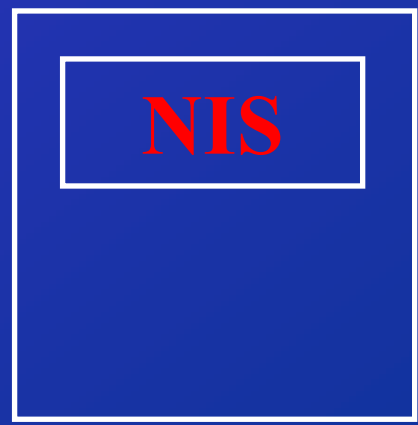
client

switch

Software di base sul Cluster

Network Information Server

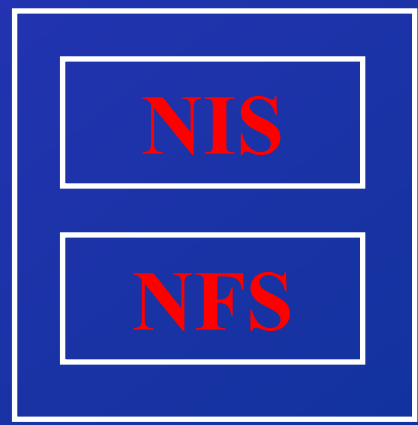
Tutti gli account sono comuni a tutte le macchine.



Software di base sul Cluster

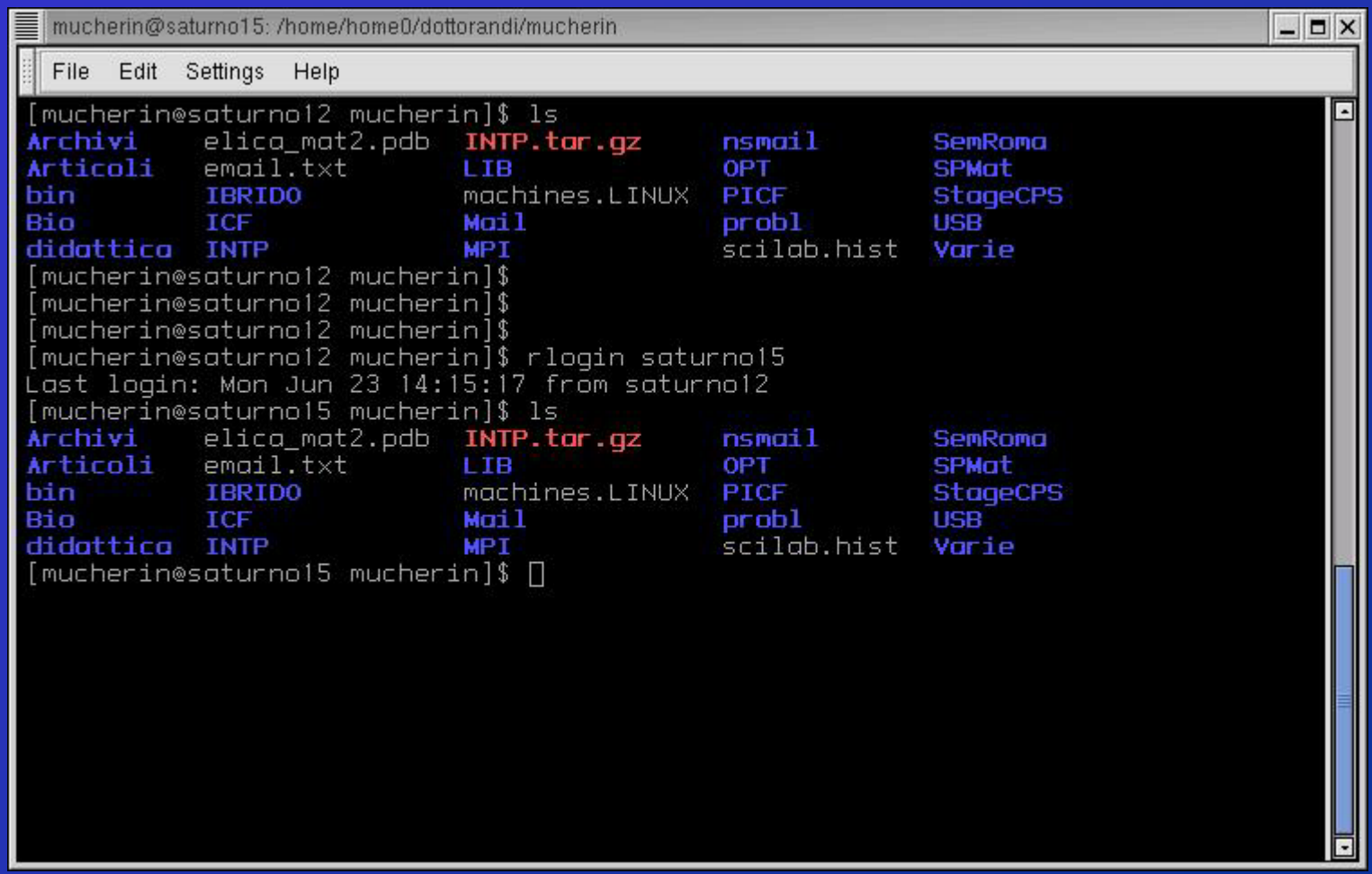
Network File System

Tutte le macchine del Cluster condividono una porzione del filesystem del server.
Di solito `/home` è una partizione di un disco del server



Software di base sul Cluster

Esempio



Message Passing

È un criterio per la progettazione di un algoritmo parallelo in termini di processi concorrenti che coordinano la propria attività attraverso la comunicazione di dati.

I processi evolvono asincronamente: si sincronizzano soltanto durante la fase di comunicazione.

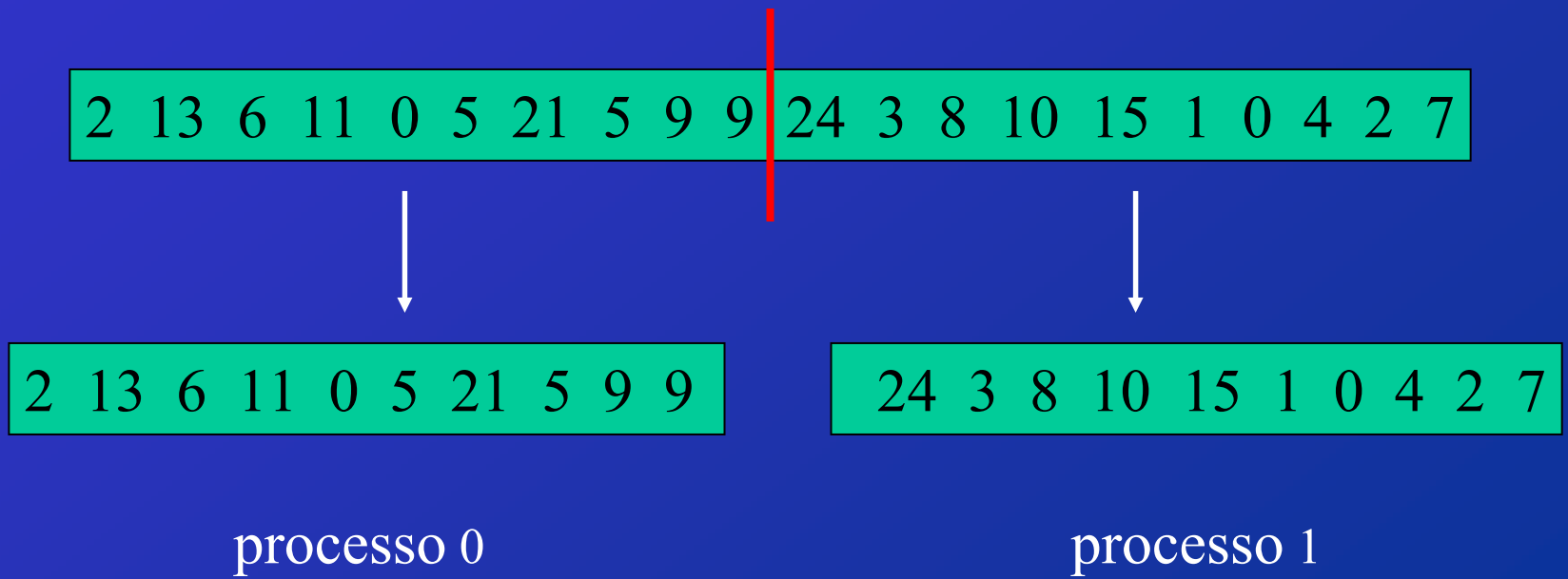
Ciascun processo esegue un proprio algoritmo utilizzando i dati residenti sulla propria memoria. Quando è necessario comunica con gli altri processi.

Progettazione di un algoritmo

secondo il modello Message-Passing

Esempio: somma di n numeri (2 processi)

1. distribuzione dei dati



Progettazione di un algoritmo

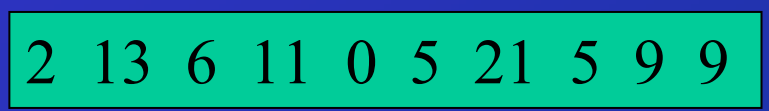
secondo il modello Message-Passing

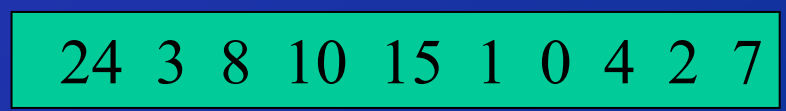
Esempio: somma di n numeri (2 processi)

2. calcolo

Somma parziale = 81

Somma parziale = 74





processo 0

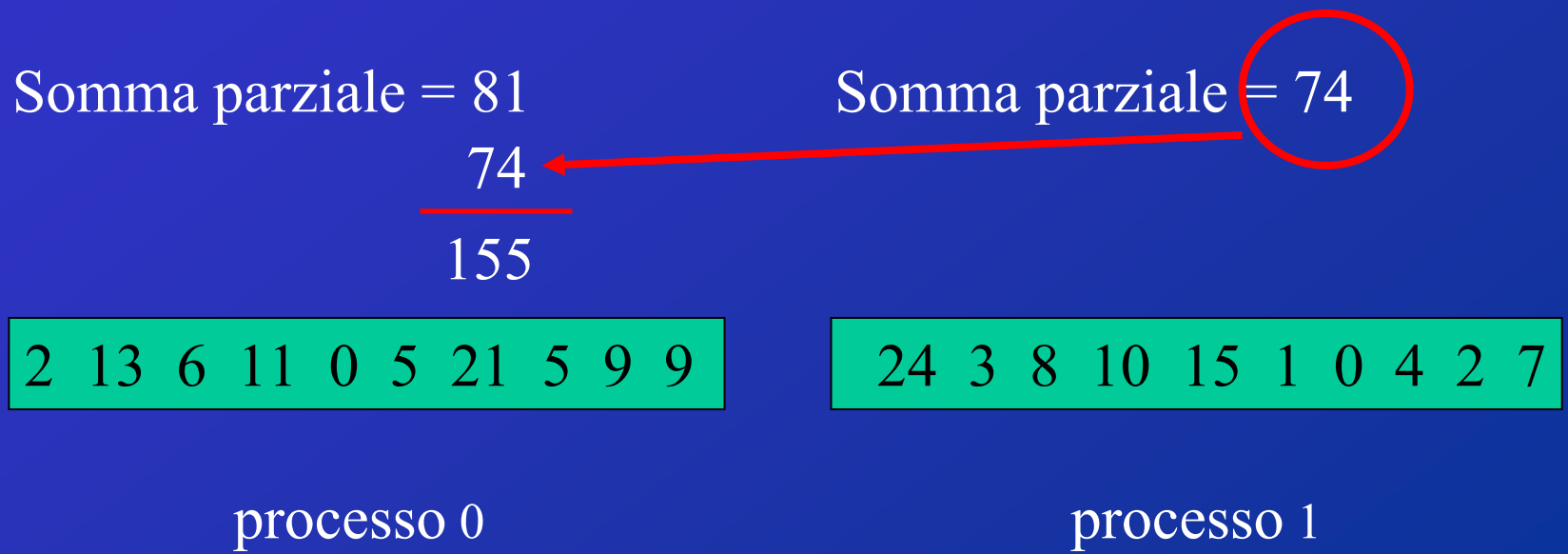
processo 1

Progettazione di un algoritmo

secondo il modello Message-Passing

Esempio: somma di n numeri (2 processi)

3. comunicazione

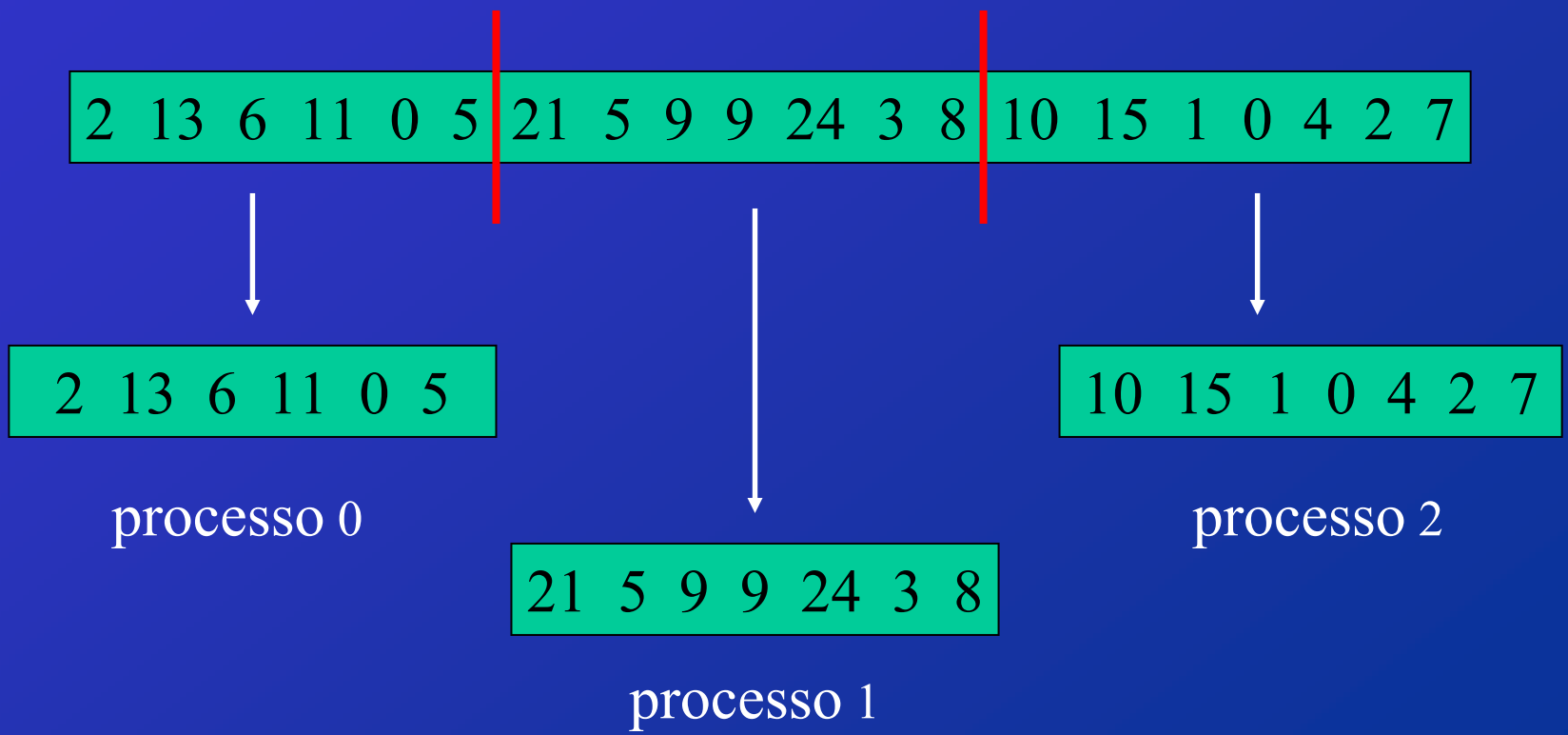


Progettazione di un algoritmo

secondo il modello Message-Passing

Esempio: somma di n numeri (3 processi)

1. distribuzione dei dati



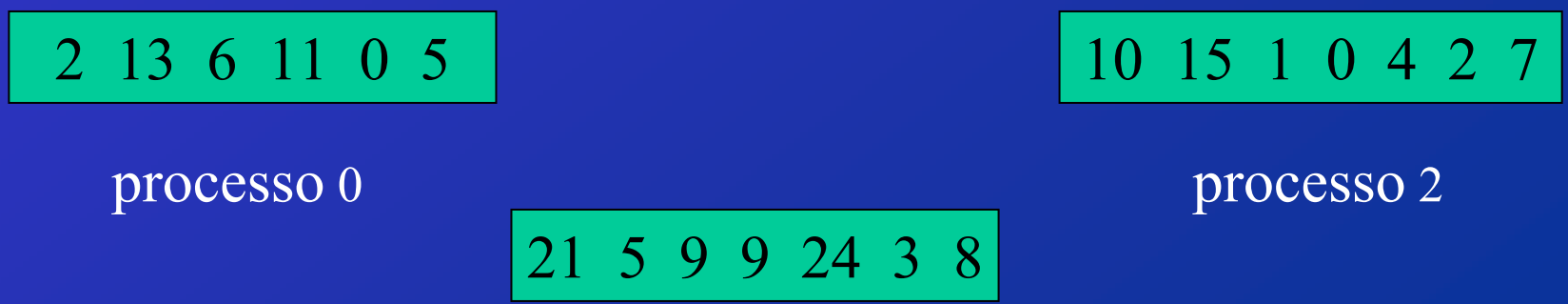
Progettazione di un algoritmo

secondo il modello Message-Passing

Esempio: somma di n numeri (3 processi)

2. calcolo

Somma parz = 37 Somma parz = 79 Somma parz = 39

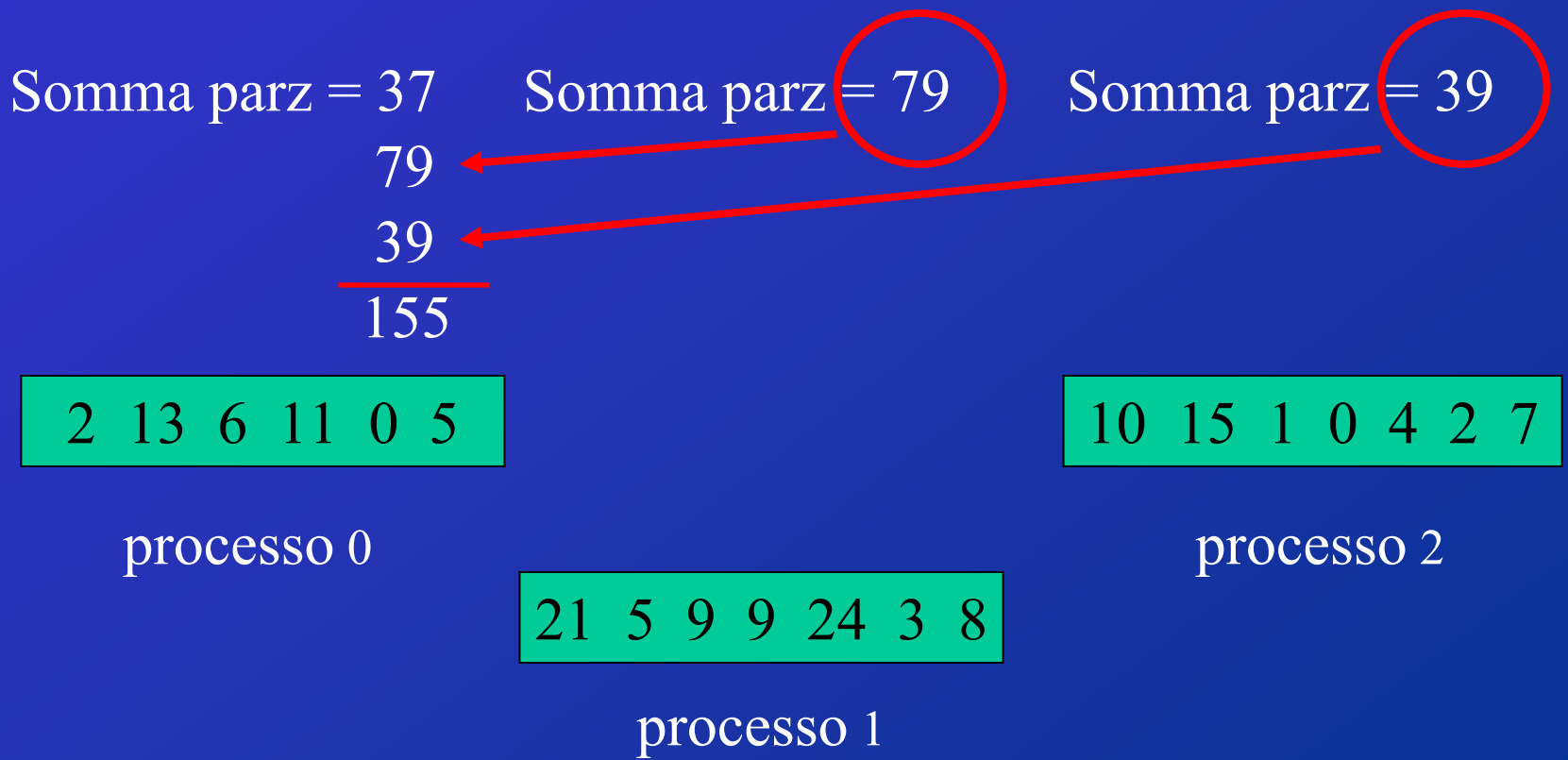


Progettazione di un algoritmo

secondo il modello Message-Passing

Esempio: somma di n numeri (3 processi)

3. comunicazione



MPI

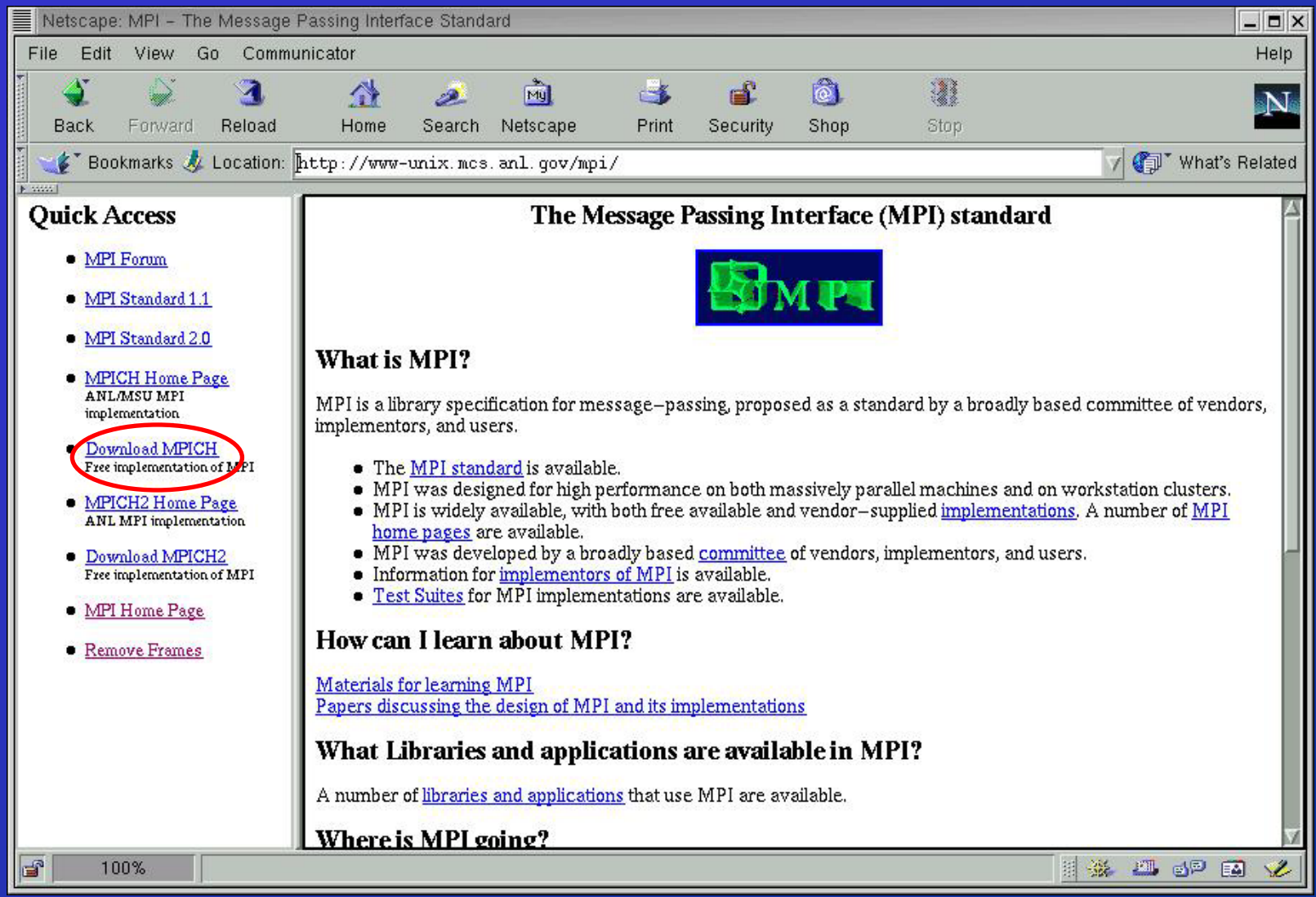
Message Passing Interface

È un ambiente standard per lo sviluppo di applicazioni parallele secondo il modello message passing per reti di calcolatori di tipo Unix.

Mette a disposizione una serie di routine che permettono lo scambio di informazioni tra i processi concorrenti.

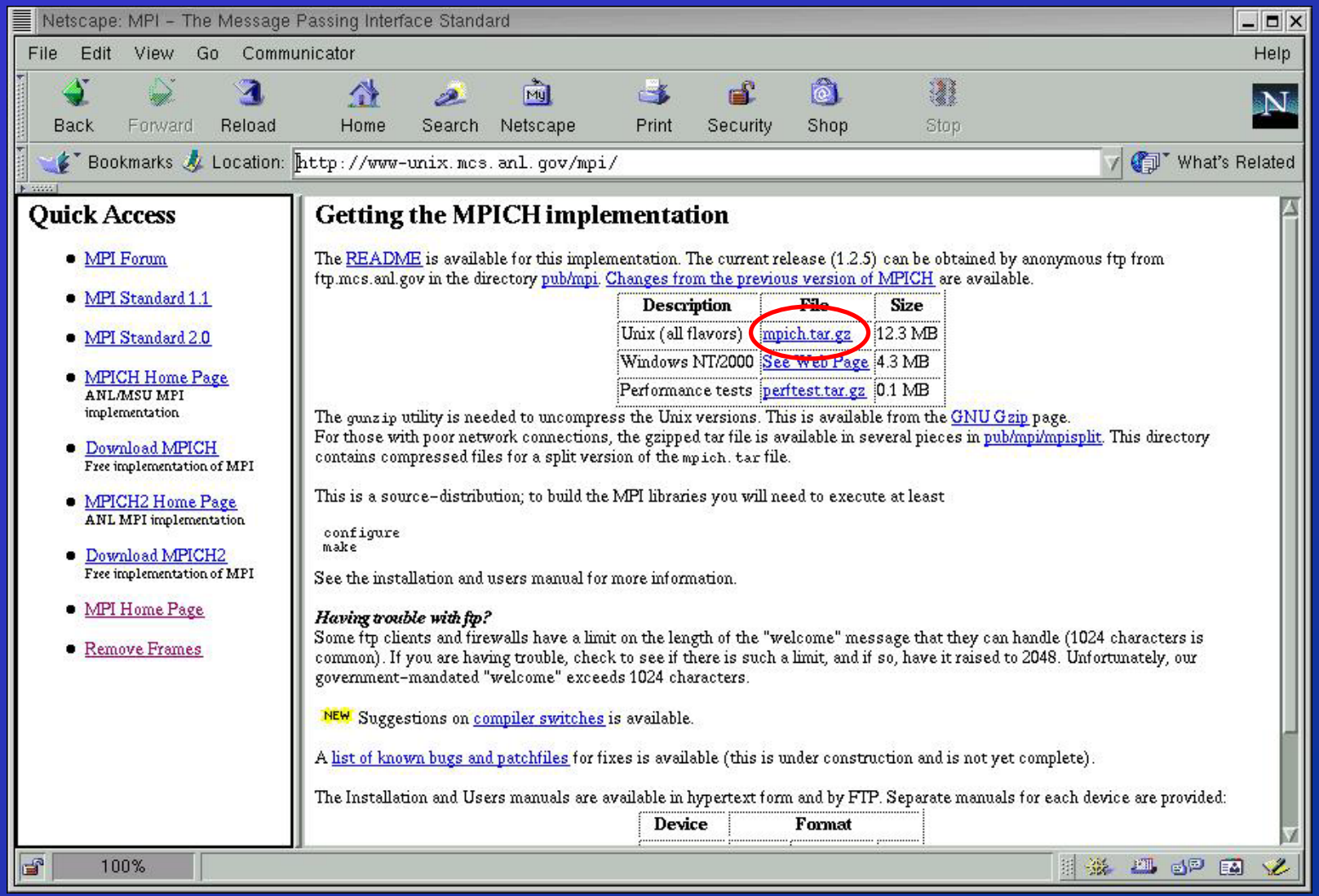
Permette ad una rete di calcolatori di divenire un'unica risorsa di calcolo.

Dove recuperare MPI



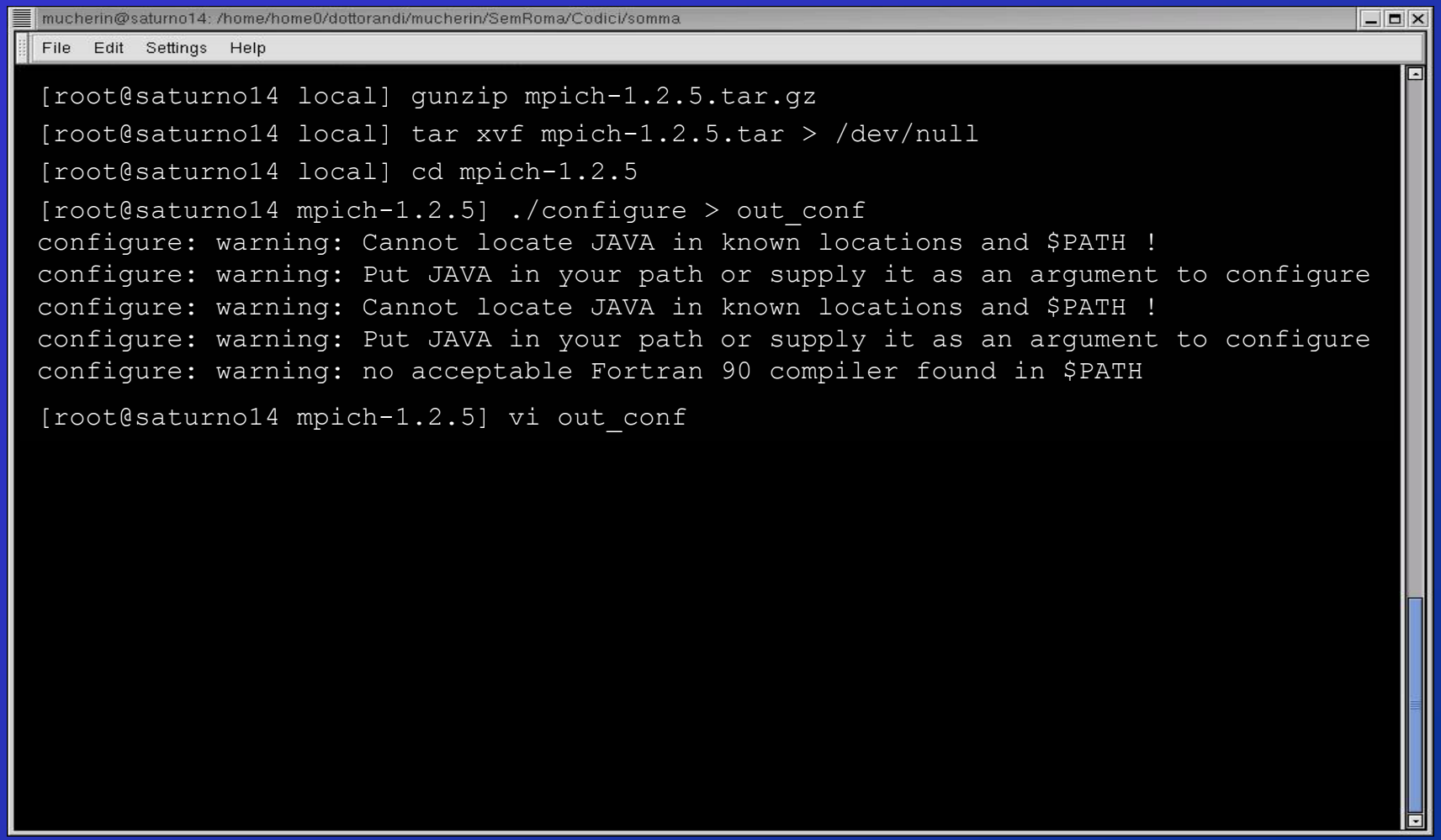
www-unix.mcs.anl.gov/mpi/

Dove recuperare MPI

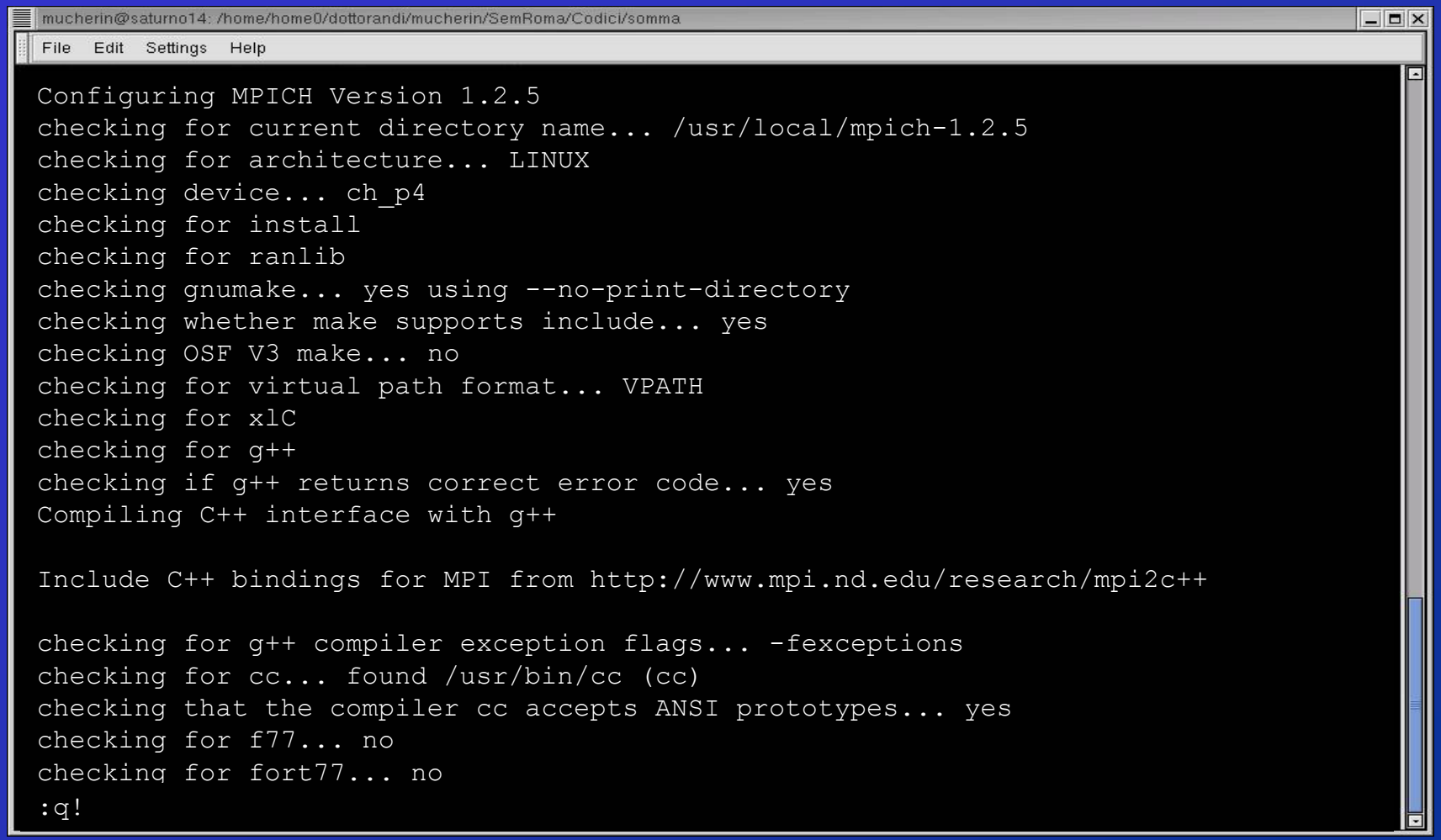


www-unix.mcs.anl.gov/mpi/

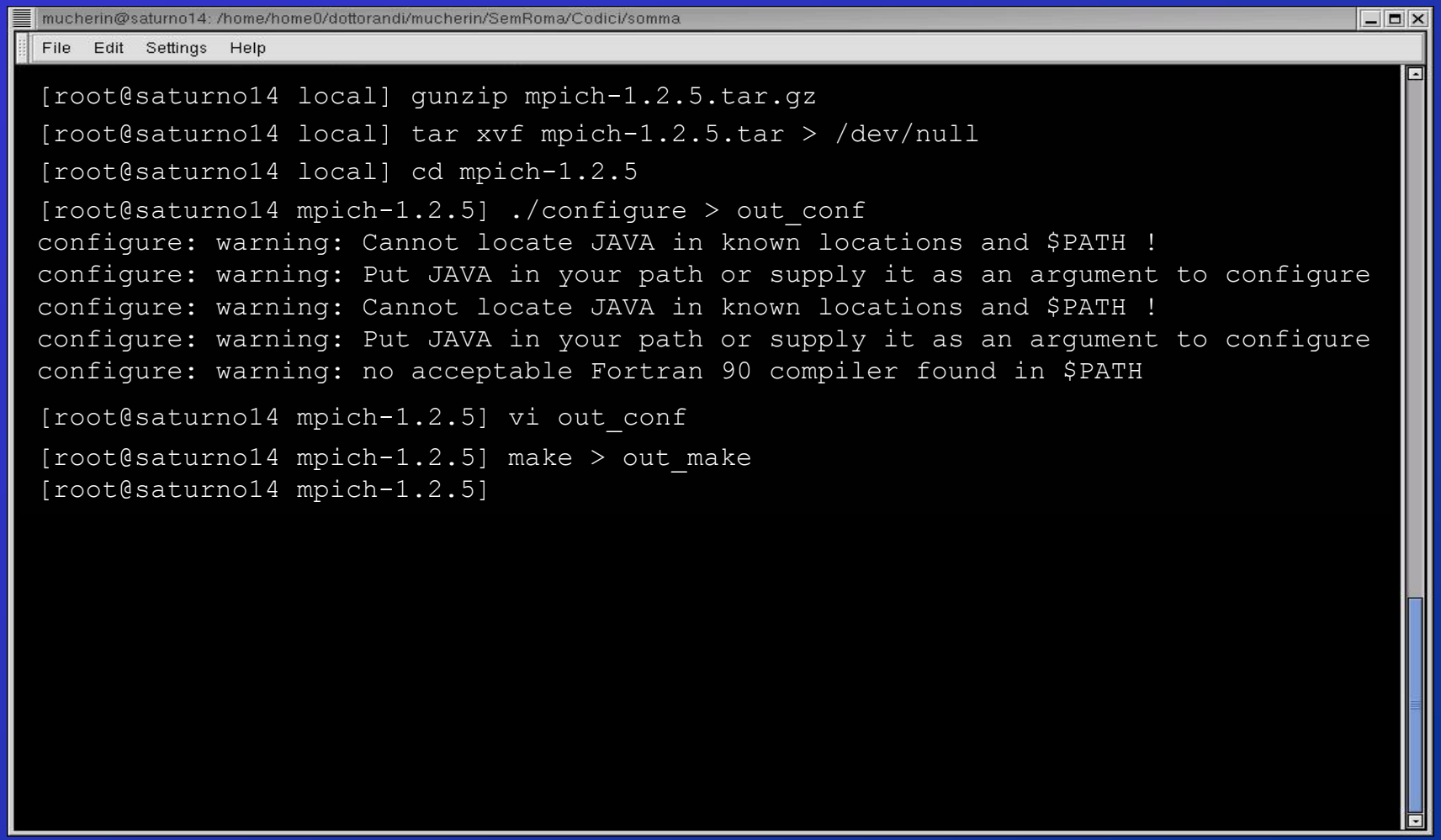
Fasi installazione MPI



Fasi installazione MPI



Fasi installazione MPI



Esplorazione directory MPI

Alcuni programmi per l'uso di MPI

MPI in C e in Fortran77

Con il sistema MPI è possibile sviluppare software parallelo sia in C che in Fortran77.

In C le routine MPI sono scritte sotto forma di function; in fortran77 sotto forma di subroutine.

In C il sistema MPI introduce una serie di tipi di dati; in fortran77 non è possibile definirli.

In seguito verrà utilizzato MPI con il linguaggio di programmazione C.

Un programma in ambiente MPI

Esempio 1: somma di n numeri

```
/* Somma dei primi n numeri interi */
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])
{
  int myid, numprocs, i, n, nloc, nloc0, namelen;
  double *a, sump, sum;  double time1, time2;
  char pname[MPI_MAX_PROCESSOR_NAME]; MPI_Status info;

  MPI_Init(&argc,&argv);  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);  MPI_Get_processor_name(pname,&namelen);

  n = atoi(argv[1]);  nloc = n/numprocs;
  nloc0 = nloc + (n - nloc*numprocs);  if (myid == 0)  nloc = nloc0;
  a = (double*)calloc(nloc,sizeof(double));
  if (myid == 0) {
    for (i=0; i<nloc; i++)  a[i] = i + 1;
  } else {
    for (i=0; i<nloc; i++)  a[i] = nloc0+1 + (myid-1)*nloc + i;
  }
  fprintf(stderr,"Process %d on `%s': I have %d numbers.\n",myid,pname,nloc);
  if (myid == 0) time1 = MPI_Wtime();
  sump = 0.;  for (i=0; i<nloc; i++)  sump = sump + a[i];
  if (myid != 0) {
     MPI_Send(&sump,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD);
  } else {
    sum = sump;
    for (i=1; i<numprocs; i++) {
        MPI_Recv(&sump,1,MPI_DOUBLE,i,i,MPI_COMM_WORLD,&info);  sum = sum + sump;
    }
    time2 = MPI_Wtime();
    fprintf(stderr,"The sum is = %lf\nTime = %f sec\n",sum,time2-time1);
  }
  MPI_Finalize();    return 0;
}
```

Routine di base MPI

```
MPI_Init(int argc, char *argv[]);
```

Routine per l'inizializzazione dell'esecuzione in ambiente MPI. Questa routine non ha nessun parametro di output ed ha per parametri di input i parametri di input della funzione main.

```
MPI_Finalize();
```

Routine per la terminazione dell'esecuzione in ambiente MPI. Questa routine non ha nessun parametro di input e nessun parametro di output.

Routine di base MPI

Osservazione

Tutte le routine MPI (tranne `MPI_Wtime` e `MPI_Wtick`) in C sono function con valore di ritorno un indicatore di errore.

Questo indicatore di errore può per esempio assumere uno di questi valori:

- `MPI_SUCCESS`, nessun errore;
- `MPI_ERR_ARG`, argomento errato;
- `MPI_ERR_RANK`, identificavo errato.

Routine di base MPI

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

Routine che determina il numero `size` di processi associati al comunicatore `comm`.

Parametri di input:

- `comm`, *comunicatore* associato a tutti o ad alcuni processi.

Parametri di output:

- `size`, numero di processi associati al comunicatore.

Esiste un comunicatore predefinito in ambiente MPI, `MPI_COMM_WORLD`, che è associato a tutti i processi concorrenti.

Routine di base MPI

```
MPI_Comm_rank(MPI_Comm comm, int *myid);
```

Routine che determina l'identificativo `myid` del processo appartenente al comunicatore `comm` che esegue la routine.

Parametri di input:

- `comm`, *comunicatore* associato al processo chiamante.

Parametri di output:

- `myid`, identificativo del processo chiamante.

Lo stesso processo può avere identificativi differenti al variare del comunicatore a cui è associato.

Routine di base MPI

```
MPI_Get_processor_name(char *name, int *len);
```

Routine che determina il nome del processore su cui è in esecuzione il processo chiamante.

Parametri di output:

- `name`, nome del processore;
- `len`, numero di caratteri che compongono `name`.

La lunghezza del nome len è limitata: esiste la variabile MPI `MPI_MAX_PROCESSOR_NAME` che contiene questo limite

Un programma in ambiente MPI

Esempio 1: somma di n numeri

```
/* Somma dei primi n numeri interi */
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])
{
  int myid, numprocs, i, n, nloc, nloc0, namelen;
  double *a, sump, sum;  double time1, time2;
  char pname[MPI_MAX_PROCESSOR_NAME];  MPI_Status info;

  MPI_Init(&argc,&argv);  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);  MPI_Get_processor_name(pname,&namelen);

  n = atoi(argv[1]);  nloc = n/numprocs;
  nloc0 = nloc + (n - nloc*numprocs);  if (myid == 0)  nloc = nloc0;
  a = (double*)calloc(nloc,sizeof(double));
  if (myid == 0) {
    for (i=0; i<nloc; i++)  a[i] = i + 1;
  } else {
    for (i=0; i<nloc; i++)  a[i] = nloc0+1 + (myid-1)*nloc + i;
  }
  fprintf(stderr,"Process %d on `%s': I have %d numbers.\n",myid,pname,nloc);
  if (myid == 0) time1 = MPI_Wtime();
  sump = 0.;  for (i=0; i<nloc; i++)  sump = sump + a[i];
  if (myid != 0) {
     MPI_Send(&sump,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD);
  } else {
    sum = sump;
    for (i=1; i<numprocs; i++) {
        MPI_Recv(&sump,1,MPI_DOUBLE,i,i,MPI_COMM_WORLD,&info);  sum = sum + sump;
    }
    time2 = MPI_Wtime();
    fprintf(stderr,"The sum is = %lf\nTime = %f sec\n",sum,time2-time1);
  }
  MPI_Finalize();    return 0;
}
```

Routine per il tempo

```
time = MPI_Wtime();
```

Il valore di ritorno di questa funzione è l'elapsed time del processo chiamante.

```
MPI_Barrier(MPI_Comm comm);
```

Blocca il processo chiamante fino a quando tutti i processi associati al comunicatore `comm` non effettuano la chiamata a questa routine.

esempio

```
MPI_Barrier(MPI_COMM_WORLD);
time1 = MPI_Wtime();
.....
MPI_Barrier(MPI_COMM_WORLD);
time2 = MPI_Wtime();
```

Un programma in ambiente MPI

Esempio 1: somma di n numeri

```
/* Somma dei primi n numeri interi */
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])
{
  int myid, numprocs, i, n, nloc, nloc0, namelen;
  double *a, sump, sum;  double time1, time2;
  char pname[MPI_MAX_PROCESSOR_NAME];  MPI_Status info;

  MPI_Init(&argc,&argv);  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);  MPI_Get_processor_name(pname,&namelen);

  n = atoi(argv[1]);  nloc = n/numprocs;
  nloc0 = nloc + (n - nloc*numprocs);  if (myid == 0)  nloc = nloc0;
  a = (double*)calloc(nloc,sizeof(double));
  if (myid == 0) {
    for (i=0; i<nloc; i++)  a[i] = i + 1;
  } else {
    for (i=0; i<nloc; i++)  a[i] = nloc0+1 + (myid-1)*nloc + i;
  }
  fprintf(stderr,"Process %d on `%s': I have %d numbers.\n",myid,pname,nloc);
  if (myid == 0) time1 = MPI_Wtime();
  sump = 0.;  for (i=0; i<nloc; i++)  sump = sump + a[i];
  if (myid != 0) {
     MPI_Send(&sump,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD);
  } else {
    sum = sump;
    for (i=1; i<numprocs; i++) {
       MPI_Recv(&sump,1,MPI_DOUBLE,i,i,MPI_COMM_WORLD,&info);  sum = sum + sump;
    }
    time2 = MPI_Wtime();
    fprintf(stderr,"The sum is = %lf\nTime = %f sec\n",sum,time2-time1);
  }
  MPI_Finalize();    return 0;
}
```

Routine per la comunicazione

```
MPI_Send(void *buf, int count,
         MPI_Datatype datatype, int dest,
         int tag, MPI_Comm comm);
```

Permette ad un processo di inviare un messaggio ad un altro processo.

Parametri di input:

- `buf,` indirizzo del primo elemento del buffer;
- `count,` numero di elementi del buffer;
- `datatype,` tipo di ogni elemento del buffer;
- `dest,` identificativo del processo destinatario;
- `tag,` message tag;
- `comm,` comunicatore.

Routine per la comunicazione

```
MPI_Send(void *buf, int count,
         MPI_Datatype datatype, int dest,
         int tag, MPI_Comm comm);
```

Permette ad un processo di inviare un messaggio
ad un altro processo.

Esempio

```
MPI_Send(a,5,MPI_FLOAT,2,100,MPI_COMM_WORLD);
```

Con questa chiamata alla routine `MPI_Send` vengono spediti dal
processo chiamante al processo con identificativo `2`
cinque elementi del vettore `a` di tipo `float`.
I processi che comunicano sono associati al comunicatore `comm`.
Il message `tag` è `100`.

Routine per la comunicazione

```
MPI_Recv(void *buf, int count,
         MPI_Datatype type, int src, int tag,
         MPI_Comm comm, MPI_Status info);
```

Permette ad un processo di ricevere un messaggio da un altro processo

Parametri di input:

- `count,` numero di elementi del buffer;
- `type,` tipo di ogni elemento del buffer;
- `src,` identificativo del processo sorgente;
- `tag,` message tag;
- `comm,` comunicatore.

Routine per la comunicazione

```
MPI_Recv(void *buf, int count,
         MPI_Datatype type, int src, int tag,
         MPI_Comm comm, MPI_Status info);
```

Permette ad un processo di ricevere un messaggio da un altro processo

Parametri di output:

- `buf,` indirizzo del primo elemento del buffer;
- `info,` alcune informazioni sulla comunicazione.

Esempio

```
MPI_Recv(a,5,MPI_FLOAT,0,100,MPI_COMM_WORLD,
         info);
```

Alcune costanti di MPI

Datatypes:

```
MPI_CHAR
MPI_BYTE
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_UNSIGNED_CHAR
MPI_LONG_DOUBLE
```

(solo su alcune macchine)

Comunicatori:

```
MPI_COMM_WORLD
MPI_COMM_SELF
```

Un programma in ambiente MPI

Esempio 1: somma di n numeri

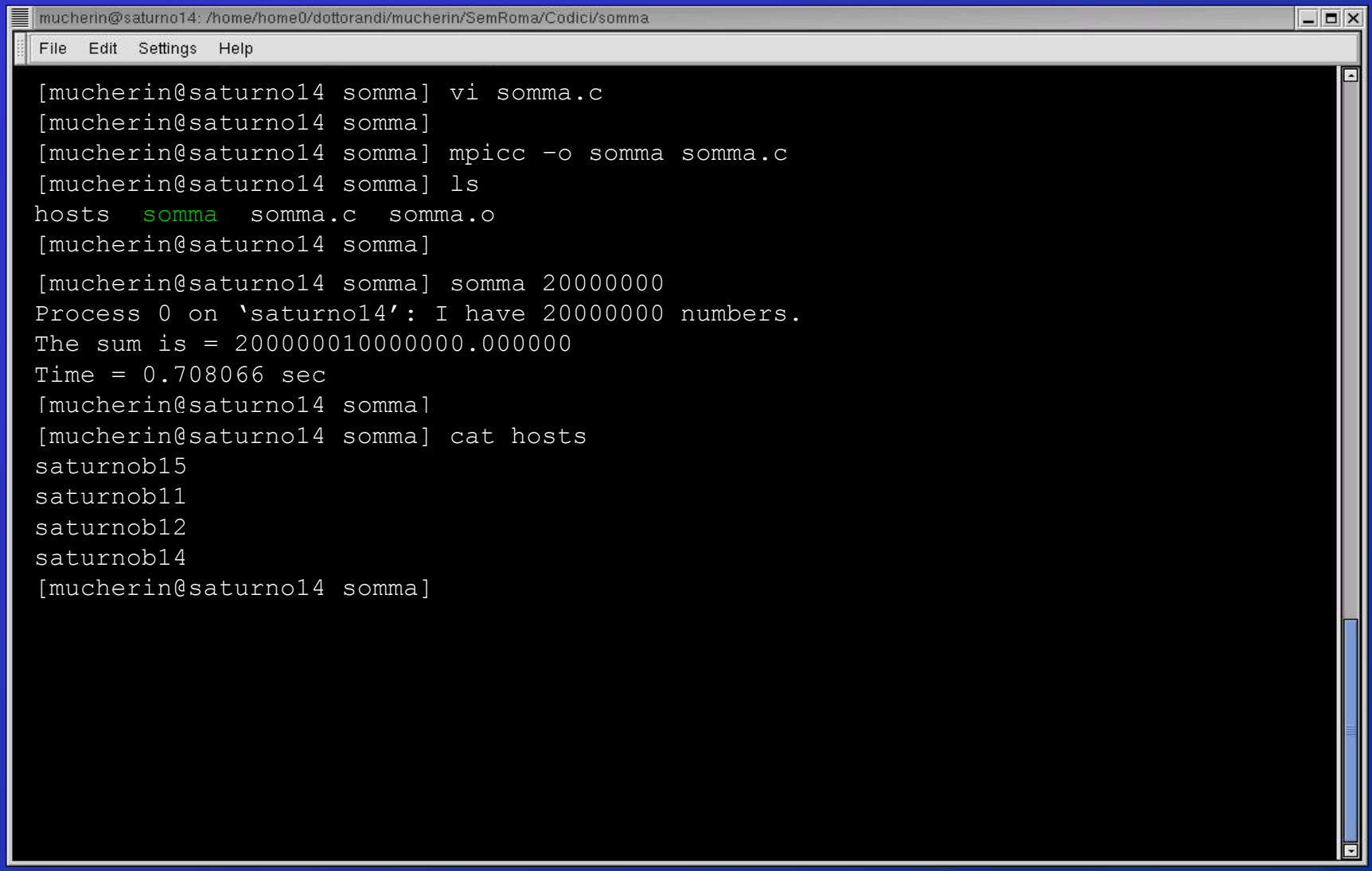
```
/* Somma dei primi n numeri interi */
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])
{
  int myid, numprocs, i, n, nloc, nloc0, namelen;
  double *a, sump, sum;  double time1, time2;
  char pname[MPI_MAX_PROCESSOR_NAME];  MPI_Status info;

  MPI_Init(&argc,&argv);  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);  MPI_Get_processor_name(pname,&namelen);

  n = atoi(argv[1]);  nloc = n/numprocs;
  nloc0 = nloc + (n - nloc*numprocs);  if (myid == 0)  nloc = nloc0;
  a = (double*)calloc(nloc,sizeof(double));
  if (myid == 0) {
    for (i=0; i<nloc; i++)  a[i] = i + 1;
  } else {
    for (i=0; i<nloc; i++)  a[i] = nloc0+1 + (myid-1)*nloc + i;
  }
  fprintf(stderr,"Process %d on `%s': I have %d numbers.\n",myid,pname,nloc);
  if (myid == 0) time1 = MPI_Wtime();
  sump = 0.;  for (i=0; i<nloc; i++)  sump = sump + a[i];
  if (myid != 0) {
     MPI_Send(&sump,1,MPI_DOUBLE,0,myid,MPI_COMM_WORLD);
  } else {
    sum = sump;
    for (i=1; i<numprocs; i++) {
        MPI_Recv(&sump,1,MPI_DOUBLE,i,i,MPI_COMM_WORLD,&info);  sum = sum + sump;
    }
    time2 = MPI_Wtime();
    fprintf(stderr,"The sum is = %lf\nTime = %f sec\n",sum,time2-time1);
  }
  MPI_Finalize();    return 0;
}
```

Un programma in ambiente MPI

Esempio 1: somma di n numeri (compilazione)



Un programma in ambiente MPI

Esempio 1: somma di n numeri (esecuzione)

mucherin@saturno14: /home/home0/dottorandi/mucherin/SemRoma/Codici/somma

File Edit Settings Help

```
[mucherin@saturno14 somma] mpirun -np 1 -machinefile hosts somma 20000000
Process 0 on 'saturno14': I have 20000000 numbers.
The sum is = 200000010000000.000000
Time = 0.708066 sec
[mucherin@saturno14 somma] mpirun -np 2 -machinefile hosts somma 20000000
Process 0 on 'saturno14': I have 10000000 numbers.
Process 1 on 'saturno15': I have 10000000 numbers.
The sum is = 200000010000000.000000
Time = 0.427169 sec
[mucherin@saturno14 somma] mpirun -np 3 -machinefile hosts somma 20000000
Process 0 on 'saturno14': I have 6666668 numbers.
Process 1 on 'saturno15': I have 6666666 numbers.
Process 2 on 'saturno11': I have 6666666 numbers.
The sum is = 200000010000000.000000
Time = 0.283501 sec
[mucherin@saturno14 somma] mpirun -np 4 -machinefile hosts somma 20000000
Process 0 on 'saturno14': I have 5000000 numbers.
Process 1 on 'saturno15': I have 5000000 numbers.
Process 2 on 'saturno11': I have 5000000 numbers.
Process 3 on 'saturno12': I have 5000000 numbers.
The sum is = 200000010000000.000000
Time = 0.211595 sec
[mucherin@saturno14 somma]
```

Altre routine di MPI

Due routine che permettono di effettuare semplici operazioni in parallelo

```
MPI_Reduce(void *sendbuf, void recvbuf, int count,
           MPI_Datatype type, MPI_Op op, int root,
           MPI_Comm comm);

MPI_Allreduce(void *sendbuf, void recvbuf,
              int count, MPI_Datatype type,
              MPI_Op op, MPI_Comm comm);
```

`root`, identificativo del processore che avrà il risultato finale. In `MPI_Allreduce` tutti i processi concorrenti avranno il risultato finale.

Altre routine di MPI

Due routine che permettono di effettuare semplici operazioni in parallelo

```
MPI_Reduce(void *sendbuf, void recvbuf, int count,
           MPI_Datatype type, MPI_Op op, int root,
           MPI_Comm comm);

MPI_Allreduce(void *sendbuf, void recvbuf,
              int count, MPI_Datatype type,
              MPI_Op op, MPI_Comm comm);
```

`op`, operazione da eseguire in parallelo.

```
MPI_MAX    MPI_MIN    MPI_SUM    MPI_PROD
MPI_LAND   MPI_BAND   MPI_LOR    MPI_BOR
MPI_LXOR   MPI_BXOR
```

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR

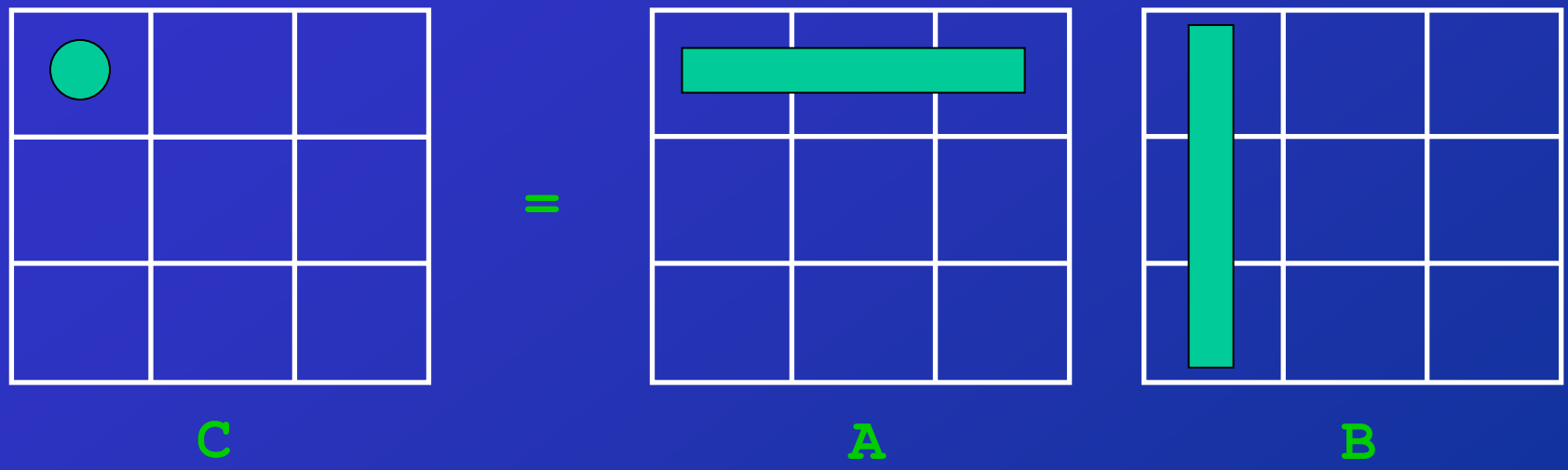
Eseguiamo tutte le fasi per sviluppare un software parallelo con MPI, dalla analisi del modello numerico alla scrittura dell'algoritmo e del software.

Per semplicità supponiamo che:

1. le due matrici da moltiplicare sono quadrate;
2. il numero di processi concorrenti è del tipo n^2;
3. l'ordine delle due matrici è proporzionale al numero di processi

Un programma in ambiente MPI

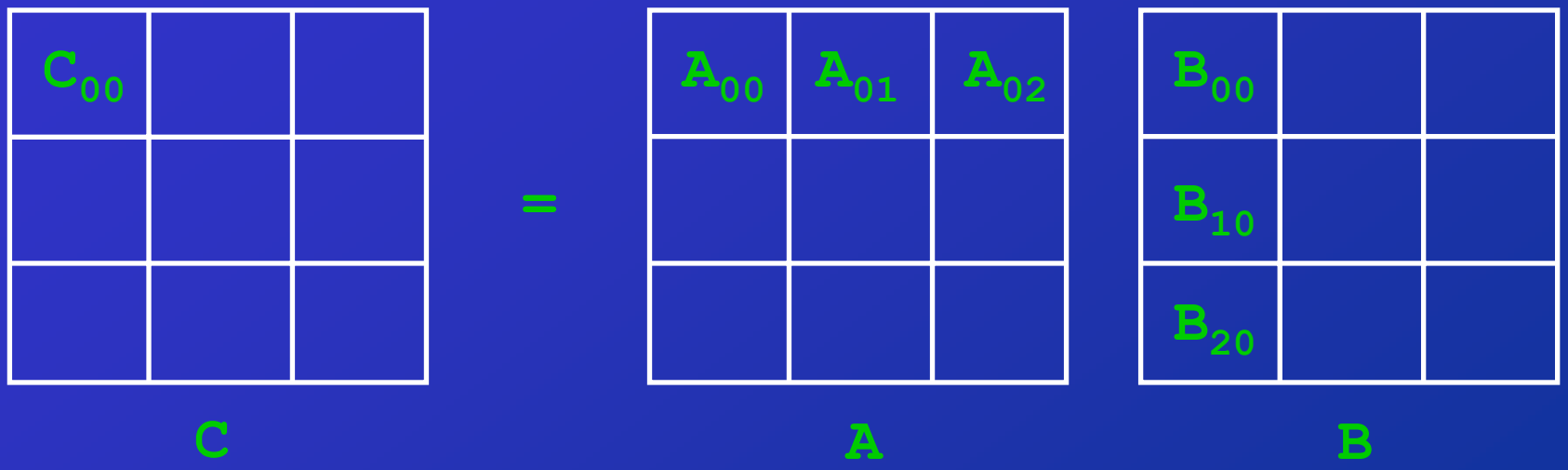
Esempio 2: prodotto tra 2 matrici con strategia BMR



Dividiamo le matrici **A**, **B** e **C** in 9 blocchi disposti su una griglia bidimensionale.
Associamo ogni blocco ad un processo concorrente.

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR

C_{00}	C_{01}	C_{02}
C_{10}	C_{11}	C_{12}
C_{20}	C_{21}	C_{22}

C

=

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

A

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

B

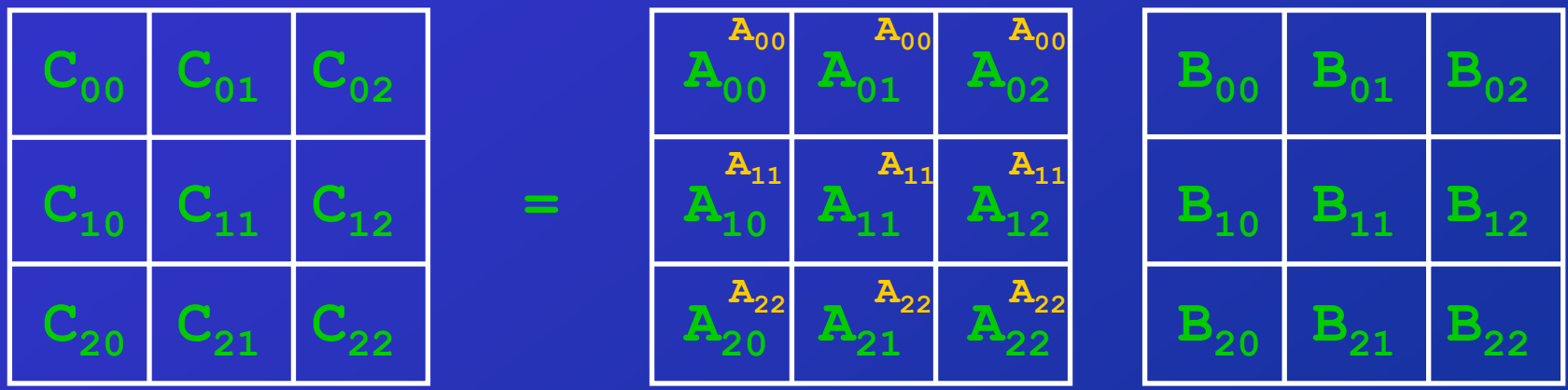
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



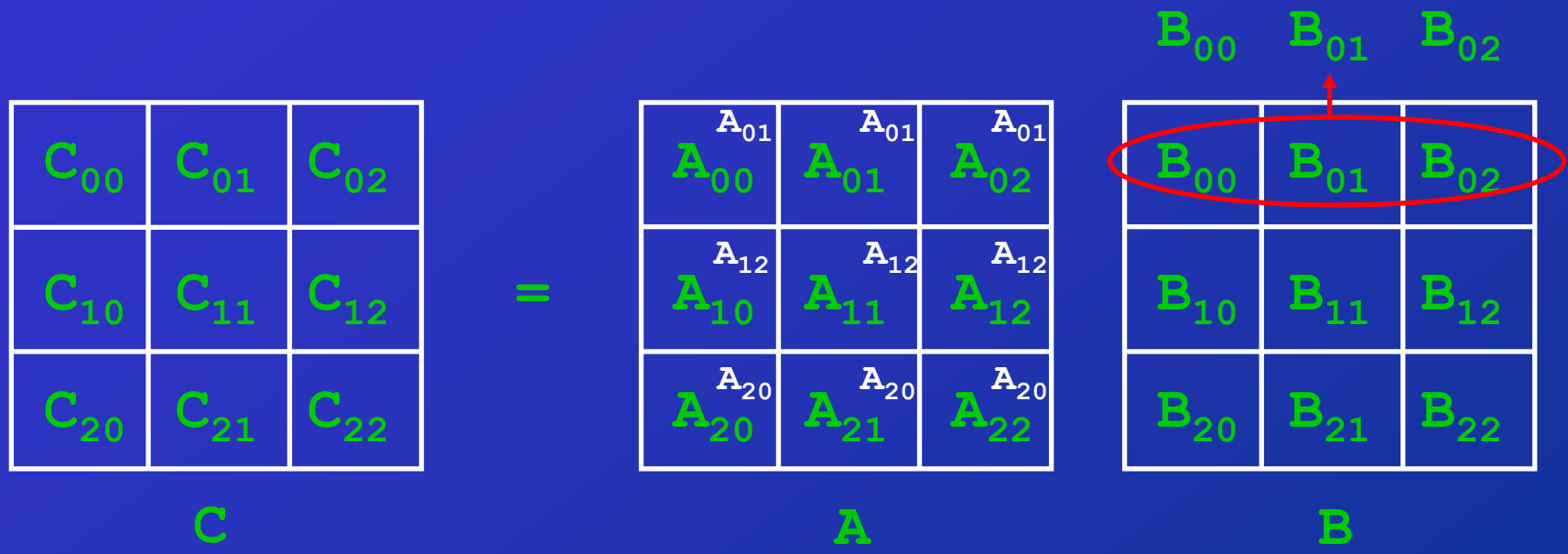
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



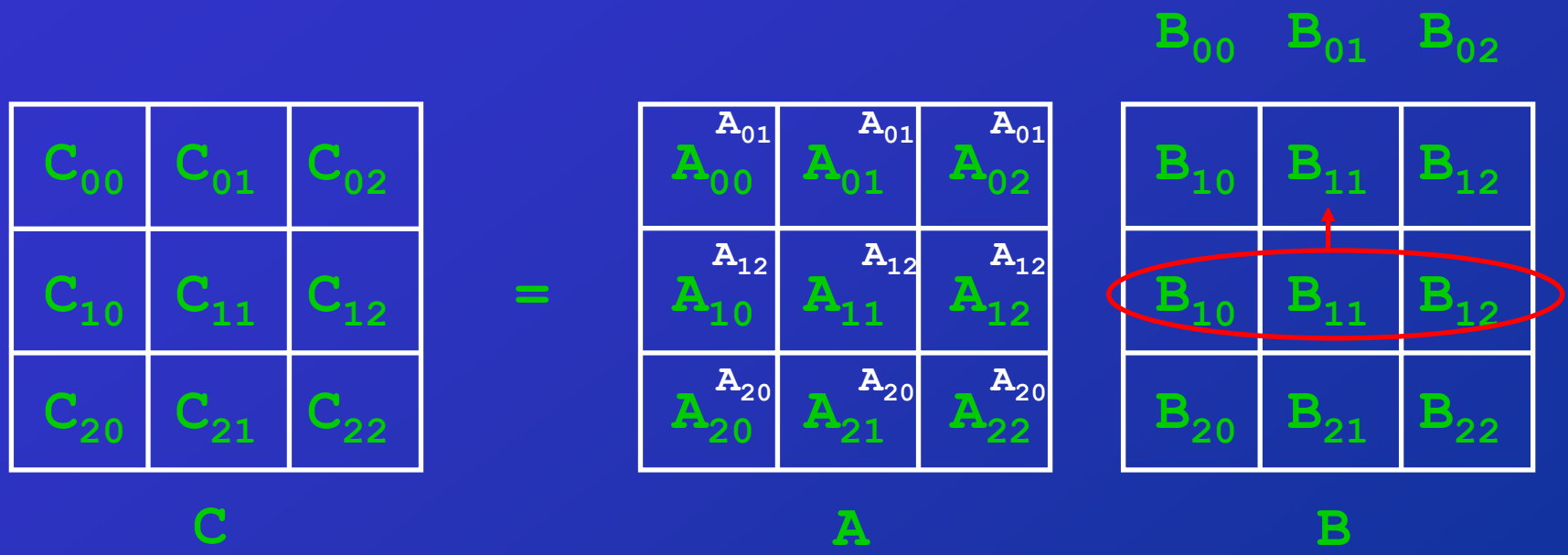
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



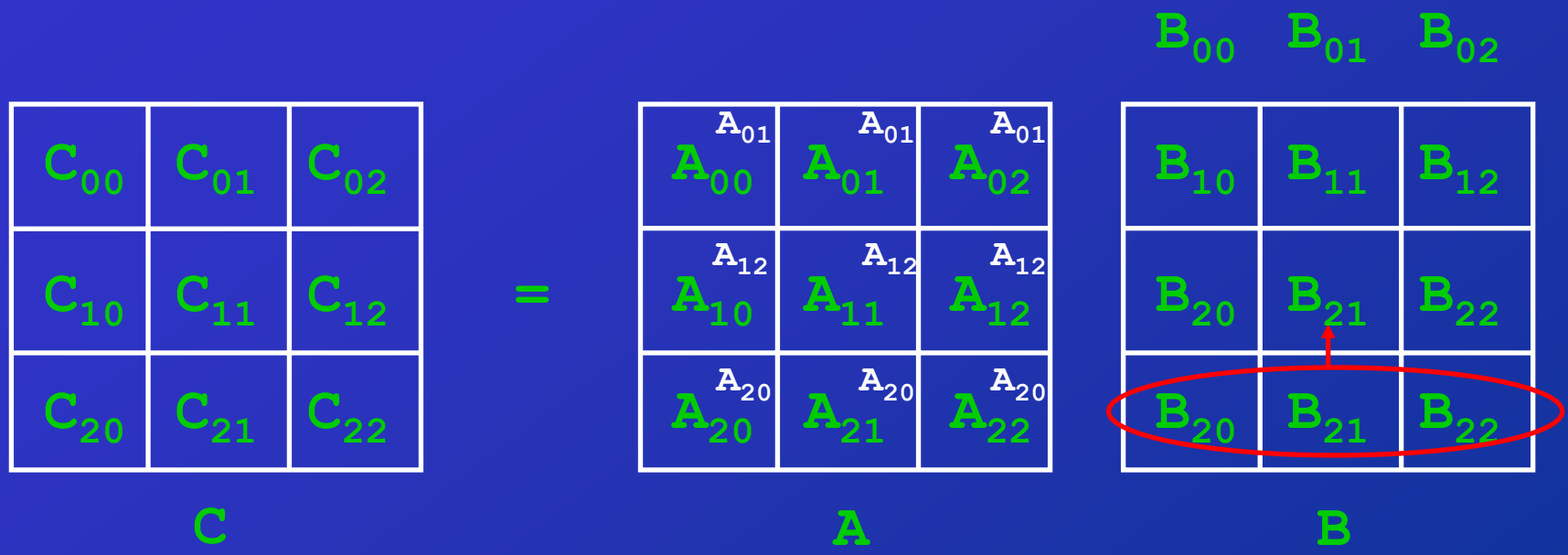
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR

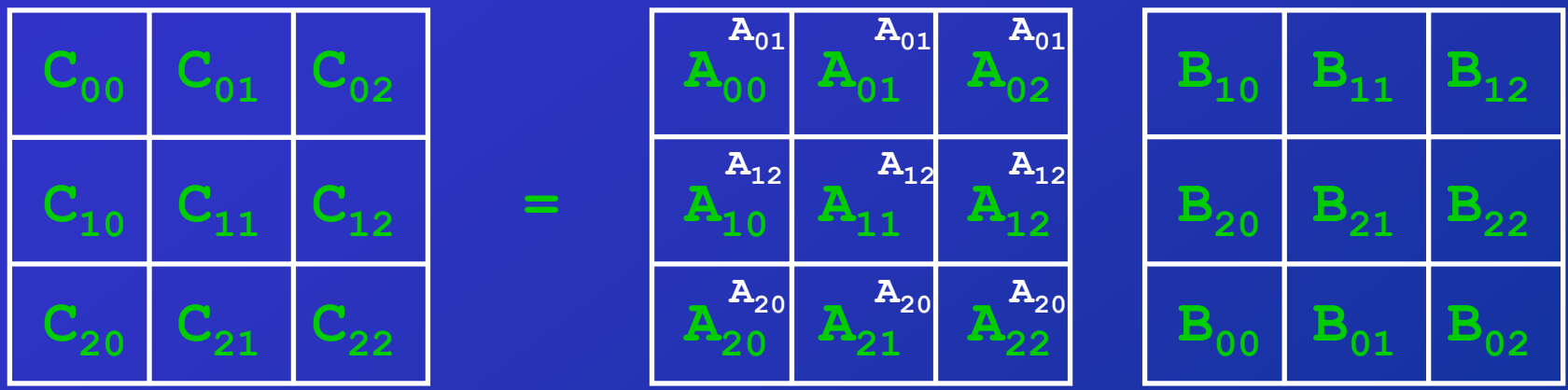
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



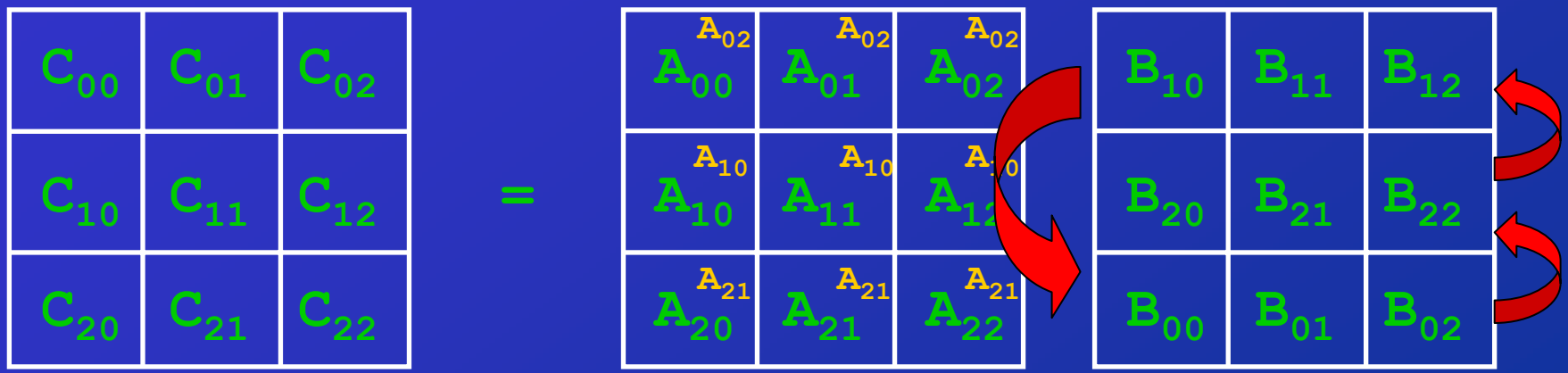
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



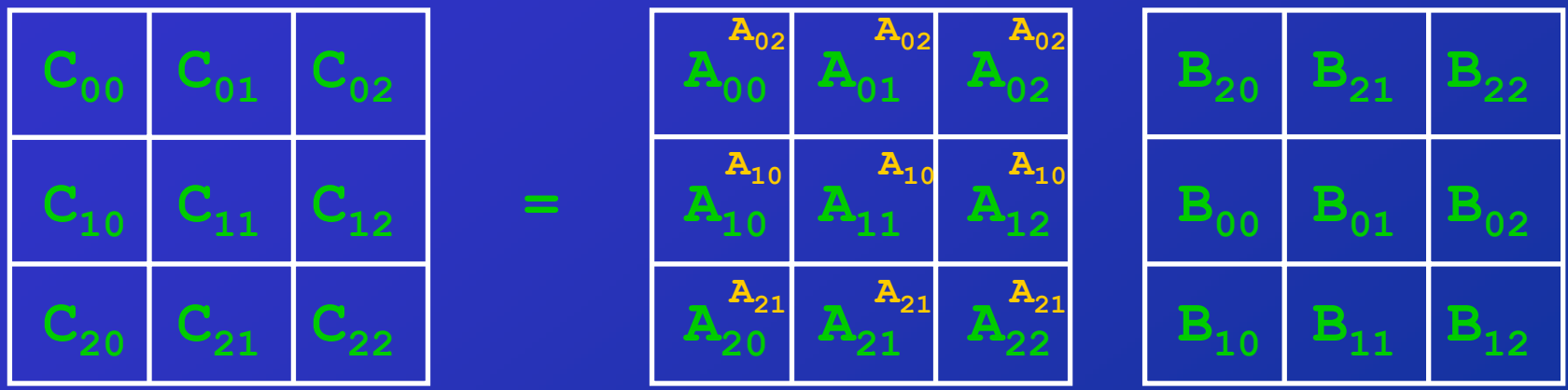
$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: prodotto tra 2 matrici con strategia BMR



$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21}$$
$$C_{02} = A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}$$

$$C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}$$

$$C_{20} = A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20}$$
$$C_{21} = A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}$$

Un programma in ambiente MPI

Esempio 2: strategia BMR (prototipo)

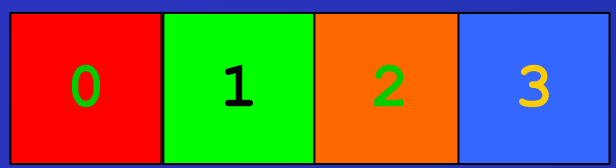
```
void BMR(float *a,float *b,float *mr,int d,int myid,
         int p,MPI_Comm comm,int np,int *coord)
```

`a,`	blocco di A del processo myid
`b,`	blocco di B del processo myid
`mr,`	blocco della matrice risultante del processo myid
`d,`	ordine dei blocchi quadrati A e B
`myid,`	identificativo del processo chiamante
`p,`	radice quadrata del numero di processi coinvolti
`comm,`	comunicatore dei processi concorrenti
`np,`	numero di processi concorrenti
`coord,`	coordinate del processo sulla griglia

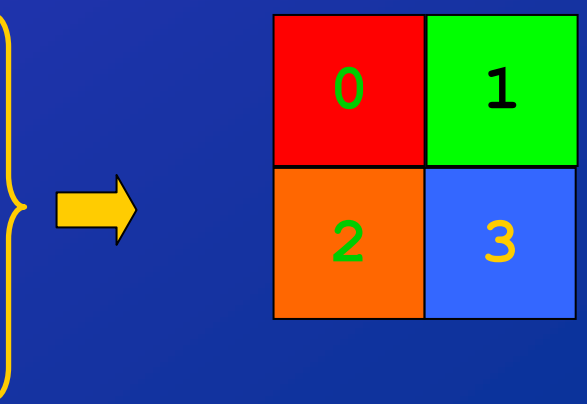
Topologie di processi

```
MPI_Cart_create(
    MPI_Comm comm_old, int ndims, int *dims,
    int *periods, int reorder,
    MPI_Comm *comm_cart);
```

Permette di creare il comunicatore `comm_cart` a cui sono associati gli stessi processi di `comm_old` disposti secondo uno schema topologico.



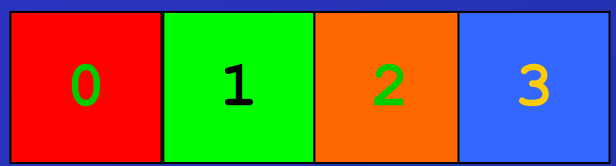

```
ndims = 2;
dims[0] = 2; dims[1] = 2;
periods[0] = 1; periods[1] = 1;
```



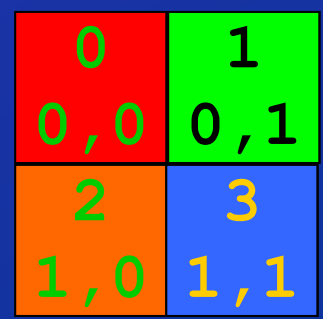
Topologie di processi

```
MPI_Cart_coords(MPI_Comm comm, int myid,
                int maxdims, int *coords);
```

Determina le coordinate del processo chiamante associato al comunicatore `comm`.




```
ndims = 2;
dims[0] = 2; dims[1] = 2;
periods[0] = 1; periods[1] = 1;
```



Gruppi di processi

```
MPI_Comm_group(MPI_Comm comm,
               MPI_group *group);
```

Permette di creare il gruppo di processi `group` relativo ai processi associati al comunicatore `comm`.

```
MPI_Group_incl(
   MPI_Group group, int n, int *ranks,
   MPI_Group *group_out);
```

Permette di creare un sottogruppo `group_out` di `n` processi contenuti nel gruppo `group`. In `ranks` sono memorizzati, ordinatamente, gli identificativi dei processi da inserire in `group_out`.

Gruppi di processi

```
MPI_Comm_create(
        MPI_Comm comm, MPI_Group group,
        MPI_Comm *comm_out);
```

Crea il comunicatore `comm_out` relativo al gruppo di processi `group`. `comm` è il comunicatore associato a tutti i processi coinvolti.

```
MPI_Group_free(MPI_Group group);
```

Libera la memoria relativa al gruppo di processi `group`.

Comunicazione collettiva

```
MPI_Bcast(void *buf, int count,
          MPI_Datatype type, int root,
          int tag, MPI_Comm comm);
```

Permette al processo con identificativo *root* di inviare un messaggio a tutti gli altri processi associati allo stesso comunicatore di root e permette a questi processi di ricevere il messaggio.

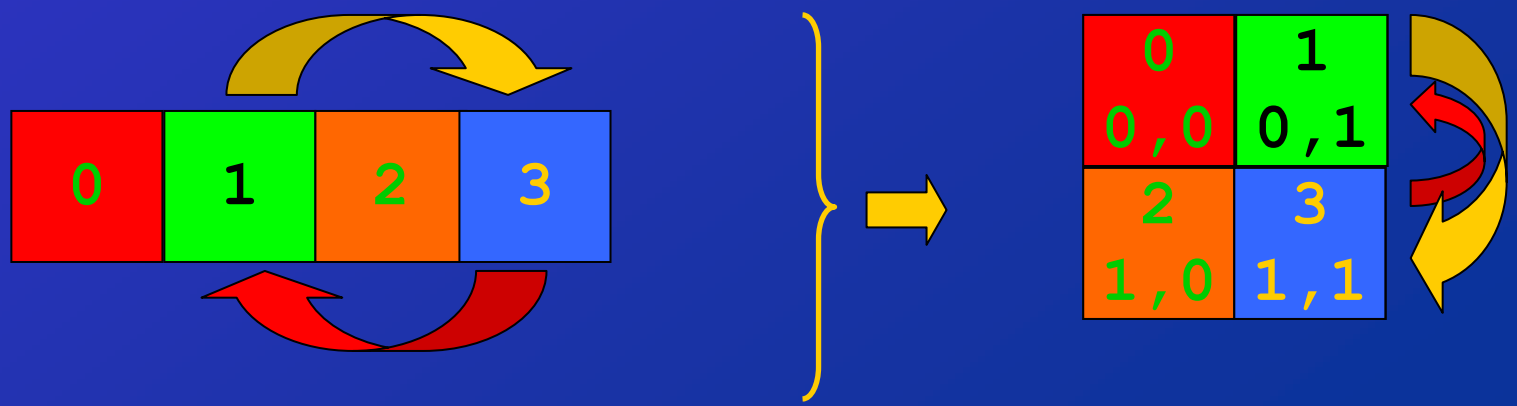
Parametri di input/output:

- `buf,` indirizzo del primo elemento del buffer;
- `count,` numero di elementi del buffer;
- `type,` tipo di ogni elemento del buffer;
- `root,` identificativo del processo root;
- `comm,` comunicatore.

Topologie di processi

```
MPI_Cart_shift(MPI_Comm comm, int direction,
               int ver, int *source
               int *dest);
```

Determina gli identificativi dei processi coi quali il processo chiamante deve ricevere (`source`) e spedire (`dest`) informazioni per effettuare una comunicazione *ad anello* lungo una dimensione di una topologia di processi.



Problema

Come possiamo visualizzare il risultato ottenuto ?

La matrice risultante è distribuita tra i processi concorrenti secondo lo schema descritto.

Soluzione 1

Si attivano una serie di comunicazioni in modo tale che almeno un processo abbia in memoria tutta la matrice risultante, che può essere visualizzata con le routine standard di I/O.

Svantaggio

Almeno un processo è costretto ad allocare memoria sufficiente a memorizzare l'intera matrice risultante.

Problema

Come possiamo visualizzare il risultato ottenuto ?

La matrice risultante è distribuita tra i processi concorrenti secondo lo schema descritto.

Soluzione 2

Si crea un file su un disco comune a tutti i processi e ogni processo vi inserisce i propri dati riguardanti la matrice risultante.

Vantaggio

C'è un notevole risparmio di memoria rispetto alla soluzione 1.

Routine per gestire file

```
MPI_File_open(
      MPI_Comm comm, char *fname, int amode,
      MPI_Info info, MPI_File *fh);
```

Permette l'apertura del file **fname** in parallelo da parte di tutti i processi associati al comunicatore **comm**. La routine da in output la variabile **fh**, che è un *puntatore* al file appena aperto. Tutte le routine per la gestione di file di MPI faranno riferimento al file aperto attraverso la variabile **fh**.

amode modalità di apertura del file

es: **MPI_MODE_CREATE**; **MPI_MODE_RDONLY**; **MPI_MODE_RDWR**; **MPI_MODE_WRONLY**.

Routine per gestire file

Lettura e scrittura di dati su file.

```
MPI_File_read(
      MPI_File *fh, void *buf, int count,
      MPI_Datatype type, MPI_Status *info);

MPI_File_write(
      MPI_File *fh, void *buf, int count,
      MPI_Datatype type, MPI_Status *info);
```

I puntatori al file sono individuali per ogni processo. Questo implica che, se viene modificato ad esempio il puntatore al file del processo 0, il puntatore al file del processo 1 resta inalterato.

Routine per gestire file

Lettura e scrittura *ordinata* di dati su file.

```
MPI_File_read_ordered(
      MPI_File *fh, void *buf, int count,
      MPI_Datatype type, MPI_Status *info);

MPI_File_write_ordered(
      MPI_File *fh, void *buf, int count,
      MPI_Datatype type, MPI_Status *info);
```

Gli accessi al file avvengono in modo ordinato: prima accede il processo 0, poi il processo 1,…, infine il processo con identificativo più alto.

Il puntatore al file è comune ad ogni processo.

Routine per gestire file

Routine per determinare la dimensione di un file.

```
MPI_File_get_size(MPI_File *fh);
```

Routine per la chiusura di un file.

```
MPI_File_close(MPI_File *fh);
```

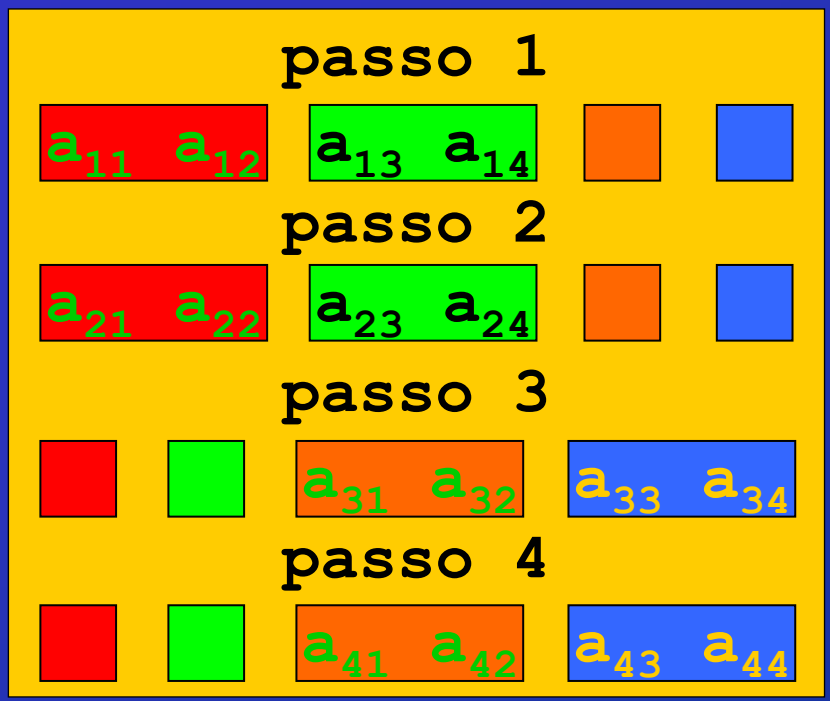
Routine per rimuovere un file.

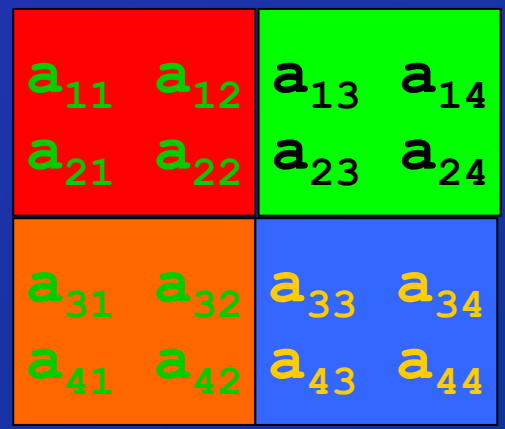
```
MPI_File_delete(char *fname, MPI_Info info);
```

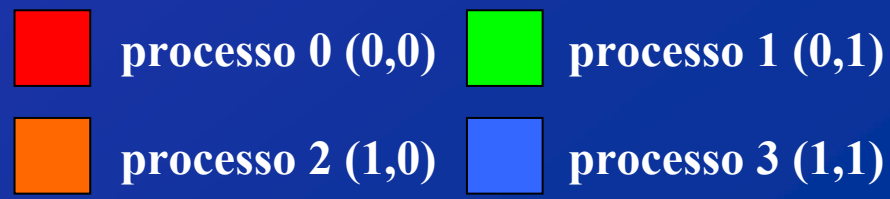
Scrittura di una matrice su file

Supponiamo che una matrice sia distribuita tra 4 processi disposti su una griglia come prevede la strategia BMR

Supponiamo di utilizzare la routine di scrittura su file di MPI che permette di scrivere *ordinatamente*







ScaLapack

A Scalable Linear Algebra Package

È una libreria di software di algebra lineare per calcolatori MIMD a memoria distribuita

La strategia di comunicazione che adotta è quella del message-passing

Contiene routine per:

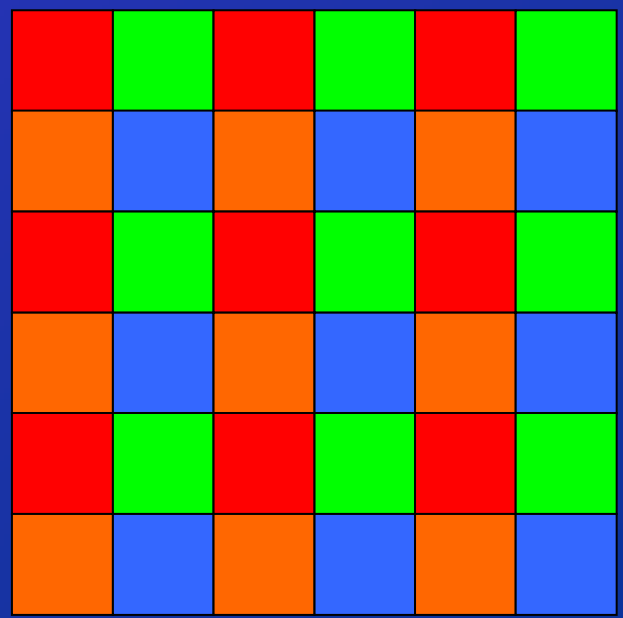
- il calcolo di prodotti righe per colonne tra matrici;
- la fattorizzazione LU e di Cholesky;
- l'inversione di matrici;
- …

ScaLapack

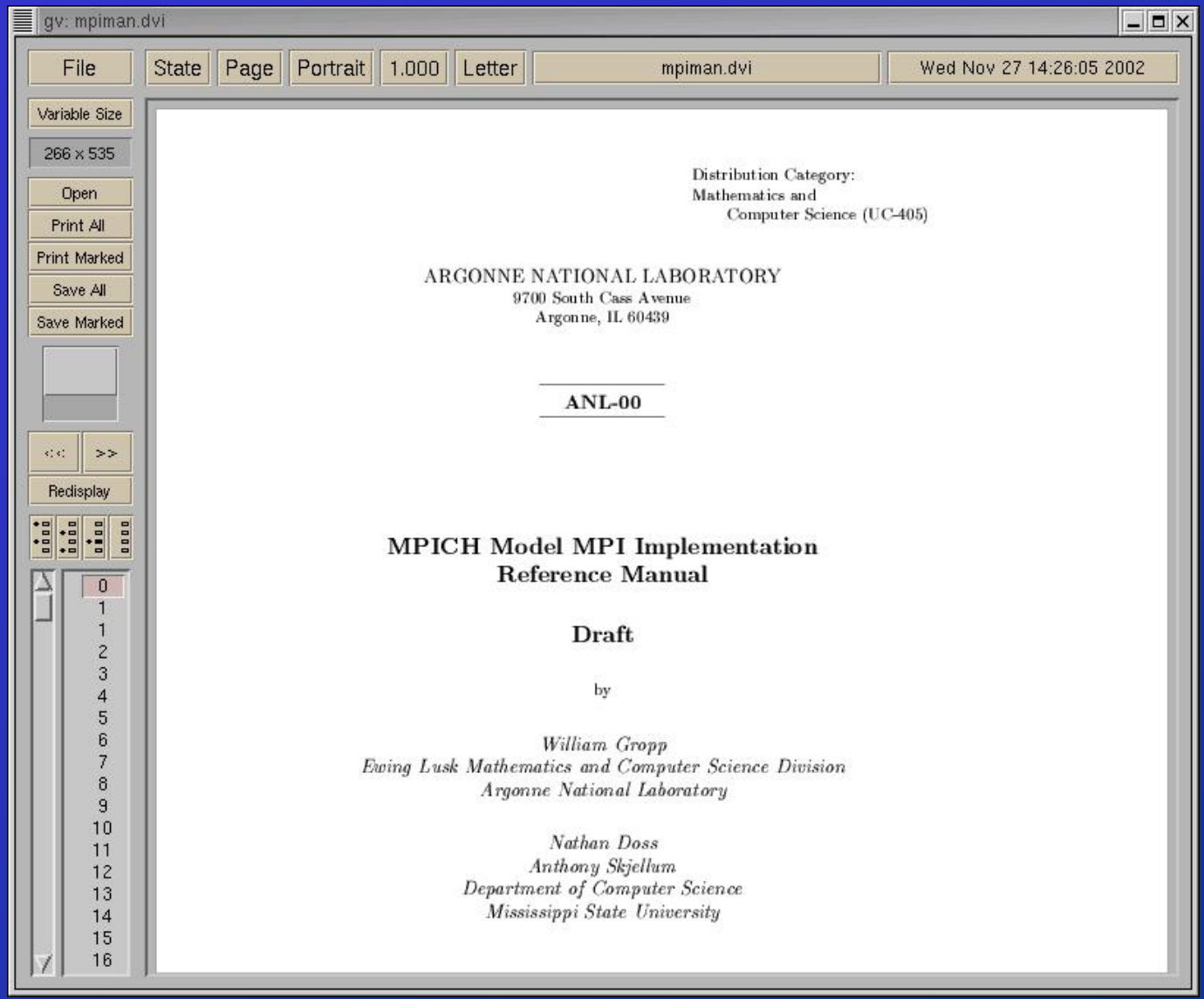
A Scalable Linear Algebra Package

La distribuzione dei dati adottata da ScaLapack è quella *ciclica a blocchi*

Si è osservato sperimentalmente che questa è la distribuzione dati che permette di avere migliori prestazioni nella maggior parte dei problemi di algebra lineare

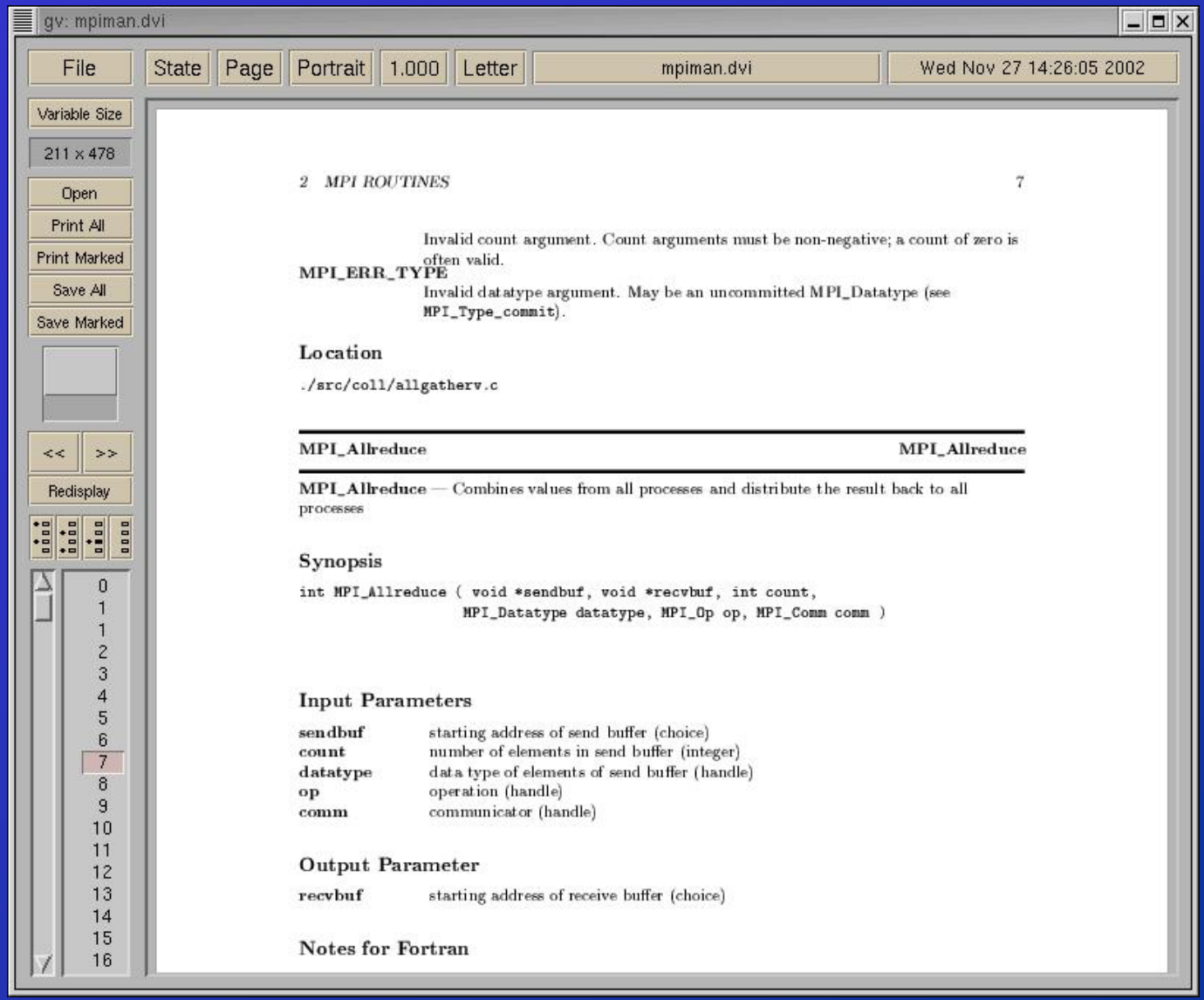


Guide utenti



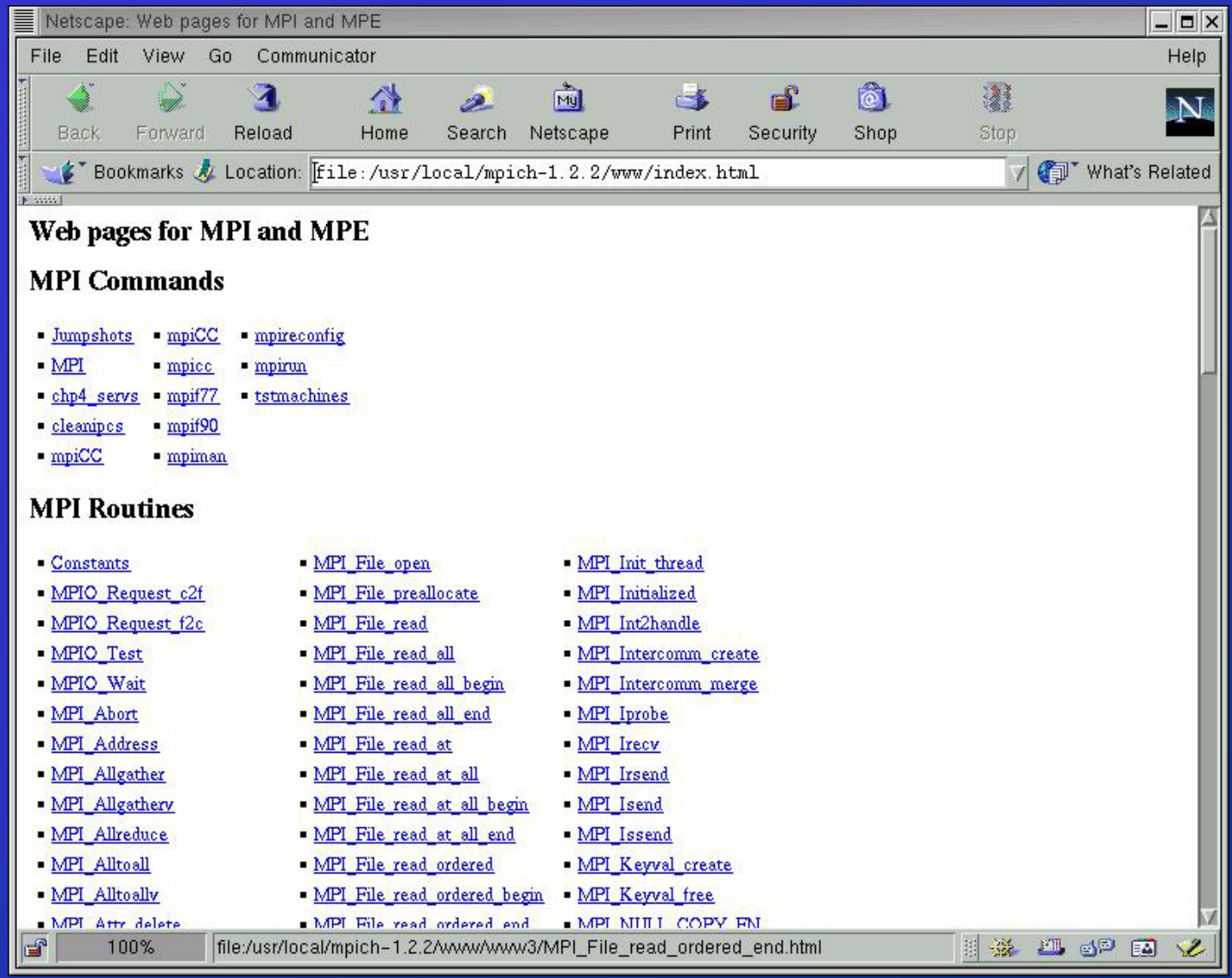
.../mpich/man/mpiman.ps

Guide utenti



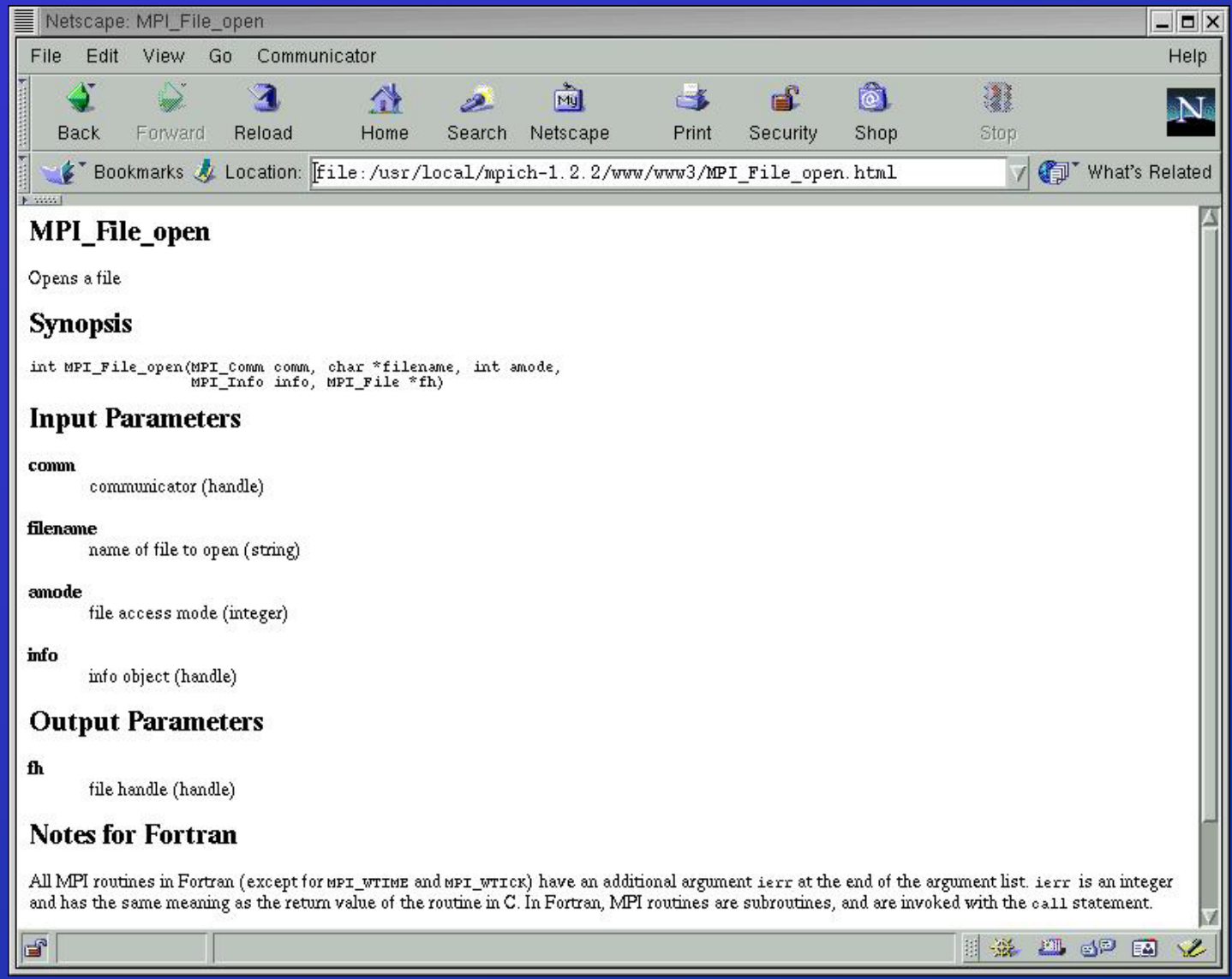
.../mpich/man/mpiman.ps

Guide utenti



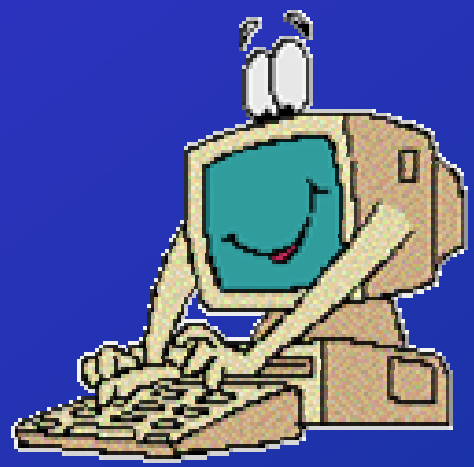
…/mpich/www/index.html

Guide utenti



.../mpich/www/www3/MPI_File_open.html

Buon lavoro



antonio.mucherino@unina2.it