Laboratorio di Calcolo Parallelo

Lezione 1: Introduzione a C ed MPI

Francesco Versaci & Alberto Bertoldo

Università di Padova

5 maggio 2009

Informazioni sul Laboratorio

Il Laboratorio è **facoltativo**

Cosa dovete fare:

1. Seguire le lezioni
2. Iscrivervi al Laboratorio
3. Sperimentare
4. Fare il progetto e la relazione

Per superare l'esame:

- Esame scritto (Prof. Bilardi)
- Relazione (da -2 a +3 punti)
  - Vi verrà comunicata la modalità di presentazione

Organizzazione del Laboratorio

Lezioni

- Oggi: Introduzione a C ed MPI
- Domani (Aula Ke): Aspetti avanzati ed esempi
- Piú avanti: Aspetti pratici

Iscrizione

1. Formare un gruppo (da 1 a 3 persone)
2. Decidere cosa fare ed eleggere un referente
3. Il referente mi spedisce un'email (`versacif@dei.unipd.it`) con:
   - Di ogni componente: Nome, Cognome, Matricola, Account del DEI
   - Argomento del progetto scelto

IMPORTANTE!

Tutti i componenti devono contribuire al progetto

Accesso al Laboratorio

- L'accesso è remoto via SSH
- Possibile sia dalla rete dipartimentale che dall'esterno
- Lezione 3: Guida all'accesso
- Utilizzo prioritario delle postazioni in aula Se:
  - Lunedí 8:00 – 12:00
  - Martedí 8:00 – 12:00
  - Giovedí 8:00 – 12:00
  - Venerdí 12:00 – 18:00

Supporto:

- Via email (`versacif@dei.unipd.it`)
- Non ponetemi questioni di debugging, ma di organizzazione dell'algoritmo
- Per problemi con la macchina, contattare il sistemista che la gestisce (`mazzon@dei.unipd.it`)
- In entrambi i casi: prima di scrivere cercate di risolvere il problema!

Progetto

Il progetto di Laboratorio si compone di:

1. **Implementazione** di un algoritmo parallelo
   - Matrix Multiplication (esempio)
   - Gaussian Elimination
   - FFT
   - Sorting
   - Altro (grafi, elaborazione delle immagini, ecc...)
2. **Valutazione sperimentale** delle prestazioni
   - Correttezza
   - Tempi di esecuzione
   - Requisiti di memoria
   - Speedup ed efficienza
   - Impatto delle comunicazioni
3. **Presentazione** dei risultati
   - Breve presentazione dell'algoritmo scelto e delle scelte implementative
   - Accurata analisi delle prestazioni
   - Non includere il codice nella relazione
   - Allegare il codice sorgente e le istruzione per compilarlo ed eseguirlo

Sommario

1. Il Linguaggio C
2. Il Compilatore GCC
3. Come fare i Makefile
4. Programmazione parallela
   - Modelli di memoria
   - Modelli di programmazione
5. Introduzione a MPI
   - Ambiente di esecuzione
   - Comunicazioni punto-punto

Il libro sacro. . .

Brian W. Kernighan, Dennis M. Ritchie
Il Linguaggio C (II Edizione)
Pearson – Prentice Hall, 27 €

Un po' di storia...

Dall'ALGOL al C

1999 ISO C99

1989-90 Ansi C (C89, ISO C90)

1978 Esce *The C Programming Language* (K&R)

1969-73 Nasce il C (Ritchie, AT&T Bell Labs)

$\sim$1969 B (Thompson)

1966 BCPL (Richards, University of Cambridge, in visita all'MIT)

1963 CPL (University of Cambridge and London)

1958 ALGOL (AA. VV., ETH Zurigo)

Dennis Ritchie e Ken Thompson – *Turing Award 1983*

Caratteristiche

> *C is a language that combines all the elegance and power of assembly language with all the readability and maintainability of assembly language*
>
> Jargon file

- Curly bracket language (come C++, Java, . . .)
- Molto semplice
- *Facile* da compilare
- Vicino alla macchina
- Ideale per ottenere alte prestazioni
- Programmazione a oggetti non esplicita

Hello, World!

hello.c

```
#include <stdio.h>

main(){
   printf("hello, world\n");
}
```

Compilazione

```
$ cc hello.c
$ ./a.out
hello, world

$ gcc -O3 -o hello hello.c
$ ./hello
hello, world
```

Tipi di dati

ANSI C

- `char` 1 byte
- `int` una word
- `float` numero in virgola mobile in precisione singola
- `double` numero in virgola mobile in precisione doppia
- `unsigned int` intero non negativo

C99

- `int8_t` intero a 8 bit
- `int16_t` intero a 16 bit
- `uint32_t` intero a 32 bit senza segno
- `int64_t` intero a 64 bit
- `bool` booleano
- `complex` numero complesso

Note

- In C i dati non hanno dimensione fissata (per adattarsi alla macchina mantenendo le prestazioni)
- I booleani sono semplici interi (FALSO=0, VERO=diverso da 0)
- Le stringhe sono array di char, terminate da NULL (`\0=0x00`)

int printf (char *format, arg1, arg2, ...)

- scrive sullo stdout gli argomenti, formattati secondo `format`
- restituisce il numero di caratteri visualizzati

Esempi

```
int x=13; char c[]="cows";
printf("number of %s: %d\n", c, x );
/* number of cows: 13 */
printf("%.3s\n", c);
/* cow */

const double pi=3.141592653589;
printf("%10.5f\n", pi);
/*    3.14159 */
```

Formattazione dell'output: `printf`

Parametri di output

CARATTERE	TIPO DELL'ARGOMENTO; STAMPATO COME
d,i	`int`; numero in notazione decimale
x,X	`unsigned int`; numero in notazione esadecimale (senza `0x` in testa)
c	`int`; singolo carattere
s	`char*`; stampa la stringa fino a raggiungere `0x00` o la lunghezza indicata
f	`double`; [-]i.dddddd, con i parte intera e dddddd cifre decimali
g	`double`; stampa i double in notazione esponenziale
e	`double`; sceglie fra f e g a seconda della grandezza del numero
p	`void *`; puntatore

- Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile.
- I puntatori possono essere incrementati per scorrere dei vettori (gli iteratori in C++ sono una generalizzazione dei puntatori)

Esempio

```
int x=10;
int *p=&x;  /* p punta a x */
(*p)++;  /* incrementa x */

double d[]={1.2, 2.3, 3.4};
double* p=d;
p++;
printf("%g\n", *p); /* 2.3 */
```

swap.c – Sbagliato

```
#include <stdio.h>

void swap(int a, int b){
  int c=b;
  b=a; a=c;
}

main(){
  int a=10; int b=20;
  printf("a: %d  b: %d\n",a,b);
  swap(a,b);
  printf("a: %d  b: %d\n",a,b);
}
```

```
$ gcc swap.c && ./a.out
a: 10  b: 20
a: 10  b: 20
```

swap.c – Corretto

```
#include <stdio.h>

void swap(int *a, int *b){
  int c=*b;
  *b=*a; *a=c;
}

main(){
  int a=10; int b=20;
  printf("a: %d  b: %d\n",a,b);
  swap(&a,&b);
  printf("a: %d  b: %d\n",a,b);
}
```

```
$ gcc swap.c && ./a.out
a: 10  b: 20
a: 20  b: 10
```

Puntatori e Vettori

scan.h

```
void v_scan(int x[], int n);
void p_scan(int *p);
```

scan.c

```
#include <stdio.h>

void v_scan(int x[], int n){
  int i=0;
  for(; i<n; ++i){
    int buf=x[i];
    if (buf>900)
      printf("%d\n",buf);
  }
}
void p_scan(int *p){
  int buf;
  while ((buf=*p++)>0)
    if (buf>900)
      printf("%d\n",buf);
}
```

main.c

```
#include <stdlib.h>
#include "scan.h"

#define N 100

int main(){
  int x[N];

  int i=0;
  /* inizializzazione valori */
  for(; i<N-1; ++i)
    x[i]=rand()%1000;
  /* terminatore */
  x[N-1]=-1;

  v_scan(x, N);
  p_scan(x);

  return 0;
}
```

Argomenti della Riga di Comando

args.c

```
#include <stdio.h>

int main(int argc, char* argv[]){
  int i=0;
  for(; i<argc; ++i)
    printf("Argomento %d: %s\n", i, argv[i]);

  return 0;
}
```

Compilazione ed esecuzione

```
$ cc -o args args.c && ./args foo bar
Argomento 0: ./args
Argomento 1: foo
Argomento 2: bar
```

Strutture

complex.c

```
#include <stdio.h>

typedef struct {
  double re;
  double im;
} co;

co prod(co x, co y){
  co z;
  z.re=x.re*y.re-x.im*y.im;
  z.im=x.im*y.re+x.re*y.im;
  return z;
}

main(){
  co x,z;
  x.re=0; x.im=1; /* x=i */
  z=prod(x,x); /* z=i^2=-1 */
  printf("(%g +i %g)\n",z.re, z.im);
}
```

- Definiscono nuovi tipi composti
- Antesignani delle classi C++

Allocazione Dinamica e I/O

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  int n, i, *dyn; FILE* fout;
  if (argc!=2)
    return 1;

  sscanf(argv[1], "%d", &n);

  dyn=malloc(n*sizeof(int));

  for(i=0; i<n; ++i)
    dyn[i]=rand()%1000;

  fout=fopen("data", "w");
  fwrite(dyn, sizeof(int), n, fout);
  fclose(fout);

  free(dyn);

  return 0;
}
```

- La funzione `sscanf` legge i valori da una stringa
- `malloc` alloca dinamicamente un'area di memoria
- `free` la libera
- `fopen` apre un file
- `fclose` lo chiude
- `fwrite` scrive sul file

Sommario

1. Il Linguaggio C
2. Il Compilatore GCC
3. Come fare i Makefile
4. Programmazione parallela
   - Modelli di memoria
   - Modelli di programmazione
5. Introduzione a MPI
   - Ambiente di esecuzione
   - Comunicazioni punto-punto

Passi per la creazione dell'eseguibile

preprocessing Vengono espanse le macro (fra cui le direttive `#include`)

compilation Il sorgente viene convertito in linguaggio *assembly*

assembling Dall'*assembly* al linguaggio macchina

linking Si uniscono i vari pezzi per formare l'eseguibile finale

runtime Vengono caricate in memoria le eventuali librerie condivise

Estensioni riconosciute

Le estensioni riconosciute dal compilatore come file C sono le seguenti:

header `.h`

codice `.c`

Opzioni per l'output

`gcc [-c] [-o output] nome.c`

-c Compila senza collegare, non crea l'eseguibile

-o *nomeoutput* Chiama il file di uscita "nomeoutput"

scan.h

```
void v_scan(int x[], int n);
void p_scan(int *p);
```

scan.c

```
#include <stdio.h>

void v_scan(int x[], int n){
  int i=0;
  for(; i<n; ++i){
    int buf=x[i];
    if (buf>900)
      printf("%d\n",buf);
  }
}
void p_scan(int *p){
  int buf;
  while ((buf=*p++)>0)
    if (buf>900)
      printf("%d\n",buf);
}
```

main.c

```
#include <stdlib.h>
#include "scan.h"

#define N 100

int main(){
  int x[N];

  int i=0;
  /* inizializzazione valori */
  for(; i<N-1; ++i)
    x[i]=rand()%1000;
  /* terminatore */
  x[N-1]=-1;

  v_scan(x, N);
  p_scan(x);

  return 0;
}
```

```
$ gcc -c scan.c
$ gcc -c main.c
$ gcc -o runme main.o scan.o
$ ./runme
[output]
```

Spiegazione

- I primi due comandi compilano i file sorgenti creando i file oggetto `.o`.
- Il terzo li collega creando l'eseguibile `runme`, lanciato alla quarta riga.

Le prime tre righe si sarebbero potute sostituire con questa:

```
gcc -o runme scan.c main.c
```

Queste opzioni segnalano in fase di compilazione delle probabile sviste del programmatore che non danno errori in compilazione.
È consigliabile usarle sempre.

Warnings

-Wall Abilita molti avvisi utili (variabili usate senza inizializzazione, classi polimorfe senza distruttore virtuale, ecc.)

–ansi Seleziona l'ANSI C

–pedantic Avvisa se si devia dallo standard ISO

-W (-Wextra) Abilita altri avvisi (confronto di unsigned con 0, funzioni che possono o meno restituire un valore, ecc.)

Il gcc può utilizzare diverse tecniche per ottimizzare l'eseguibile prodotto.

Opzioni per l'ottimizzazione

- -O1 Ottimizza salti condizionali e cicli, prova a eliminare alcuni if, srotola i cicli, ...
- -O2 Ottimizza l'uso di sottoespressioni, riordina il codice minimizzando i salti, ...
- -O3 Rende inline le funzioni piccole, ...

Opzioni di linking

-l*nome* Usa la libreria esterna “nome”

-L*dir* Cerca le librerie nelle directory “dir”

-shared Crea un file oggetto condiviso, da usare come libreria dinamica

-static Collega tutte le librerie in modo statico (anche le condivise)

-s Togli la tavola dei simboli dall’eseguibile

pi.c

```
#include<stdio.h>
#include<math.h>

int main(){
  printf("pi=%.10f\n", 4*atan(1));

  return 0;
}
```

Compilazione

```
$ gcc -o runme pi.c -lm
$ ./runme
pi=3.1415926536
```

Per usare una libreria esterna è necessario installare il pacchetto con gli header (in Linux *libqualcosa-dev*), includere l'header e dichiarare al linker in quali librerie recuperare gli oggetti (si veda il manuale della libreria usata).

Sommario

1. Il Linguaggio C
2. Il Compilatore GCC
3. Come fare i Makefile
4. Programmazione parallela
   - Modelli di memoria
   - Modelli di programmazione
5. Introduzione a MPI
   - Ambiente di esecuzione
   - Comunicazioni punto-punto

Per automatizzare la compilazione dei sorgenti lo strumento standard è il `make`. Noi analizzeremo la versione GNU di questo programma.

Il `Makefile`

In ogni directory che contiene sorgenti è necessario creare un file di testo, chiamato *Makefile*, che contiene le istruzioni per la compilazione.

Come avviare la compilazione

Una volta creato il Makefile è sufficiente lanciare il comando `make` (o `make obiettivo`) per avviare la compilazione.

Sintassi delle regole

```
obiettivo: prerequisiti
        comando1
        comando2
        ...
```

Spiegazione

obiettivo Il file da creare

prerequisiti I file che servono per crearlo

comando I comandi (si noti il *tab* obbligatorio) per creare l'*obiettivo*

Cosa fa il make?

Controlla se qualcuno frai *prerequisiti* è stato modificato piú recentemente dell'*obiettivo*. In caso affermativo esegue i *comandi*.

Makefile

```
grosso.txt:  piccolo1.txt piccolo2.txt
        cat piccolo1.txt piccolo2.txt > grosso.txt
```

Il comando `make grosso.txt` controlla che il file *grosso.txt* non esista oppure sia piú vecchio dei file *piccolo1.txt* e *piccolo2.txt*. Quindi esegue il comando *cat* che crea il file *grosso.txt*.
Nel caso l'obiettivo sia già aggiornato make notifica la cosa con la seguente stringa:

```
make: 'grosso.txt' is up to date.
```

Spesso è comodo poter creare dei comandi svincolati da particolari file.
Si usano allora degli obiettivi *phony*.

Makefile

```
.PHONY: clean
clean:
        rm *.o
```

Spiegazione

Eseguire il comando `make clean` è equivalente ad eseguire `rm *.o`

Nei Makefile è possibile utilizzare delle variabili per evitare di riscrivere piú volte gli stessi comandi.

Makefile

```
CFLAGS = -Wall -W -pedantic -ansi

scan.o : scan.c
	gcc $(CFLAGS) -c scan.c

main.o : main.c
	gcc $(CFLAGS) -c main.c
```

Spesso si deve eseguire lo stesso comando su piú file.

Makefile

```
CFLAGS = -Wall -W -pedantic -ansi
%.o : %.c
        gcc $(CFLAGS) -c -o $@ $<
```

Regole implicite

La seconda riga fornisce una regola per trasformare qualunque file `.c` in un file `.o`

Variabili automatiche

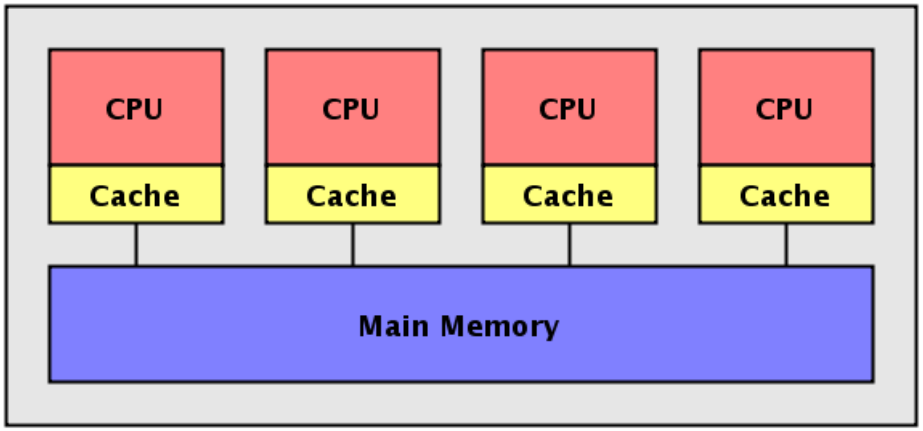
- `$@` Bersaglio
- `$<` Primo prerequisito
- `$^` Tutti i prerequisiti

```
CXXFLAGS = -Wall -W -pedantic

%.o : %.cc
        g++ $(CXXFLAGS) -o $@ -c $<

runme :  test.o saluti.o
        g++ $(CXXFLAGS) -o $@ $^
```

Sommario

1. Il Linguaggio C
2. Il Compilatore GCC
3. Come fare i Makefile
4. Programmazione parallela
   - Modelli di memoria
   - Modelli di programmazione
5. Introduzione a MPI
   - Ambiente di esecuzione
   - Comunicazioni punto-punto

Architettura a memoria condivisa



Caratteristiche:

- Spazio di indirizzamento globale
- Processori uguali = SMP
- Può essere UMA or NUMA
- Esempi:
  - SGI Origin 3000
  - IBM Server p595

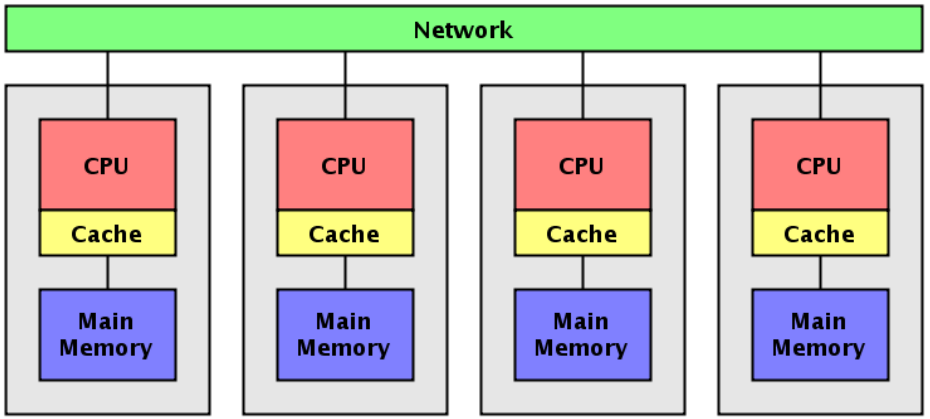
Pro

- Facilità di programmazione (multithreading/IPC)
- Condivisione dati veloce
- Il SO può controllare il bilanciamento del carico

Contro

- Collo di bottiglia tra CPU e memoria
- Scalabilità limitata
- Coerenza della cache

Architetture a memoria distribuita



Caratteristiche:

- Spazio di indirizzamento locale
- Richiede una rete di interconnessione
- Esempi:
  - Cray T3E
  - CoW (Beowulf)

Pro

- Alta scalabilità (n. di CPU e memoria)
- Costi limitati (es. Beowulf)

Contro

- Difficili da programmare
- Occorre distribuire i dati
- Costo delle comunicazioni

Modelli di programmazione parallela più comuni:

- Shared-memory (memoria condivisa)
- Multithreading
- Message passing (scambio di messaggi)
- Data parallel
- Ibridi

- **Astrazione** delle architetture di memoria
- Possono (teoricamente) essere realizzati su qualsiasi architettura di memoria:
  - Modello a memoria condivisa su macchine a memoria distribuita
  - Modello a scambio di messaggi su macchine a memoria condivisa (es. MPI)

Single-Program Multiple-Data

I task:

- Eseguono tutti lo stesso programma
- Possono eseguire istruzioni diverse
- Possono usare dati diversi
- Comunicano tra loro e si sincronizzano

Adatto per applicazioni parallele regolari

Multiple-Program Multiple-Data

- I task eseguono programmi diversi
- Adatto per applicazioni parallele irregolari, e.g. Master/Worker

Sommario

1. Il Linguaggio C
2. Il Compilatore GCC
3. Come fare i Makefile
4. Programmazione parallela
   - Modelli di memoria
   - Modelli di programmazione
5. Introduzione a MPI
   - Ambiente di esecuzione
   - Comunicazioni punto-punto

Cos'è MPI?

MPI = Message Passing Interface

Con il termine MPI si intendono tre cose:

1. **Paradigma** di programmazione a scambio di messaggi
2. Insieme di **specifiche** che definiscono una API
3. **Libreria** per la programmazione di applicazioni parallele

In realtà sarebbe corretto dire che:

1. Message passing è il paradigma di programmazione
2. **MPI è una API** (protocollo di comunicazione e semantica delle operazioni)
   - Non è uno standard *de iure*
   - È lo standard *de facto* per la programmazione message passing
   - Se ne occupa un apposito forum: `http://www.mpi-forum.org`
3. Esistono varie implementazioni di MPI, es: OpenMPI, MPICH, IBM MPL, ecc.

Perché MPI?

Pro

- Indipendenza dall'implementazione
  - Librerie proprietarie, es. IBM MPI
  - Open Source, es. OpenMPI
- Indipendenza dall'architettura di calcolo e di rete
  - Shared-memory, NUMA, distributed memory, multithreading, ecc.
  - Little-endian, big-endian, ecc...
  - Infiniband, TCP, Myrinet, ecc...
- Indipendenza dal linguaggio
  - C/C++, Java, Fortran, ecc...

$\Rightarrow$ **Portabilità, scalabilità, efficienza**

Contro

- Parallelismo troppo a basso livello
  - Controllo diretto su come, quando e tra chi scambiare messaggi
- Difficoltà di utilizzo
  - Sincronizzazione, deadlock

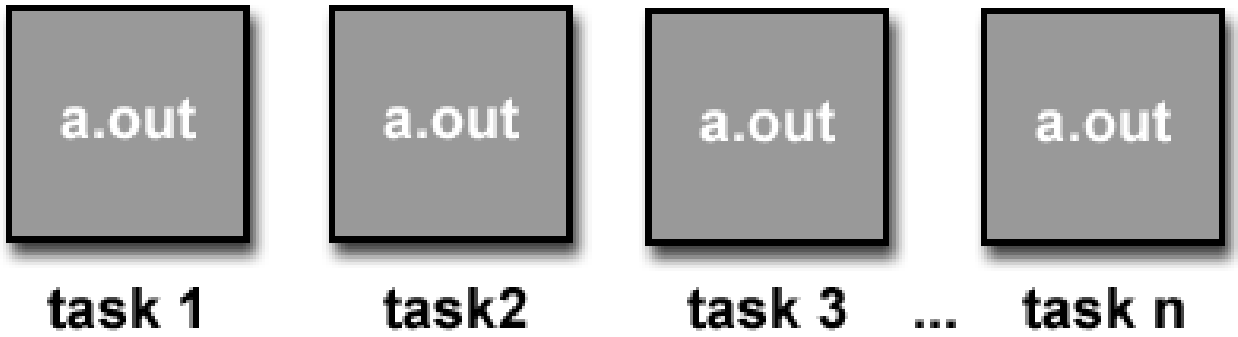
Panoramica su MPI

Versioni principali dello standard

- MPI-1.2 (Detta anche MPI-1)
  - Primitive tradizionali per lo scambio di messaggi
  - Ambiente di esecuzione statico
- MPI-2.1 (Detta anche MPI-2)
  - I/O parallelo
  - Ambiente di esecuzione dinamico
  - Accesso alla memoria remota

Primitive principali in MPI-1:

- Gestione dell'ambiente di esecuzione
- Comunicazioni punto-punto
- Comunicazioni collettive



- Il sistema run-time lancia n processi
- Ogni processo:
  - esegue una copia di `a.out` indipendentemente
  - ha il suo spazio di memoria locale
  - può essere mappato su un processore diverso
  - ha una sua identità
- `a.out` può essere un programma MPI

Ora ci occupiamo solo di programmi MPI

Definizione

Un **comunicatore** è un insieme di processi MPI che possono comunicare tra loro

- Ha un nome che lo identifica
- Ha una dimensione (numero di processi)
- Ogni processo può essere identificato univocamente
- I processi sono tra loro equivalenti

- Esiste un comunicatore di default
- Ha un nome: `MPI_COMM_WORLD`
- Comprende gli n processi lanciati dal sistema run-time
- I processi sono identificati con un intero da 0 a $n-1$
- Possono essere definiti altri comunicatori, anche non disgiunti

Inizializzare MPI

int MPI_Init(**int** *argc, **char** ***argv)

- Deve essere chiamata da tutti i processi prima di ogni altra chiamata MPI

Terminare MPI

int MPI_Finalize(**void**)

Dato un comunicatore `comm` (per noi è sempre `MPI_COMM_WORLD`):

Quanti siamo?

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

- `size` è la dimensione del comunicatore

Chi sono **io**?

int MPI_Comm_rank(MPI_Comm comm, **int** *rank)

- `rank` è l'identificatore del processo corrente (da 0 a `size`-1)

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

- `MPI_COMM_WORLD` è definito in `mpi.h`
- Ogni comando è eseguito da ogni processo indipendentemente
- La compilazione produce un unico eseguibile
- Il sistema run-time dipende dall'implementazione e definisce:
  - Se e come si replica l'eseguibile nei nodi di calcolo
  - Come si lanciano i processi
  - Come vengono gestiti lo standard output/error

Esempio di esecuzione:

```
bash-2.05$ mpcc hello.c
bash-2.05$ poe ./a.out -procs 4
Hello world from process 1 of 4
Hello world from process 0 of 4 <-- NOTARE L'ORDINE
Hello world from process 2 of 4
Hello world from process 3 of 4
bash-2.05$
```

Comunicazione tra due processi MPI

- Copia di dati tra spazi di memoria distinti
- Permette di identificare univocamente i messaggi
- Usa tipi di dati propri
- Le primitive possono essere **bloccanti** o **non bloccanti**
- Varie modalità di comunicazione:

  Standard: buffering e sincronizzazione automatici
  Buffered: buffering utente intermedio
  Sincrona: rendevouz stretto
  Pronta: spedizione immediata (no handshaking)

I messaggi sono composti da due parti:

Intestazione: permette di identificare univocamente il messaggio

Dati: ciò che si vuole scambiare tra i processi

L'intestazione è composta da:

Sorgente: è l'identificatore (nel comunicatore) del processo che invia il messaggio

Destinazione: è l'identificatore (nel comunicatore) del processo che deve ricevere il messaggio

Tag: identifica il messaggio (intero da 0 a `MPI_TAG_UB`)

Comunicatore: è il contesto di comunicazione (già visto)

I dati sono composti da:

Tipo: tipo di dati MPI (tabella)

Lunghezza: come numero di elementi

Buffer: array di elementi consecutivi nella memoria utente

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

Spedire un messaggio

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

- Ritorna quando il messaggio è stato copiato dal sistema
- Quando ritorna `buf` puo' essere sovrascritto
- Non è detto che quando ritorna il messaggio sia stato ricevuto!!!
- Potrebbe essere nella memoria di sistema del mittente

Note sul buffering in spedizione

- Dipende dall'implementazione di MPI
- Può essere controllato dall'utente (lo vedremo prossimamente)
- **Pro:** disaccoppia send e recv, elimina la sincronizzazione
- **Contro:** memoria aggiuntiva, tempo per la copia

Ricevere un messaggio

```
int MPI_Recv(void* buf, int count, \
             MPI_Datatype datatype, \
             int source, int tag, \
             MPI_Comm comm, MPI_Status *status)
```

- Ritorna quando il messaggio è stato ricevuto nel buffer nello spazio utente
- `count` deve essere $\geq$ della lunghezza del msg ricevuto
- `source` può essere `MPI_ANY_SOURCE`
- Il `tag` identifica quale messaggio da `source` si vuole ricevere
- `tag` può essere `MPI_ANY_TAG`
- `status` contiene informazioni aggiuntive

Attenzione

I tipi di dato devono coincidere con quelli della send corrispondente!

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    buf = 123456;
    MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("%d has sent %d to %d\n", rank, buf, 1);
  }
  else if (rank == 1) {
    MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("%d has received %d from %d\n", rank, buf, 0);
  }
  MPI_Finalize();
  return 0;
}
```

- Stesso codice eseguibile, ma...
- ... processi diversi eseguono parti di codice diverse
- Solo p_0 inizializza `buf`
- p_1 ottiene il valore 123456 in `buf` solo dopo aver ricevuto il messaggio da p_0

Esempio di esecuzione:

```
bash-2.05$ mpcc blocking.c
bash-2.05$ poe ./a.out -procs 8
0 has sent 123456 to 1
1 has received 123456 from 0
bash-2.05$
```

Cosa fanno i processi $p_2 \ldots p_7$?

Riferimenti

- Per lo standard:
  `http://www.mpi-forum.org/docs`
- Tutorial:
  `https://computing.llnl.gov/tutorials/mpi`
- Per l'implementazione IBM:
  `http://www.dei.unipd.it/~addetto/manuali_online/index.html`
- In particolare:
  `http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf`