Università degli Studi di Napoli Parthenope

Corso di Laurea Triennale in Informatica

Manuale utente MPI in FORTRAN

Francesco Fossari

Report creato basandosi su "MPI's User Guide in FORTRAN"
di Dr.Peter S.Pacheco e Woo Chat Ming
per il Corso di Laurea Triennale in *Informatica*

April 14, 2025

Dichiarazione

Io, Francesco Fossari, del Dipartimento di Informatica, Università degli Studi di Napoli Parthenope, confermo che questo è il mio lavoro e che figure, tabelle, equazioni, frammenti di codice, opere d'arte e illustrazioni in questo rapporto sono originali e non sono stati presi dal lavoro di nessun'altra persona, tranne nei casi in cui i lavori di altri siano stati esplicitamente riconosciuti, citati e referenziati. Comprendo che se non lo faccio sarà considerato un caso di plagio. Il plagio è una forma di cattiva condotta accademica e sarà penalizzato di conseguenza.

Acconsento alla condivisione di tale materiale come esempio per supportare altri studenti.

Acconsento che il mio lavoro venga reso disponibile più ampiamente al pubblico interessato all'insegnamento, all'apprendimento e alla ricerca.

Francesco Fossari
April 14, 2025

Contents

Chapter 1

Introduzione

1.1 Breve Presentazione del Message Passing Interface

Message-Passing Interface o MPI è una libreria di funzioni e macro che può essere utilizzata in C, C++ e FORTRAN. MPI è stata sviluppata da un gruppo di ricercatori dell'industria, ricercatori governativi e accademici tra il 1993 e il 1994. Questa guida è stata scritta da Pacheco (1995), elaborata da Pacheco and Ming (1995) e tradotta da Francesco Fossari, ed è una piccola introduzione per alcune delle più importanti feature della libreria MPI per i programmatori in FORTRAN. Molti utilizzatori di computer paralleli che sono nella comunità scientifica ed ingegneristica, usano FORTRAN come linguaggio principale, motivo per cui i vari esempi presenti in questo report sono implementati usando questo linguaggio di programmazione.

1.2 Importanza di MPI nel Calcolo Parallelo

Questa libreria risulta essere fondamentale per far scambiare messaggi tra sistemi distribuiti e cluster di computer che eseguono un programma parallelo attraverso una memoria distribuita.

1. **Efficienza e Scalabilità** L'utilizzo di MPI permette la suddivisione di problemi in sotto-problemi, in modo da assegnare ad ognuno di essi un compito preciso diverso, riducendo il tempo d'esecuzione. Ottimizzando l'uso delle risorse, MPI ci permette di distribuire il carico di lavoro in modo efficace.

2. **Flessibilità e Portabilità** Come accennato precedentemente, MPI è supportato da diversi linguaggi di programmazione, come FORTRAN, C, C++ ed è compatibile con sistemi operativi diversi. Inoltre, è indipendente dall'architettura del sistema e per questo può essere eseguito su cluster di computer, supercomputer e cloud computing.

3. **Comunicazione efficiente tra processi** MPI fornisce meccanismi di comunicazione robusti, come lo scambio di messaggi punto-punto e le operazioni collettive (broadcast, scatter, gather). Supporta sincronizzazione e coordinamento tra processi, evitando situazioni di conflitto o deadlock.

4. **Settori d' Applicazione** La libreria MPI può essere utilizzata in settori che richiedono un'elevata potenza di calcolo, come Analisi massiva di Dati, Machine Learning e Intelligenza Artificiale per distribuire il carico di calcolo nei training dei modelli. Inoltre, è utilizzata anche per simulazioni scientifiche, come previsioni meteo, oltre ad alcuni utilizzi nei settori chimici e fisici.

In poche parole, MPI è una tecnologia essenziale quando si vuole lavorare con un grande numero di dati e risulta necessario risolvere problemi complessi in tempi significativamente ridotti rispetto ai modelli sequenziali. Infine, grazie alla sua efficienza e flessibilità, si è affermato come standard fondamentale per il calcolo parallelo ad alte prestazioni.

1.3 Obiettivo

Questo report ha l'obiettivo di introdurre e fornire una guida introduttiva in lingua italiana sull'utilizzo della libreria MPI nel linguaggio di programmazione FORTRAN. Sebbene molti studenti dell'Università degli Studi di Napoli programmino principalmente in C e C++, questo documento mira a incentivare l'uso di FORTRAN, offrendo ai lettori un'opportunità per ampliare le proprie conoscenze e competenze nel calcolo parallelo. Oltre a servire come manuale pratico e intuitivo, il report illustra i principi fondamentali della programmazione parallela con MPI attraverso esempi esplicativi, facilitando l'apprendimento e l'applicazione della libreria in contesti scientifici e ingegneristici.

Chapter 2

Notazioni di Base su MPI in FORTRAN

2.1 Inizializzazione e Terminazione

Nel momento in cui è necessario compilare un programma che faccia uso della libreria MPI, è necessario come primo passo; includere la libreria "mpif.h".

`include 'mpif.h'`

Il seguente file contiene tutte le funzioni e macro, necessarie per compilare un programma MPI. Prima di poter utilizzare una funzione della libreria 'mpif.h', è necessario chiamare la funzione `MPI_Init` una sola volta, mentre per terminare un programma che ha terminato l'utilizzo di MPI bisogna chiamare la funzione `MPI_Finalize`. Quest'ultima chiamata termina ogni possibile task lasciata incompleta, per esempio mostreremo la struttura di un semplice programma MPI :

```
1  program mpi_example
2      include 'mpif.h'
3
4      integer ierr
5
6      call MPI_Init(ierr)
7
8      ! Parte del Codice Parallelo
9
10     call MPI_Finalize(ierr)
11
12 end program mpi_example
```

Listing 2.1: Struttura di un programma MPI

Le routine MPI in FORTRAN hanno un argomento integer "IERROR", che restituisce l'errore verificatosi durante l'esecuzione di `mpi_example`.

2.2 Rango del Processo

Esiste un'altra funzione disponibile nella libreria MPI che permette all'utente di identificare il rango di un processo, restituito come secondo argomento della funzione `MPI_COMM_RANK`, vediamo l' implementazione di seguito :

```
1  program mpi_example
2      include 'mpif.h'
3
4      integer ierror
5      integer comm
6      integer rank
7
8      call MPI_COMM_RANK(COMM, RANK, IERROR)
9
10 end program mpi_example
```

Listing 2.2: utilizzo di `MPI_COMM_RANK`

Notiamo che questa funzione è costituita da tre parametri, dove il primo argomento è un communicator, cioè una collezione di processi che possono scambiare messaggi tra loro. Noi useremo `MPI_COMM_WORLD` per i nostri programmi base ed è l'unico communicator di cui abbiamo bisogno, incluso di base nel file 'mpif.h'.

```
1  program mpi_example
2      include 'mpif.h'
3
4      integer ierror
5      integer comm
6      integer p
7
8      call MPI_COMM_SIZE(COMM, P,IERROR)
9
10 end program mpi_example
```

Listing 2.3: utilizzo di `MPI_COMM_SIZE`

Il numero di processi che eseguono un programma giocano un ruolo fondamentale per i costrutti della libreria MPI, per questo tale libreria fornisce la funzione `MPI_COMM_SIZE` che ritorna il numero di processi presenti nel communicator, come secondo argomento.

2.3 Scambio di Messaggi tra Processi

Utilizziamo la libreria MPI, per permettere lo scambio di messaggi tra due o più processi attraverso l'utilizzo delle funzioni `MPI_Send` e `MPI_Recv`. La prima funzione permette di inviare un messaggio ad un processo designato, la seconda invece; consente la ricezione di un messaggio da parte di un processo. La sintassi di `MPI_SEND` è la seguente :

```
program mpi_example
    include 'mpif.h'

    integer count
    integer datatype
    integer dest
    integer tag
    integer comm
    integer ierror

    <TYPE> message(*)
    call MPI_SEND(message, count, datatype,dest,tag,comm,ierror)

end program mpi_example
```

Listing 2.4: utilizzo di `MPI_SEND`

Di seguito, la sintassi di `MPI_RECV` :

```
program mpi_example
    include 'mpif.h'

    integer count
    integer datatype
    integer source
    integer tag
    integer comm
    integer status(MPI_STATUS_SIZE)
    integer ierror
    <TYPE> message(*)

    call MPI_RECV(message, count, datatype,source,tag,comm,status,ierror)

end program mpi_example
```

Listing 2.5: utilizzo di `MPI_RECV`

Iniziamo a parlare del **Tag**, argomento necessario per distinguere un insieme di messaggi inviati da un singolo processo. MPI garantisce che possono essere usati numeri interi inclusi tra 0-32767 come Tags, ma alcune implementazioni permettono l'uso di valori interi molto più ampii. Per quanto riguarda il messaggio, il suo contenuto è memorizzato in un blocco di memoria referenziato da `message`. Inoltre, i parametri `count` e `datatype` permettono al sistema di identificare la fine del messaggio, infatti contiene una sequenza di valori `count`, ognuno avente come MPI type `datatype`. Costruiamo una tabella per confrontare i types predefiniti di MPI e di FORTRAN, eccoli di seguito:

MPI datatype	**FORTRAN datatype**
`MPI_INTEGER`	INTEGER
`MPI_REAL`	REAL
`MPI_DOUBLE_PRECISION`	DOUBLE PRECISION
`MPI_COMPLEX`	COMPLEX
`MPI_LOGICAL`	LOGICAL
`MPI_CHARACTER`	CHARACTER(1)
`MPI_BYTE`	-
`MPI_PACKED`	-

Table 2.1: Corrispondenza tra i tipi di dati MPI e FORTRAN

MPI permette di allocare uno spazio di memoria per il messaggio fin quando c'è spazio sufficiente, nel caso non fosse cosi allora si verificherà un errore di **overflow**[Message Passing Interface Forum (1994)]. Per quanto riguarda **dest** e **source**, essi sono rispettivamente il rango del processo che invia e del processo che riceve il messaggio. Nel momento in cui un processo è inizializzato per ricevere un messaggio da qualsiasi processo "sender", allora possiamo utilizzare la wildcard `MPI_ANY_SOURCE`,ma per ovvi motivi tale macro non può essere usata per **dest**. Inoltre, esiste un'altra wildcard, conosciuta come `MPI_ANY_TAG` che `MPI_RECV` può usare come tag. Esiste un argomento che non deve assolutamente variare sia per il sender che per il receiver, e questo è il communicator(comm). Infatti, se quest'argomento è diverso; lo scambio di messaggi tra il processo X ed il processo Y non può avvenire in modo corretto. L'ultimo argomento dell'`MPI_RECV`, status, ritorna il rango appartenente al processo che ha spedito il messaggio.

Chapter 3

Comunicazione Collettiva

L'obiettivo principale della computazione parallela è quella di sfruttare un insieme di processi che collaborano, affinchè un compito assegnato possa essere risolto in tempi più rapidi. Se cosi non fosse, allora converrebbe usare macchina convenzionale con singolo processore. Ma come possiamo dividere il lavoro a più processi in modo da ridurre il carico di lavoro e distribuire i dati di input uniformemente? Una soluzione potrebbe essere quella di utilizzare un albero dei processi, dove abbiamo 0 alla radice. Supponendo di avere p processi, tramite questa tecnica possiamo distribuire i dati di input in $\log_2 p$ passi, piuttosto che in p passi dove se p è un numero di grandi dimensioni, è un risparmio notevole.

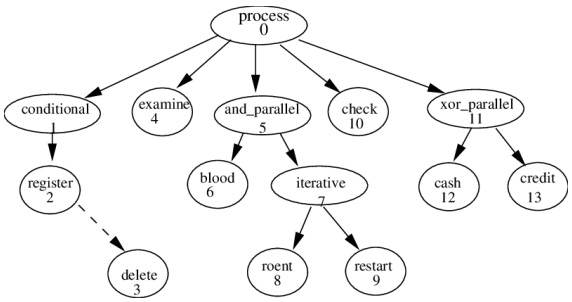

Figure 3.1: Albero dei Processi

Per utilizzare un albero dei processi, bisogna introdurre un cosiddetto loop composto da $\log_2 p$ step. Per far si che il loop funzioni correttamente, ad ogni step il processo preso in considerazione deve calcolare :

- Cosa riceve e da chi?
- Cosa manda e a chi?

Questi calcoli, svolti alle prime armi possono risultare complessi, soprattutto perchè non esiste una scelta canonica per effettuare l'ordinamento.

Purtroppo se non si conosce la topologia della macchina, risulta complicato decidere quale schema sia il migliore. MPI fornisce una funzione che ci permette di non preoccuparci di tali dettagli e che si adatta perfettamente alla macchina che stiamo utilizzando, in modo da non dover modificare il codice ogni tal volta si cambia calcolatore. Parliamo quindi di `BroadCast`, il quale permette ad un processo di inviare un singolo messaggio a più processi, definibile come comunicazione collettiva. La funzione fornita dalla libreria MPI è riportata di seguito :

```
1  program mpi_example
2      include 'mpif.h'
3
4      <TYPE> buffer(*)
5      integer count
6      integer data
7      integer root
8      integer comm
9      integer ierror
10
11     call MPI_BCAST(buffer, count, datatype,root,comm,ierror)
12
13 end program mpi_example
```

Listing 3.1: utilizzo di `MPI_BCAST`

Lo scopo di questa funzione è di semplicemente inviare una copia presente nel buffer a tutti i processi facenti parte di comm(communicator). Dovrebbe essere chiamata da tutti i processi nel communicator con stesso ROOT e stesso COMM. Per quanto riguarda i parametri DATATYPE e COUNT, essi hanno lo stesso scopo ed utilizzo che hanno nell'`MPI_Send` e `MPI_Recv`.

3.1 Riduzione

Bisogna considerare ora un caso diverso e per questo la domanda è differente: siamo sicuri che la nostra struttura dell'albero dei processi sia efficiente per la topologia della nostra macchina? Ovviamente, come successo prima, MPI ci fornisce un'altra funzione che permette di non farci preoccupare e procedere con il nostro lavoro. La funzione che ci permette di fare tutto ciò è colei che segue :

```
1  program mpi_example
2      include 'mpif.h'
3
4      <TYPE> Result(*)
5      <TYPE> Operand(*)
6      integer count
7      integer datatype
8      integer op
9      integer root
10     integer comm
11     integer ierror
12
13     call MPI_Reduce(Operand, Result, Count, Datatype, Op, Root,Comm,Ierror
       )
14
15 end program mpi_example
```

Listing 3.2: utilizzo di MPI_REDUCE

Quest' espressione combina gli operandi salvati in OPERAND usando Op e salva il risultato in RESULT sul processo ROOT.

L'argomento OP può essere sostituito con i seguenti valori predefiniti, dove tipicamente le operazioni binarie sono :

Nome dell' Operazione	Significato
MPI_MAX	Massimo
MPI_MIN	Minimo
MPI_SUM	Somma
MPI_PROD	Prodotto
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwist Or
MPI_LXOR	Logical Exclusive Or
MPI_BXOR	Bitwise Exclusive Or
MPI_MAXLOC	Massimo e la sua locazione
MPI_MINLOC	Minimo e la sua locazione

Table 3.1: Tabella con i sostituti di OP

3.2 Funzioni Alternative per la Comunicazione Collettiva

Grazie ad MPI esistono altre funzioni, le quali sono delle valide alternative a `MPI_Reduce`, un esempio è la funzione seguente :

```
program mpi_example
    include 'mpif.h'

    integer comm
    integer ierror

    call MPI_Barrier(comm, ierror)

end program mpi_example
```

Listing 3.3: utilizzo di `MPI_Barrier`

Per tutti i dettagli, guarda Message Passing Interface Forum (1994).

Tale funzione fornita dalla libreria MPI ci permette di sincronizzare tutti i processi appartenenti allo stesso communicator comm. Un' altra funzione molto utile, sempre fornitaci dalla libreria MPI è :

```
program mpi_example
    include 'mpif.h'

    <type> send_buf(*)
    <type> recv_buf(*)
    integer send_count
    integer send_type
    integer recv_count
    integer recv_type
    integer root
    integer comm
    integer ierror

    call MPI_Gather(send_buf, send_count,send_type, recv_buf,recv_count,
    recv_type,root,comm, ierror)

end program mpi_example
```

Listing 3.4: utilizzo di `MPI_Gather`

Nonostante i vari parametri presenti nella funzione, essa risulta essere molto facile da interpretare e di grande utilità. Si proceda con l'analisi: ogni processo presente in comm invia il contenuto di `send_buf` al processo col rango associato alla radice. Dopo di che, il processo root concatena i dati ricevuti in `Recv_Buf` seguendo l'ordine del rango dei processi. Per ultimo ma non meno importante, abbiamo `Recv_count`; il quale è un contatore che indica il numero di dati ricevuti da ogni processo, ma non il numero totale di dati ricevuti.

Dopo aver trattato le funzioni precedenti, continuiamo a trattarne altre, come :

```
program mpi_example
    include 'mpif.h'

    <type> send_buf(*)
    <type> recv_buf(*)
    integer send_count
    integer send_type
    integer recv_count
    integer recv_type
    integer root
    integer comm
```

```
12      integer ierror
13
14      call MPI_Scatter(send_buf, send_count,send_type, recv_buf,recv_count,
        recv_type,root,comm, ierror)
15
16  end program mpi_example
```

Listing 3.5: utilizzo di MPI_Scatter

Questa funzione ha gli stessi parametri di input della funzione MPI_Gather, ma la differenza sta nel fatto che il processo con rango root distribuisce il contenuto di send_buf a tutti i processi. Il contenuto di send_buf è spezzato in p segmenti, ognuno contenente send_count items.

La prossima funzione della libreria MPI che analizzeremo sarà :

```
1  program mpi_example
2      include 'mpif.h'
3
4      <type> send_buf(*)
5      <type> recv_buf(*)
6      integer send_count
7      integer send_type
8      integer recv_count
9      integer recv_type
10     integer root
11     integer comm
12     integer ierror
13
14     call MPI_AllGather(send_buf, send_count,send_type, recv_buf,recv_count
       ,recv_type,root,comm, ierror)
15
16 end program mpi_example
```

Listing 3.6: utilizzo di MPI_AllGather

Il compito di questa funzione, con gli stessi parametri di input dell'ultima funzione analizzata, è di comportarsi come una sequenza di p chiamate alla funzione MPI_Gather, ma la vera peculiarità di questa funzione sta nel fatto che ogni processo si comporta come root.

Ultima funzione che analizzeremo in questo capitolo è la MPI_AllReduce, che a differenza di MPI_Reduce, il risultato dell'operazione di riduzione OP nel buffer Result di ogni singolo processo. Ecco come si implementa in FORTRAN :

```
1  program mpi_example
2      include 'mpif.h'
3      <type> Operand(*)
4      <type> Result(*)
5      integer count
6      integer datatype
7      integer op
8      integer comm
9      integer ierror
10
11     call MPI_AllReduce(Operand,Result,count,datatype,op,comm, ierror)
12
13 end program mpi_example
```

Listing 3.7: utilizzo di MPI_AllGather

Chapter 4

Raggruppare i Dati per la Comunicazione

Con i calcolatori moderni, anche inviare un semplice messaggio può risultare un'operazione molto costosa e complessa, per questo la performance della macchina migliorano quando il numero di messaggi inviati è di trascurabile quantità. Grazie a MPI possiamo raggruppare i messaggi, sfruttando 3 possibili modalità differenti offerte da "mpif.h", in un singolo messaggio:

- Parametro Count
- Datatypes derivati
- `MPI_Pack` e `MPI_Unpack`

Esamineremo ogni singola possibilità che possiamo sfruttare per migliorare le prestazioni.

4.1 Il Parametro Count

Grazie a FORTRAN, abbiamo la garanzia che tutti gli elementi di un array vengono memorizzati in locazioni contigue di memoria, proprio per questo se volessimo inviare tutti gli elementi di un array, possiamo farlo utilizzando un singolo messaggio. Di seguito un esempio, dove supponiamo di voler inviare la metà dell'array :

```
1  program mpi_example
2      include 'mpif.h'
3
4      real vector(100)
5      integer p
6      integer my_rank
7      integer ierr
8      integer i
9      integer status(MPI_STATUS_SIZE)
10
11     tag=47
12     count=50
13     dest=1
14     call MPI_SEND(vector(51),count,MPI_REAL,dest,tag,MPI_COMM_WORLD,ierr)
15
16 end program mpi_example
```

Listing 4.1: invio di alcuni elementi di un array

4.2 Tipi Derivati e `MPI_TYPE_STRUCT`

Nella programmazione con MPI, uno dei problemi principali riguarda l'uso di tipi di dati definiti dall'utente nei linguaggi come Fortran 90. MPI è stato progettato senza considerare direttamente i tipi derivati, il che può creare difficoltà nell'uso di alcune funzioni di comunicazione. Fortran 90 consente la definizione di tipi derivati. Tuttavia, il sistema di tipi di MPI, che utilizza `'MPI_Datatype'`, non riconosce automaticamente queste strutture personalizzate.

Se si prova a trasmettere un'istanza di `'INDATA_TYPE'` utilizzando `'MPI_Bcast'` con un `'MPI_Datatype'` predefinito, la chiamata fallisce. Questo accade perché MPI non è in grado di interpretare correttamente la disposizione in memoria degli elementi contenuti nei tipi derivati.

Anche se per il programmatore la struttura di un tipo derivato può sembrare lineare e coerente, in realtà la disposizione fisica in memoria potrebbe non essere compatibile con le aspettative di MPI. Questo è dovuto a eventuali spazi o allineamenti che il compilatore può inserire tra i campi di un tipo derivato per ottimizzare l'accesso alla memoria.

Per superare questa limitazione, MPI offre un meccanismo per definire manualmente il layout di un tipo derivato utilizzando `'MPI_Type_struct'`. Questo permette di specificare esplicitamente:

1. Il numero di elementi nel tipo derivato.
2. La loro posizione in memoria.
3. Il tipo MPI corrispondente a ciascun campo.

Di seguito il codice per avere un esempio pratico :

```
1 program mpi_example
2     include 'mpif.h'
3
4     integer count, array_of_block_lengths(*),array_of_displacements(*),
      array_of_types(*)
5     integer newtype, ierror
6
7     call MPI_TYPE_STRUCT(count,array_of_block_lengths,
      array_of_displacement,array_of_types,newtype,ierror)
8
9 end program mpi_example
```

Listing 4.2: Utilizzo di `MPI_TYPE_STRUCT`

Il parametro Count è il numero di elementi nel tipo derivato, inoltre corrisponde anche alla dimensione dei tre array sopra dichiarati. L'array `array_of_block_lengths` contiene il numero di elementi in ogni elemento del tipo, quindi se un elemento del tipo è un array di `m` elementi, allora l'elemento corrispondente in `array_of_block_lengths` sarà `m`. Per quanto riguarda `array_of_displacements`, esso contiene i displacements di ogni elemento dall'inizio del messaggio, e l'array `array_of_types` contiene i Datatype MPI di ogni entry. Per finire, l'argomento **newtype** ritorna un puntatore al datatype creato dalla chiamata `MPI_TYPE_STRUCT`. Un'altra peculiarità di tale funzione è che può essere chiamata ricorsivamente per costruire più Datatype derivati complessi. Solo definendo correttamente questi aspetti, è possibile far comunicare strutture complesse tra processi MPI in modo sicuro ed efficiente.

4.3 Altri Costruttori per Datatype Derivati

Come abbiamo visto precedentemente, MPI_TYPE_STRUCT è il costruttore Datatype più gene rale in MPI, e l'utente deve provvedere a fornire una completa descrizione di ogni elemento del tipo. MPI fornisce tre costruttori di datatype derivati per poter trattare situazioni in cui gli elementi di un array che hanno lo stesso tipo. Iniziamo con il trattare MPI_TYPE_CONTIGUOUS, che costruisce un tipo derivato dove i suoi elementi sono contigui all'interno di un array. Di seguito l' implementazione :

```
1 program mpi_example
2     include 'mpif.h'
3
4     integer count, oldtype,newtype,ierror
5
6     call MPI_TYPE_CONTIGUOUS(count,oldtype,newtype,ierror)
7
8 end program mpi_example
```

Listing 4.3: Utilizzo di MPI_TYPE_CONTIGUOUS

Questa funzione crea un datatype derivato che consiste di **count** elementi di tipo **oldtype**, dove gli elementi appartengono a locazioni contigue di memoria. Adesso trattiamo MPI_TYPE_VECTOR, che crea un tipo derivato che consiste di **count** elementi, dove tali elementi sono equidistanti all'interno di un array. Ti fornisco un esempio :

```
1 program mpi_example
2     include 'mpif.h'
3
4     integer count, block_length,stride,oldtype,element_type,newtype,ierror
5
6     call MPI_TYPE_VECTOR(count,block_length,stride,element_type,newtype,
      ierror)
7
8 end program mpi_example
```

Listing 4.4: Utilizzo di MPI_TYPE_VECTOR

Ogni elemento contiene **block_length** voci di tipo **element_type**. Invece, per stride si intende il numero di elementi di tipo element_type tra gli elementi successivi di tipo new_type.

Per quanto riguarda l'ultimo costruttore, MPI_TYPE_INDEXED crea un tipo derivato costituito da count elementi. Il i-esimo elemento consiste di array_of_block_lengths[i] voci di tipo element_type, ed è spostato di array_of_displacements[i] unità di tipo element_type dall'inizio del nuovo tipo newtype.

```
1 program mpi_example
2     include 'mpif.h'
3
4     integer count,array_of_block_lengths(*),array_of_displacements,
      element_type
5     integer newtype,ierror
6
7     call MPI_TYPE_INDEXED(count,array_of_block_lengths,
      array_of_displacements,element_type,newtype,ierror)
8
9 end program mpi_example
```

Listing 4.5: Utilizzo di MPI_TYPE_INDEXED

4.4 Pack / Unpack

MPI fornisce un approccio alternativo per raggruppare dati attraverso le funzioni `MPI_PACK` e `MPI_UNPACK`, dove la prima funzione permette di immagazzinare dati non contigui in locazioni di memoria contigue, mentre la seconda funzione può essere usata per copiare dati da un buffer contiguo in locazioni di memoria non contigue. La sintassi di `MPI_PACK` è :

```
program mpi_example
    include 'mpif.h'

    <type> pack_data(*), buffer(*)
    integer in_count, datatype,size,position_ptr,comm,ierror

    call MPI_PACK(pack_data,in_count,datatype,buffer,size,position_ptr,
    comm,ierror)

end program mpi_example
```

Listing 4.6: Utilizzo di `MPI_PACK`

Il parametro `pack_data` fa riferimento ai dati da memorizzare nel buffer. Dovrebbe consistere di `in_count` elementi, ciascuno di tipo datatype. Il parametro `position_ptr` è un parametro di tipo in/out. In ingresso, i dati a cui fa riferimento `pack_data` vengono copiati nella memoria a partire dall'indirizzo buffer + `position_ptr`¨. Al ritorno, `position_ptr` fa riferimento alla prima posizione nel buffer dopo i dati copiati. Il parametro size contiene la dimensione in byte della memoria a cui fa riferimento buffer, e comm è il comunicatore che utilizzerà il buffer. Invece, la sintassi di `MPI_UNPACK` è la seguente :

```
program mpi_example
    include 'mpif.h'

    <type> buffer(*), outbuf(*)
    integer size, position, count, datatype, comm, ierror
    integer position_ptr

    call MPI_UNPACK(buffer, size, position_ptr, unpack_data, count,
    datatype, comm, ierror)

end program mpi_example
```

Listing 4.7: Utilizzo di `MPI_UNPACK`

Il parametro buffer fa riferimento ai dati da decomprimere. Consiste di size byte. Il parametro `position_ptr` è ancora una volta un parametro di tipo in/out. Quando viene chiamata `MPI_Unpack`, i dati a partire dall'indirizzo buffer + `position_ptr` vengono copiati nella memoria a cui fa riferimento `unpack_data`. Al ritorno, `position_ptr` fa riferimento alla prima posizione nel buffer dopo i dati appena copiati. `MPI_Unpack` copierà count elementi di tipo datatype in `unpack_data`. Il comunicatore associato a buffer è comm.

4.5 Come Decidere Quale Metodo Usare

Come giusto che sia, chiunque alle prime armi può porsi una domanda semplice, ma non banale; qual è la funzione migliore da usare quando vogliamo raggruppare un insieme di dati da inviare?.

Se i dati da inviare sono memorizzati in voci consecutive di un array, allora si dovrebbero semplicemente utilizzare gli argomenti count e datatype nelle funzioni di comunicazione. Questo approccio non comporta alcun overhead aggiuntivo sotto forma di chiamate a funzioni di creazione di tipi derivati o chiamate a `MPI_Pack/MPI_Unpack`.

Se c'è un gran numero di elementi che non si trovano in posizioni di memoria contigue, allora costruire un tipo derivato probabilmente comporterà meno overhead rispetto a un gran numero di chiamate a `MPI_Pack/MPI_Unpack`.

Se i dati hanno tutti lo stesso tipo e sono memorizzati a intervalli regolari in memoria (ad esempio, una colonna di una matrice), sarà quasi certamente molto più facile e veloce utilizzare un tipo derivato piuttosto che utilizzare `MPI_Pack/MPI_Unpack`. Inoltre, se i dati hanno tutti lo stesso tipo, ma sono memorizzati in posizioni irregolarmente distanziate in memoria, sarà probabilmente più facile ed efficiente creare un tipo derivato utilizzando `MPI_Type_INDEXED`. Ci sono anche un paio di situazioni in cui l'uso di `MPI_Pack` e `MPI_Unpack` è preferito. Bisogna notare innanzitutto che si potrebbe evitare l'uso del buffering di sistema con pack, poiché i dati sono esplicitamente memorizzati in un buffer definito dall'utente. Il sistema può sfruttarlo notando che il tipo di dato del messaggio è `MPI_PACKED`. Inoltre, nota che l'utente può inviare messaggi 'a lunghezza variabile' impacchettando il numero di elementi all'inizio del buffer.

Chapter 5

Esercizi Pratici

5.1 Hello World con MPI

Iniziamo introducendo uno degli esercizi più semplici da svolgere, e li raggrupperemo e qualificheremo in ordine di difficoltà, in modo da avere un approccio più morbido.

```
1  program hello_mpi
2    use mpi
3    implicit none
4
5    integer :: ierr, rank, size
6
7    call MPI_Init(ierr)
8    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
9    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
10
11   print *, 'Hello from process', rank, 'of', size
12
13   call MPI_Finalize(ierr)
14 end program hello_mpi
```

Listing 5.1: Primo Esercizio

L'output che ci attende è il seguente :
Hello from process 0 of 4
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 3 of 4

In questo semplice programma facciamo stampare ad ogni processo la scritta "Hello from process * ", ma bisogna considerare che l 'ordine di stampa non è garantito poichè i processi lavorano in parallelo.

5.2 Scambio di Messaggi

In questo secondo esercizio, l'obiettivo è quello di far scambiare un messaggio tra due processi, in modo da comprendere la comunicazione punto a punto.

```
1  program send_receive_example
2    use mpi
3    implicit none
4
5    integer :: ierr, rank, size
6    integer :: msg
7
8    call MPI_Init(ierr)
9    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
10   call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
11
12   if (size < 2) then
13     print *, 'Questo programma richiede almeno 2 processi.'
14     call MPI_Finalize(ierr)
15     stop
16   end if
17
18   if (rank == 0) then
19     msg = 42
20     call MPI_Send(msg, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
21     print *, 'Processo 0 ha inviato il messaggio:', msg
22   else if (rank == 1) then
23     call MPI_Recv(msg, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE, ierr)
24     print *, 'Processo 1 ha ricevuto il messaggio:', msg
25   end if
26
27   call MPI_Finalize(ierr)
28 end program send_receive_example
```

Listing 5.2: Secondo Esercizio

L'output che ci aspettiamo è illustrato di seguito :

Processo 0 ha inviato il messaggio: 42
Processo 1 ha ricevuto il messaggio: 42

In questo caso, identifichiamo il processo che ha inviato il messaggio e anche il processo che l'ha ricevuto, e cosi facendo, siamo sicuri che lo scambio di messaggio sia avvenuto senza errori e con estrema semplicità.

5.3 Somma di array distribuita

Nel seguente programma facciamo si che ogni processo possiede un array di 5 elementi interi , ma il nostro compito principale è quello di usare `MPI_REDUCE` in modo da calcolare la somma di tutti gli elementi globali e finalizzarli nel processo 0.

```
program reduce_sum_example
  use mpi
  implicit none

  integer, parameter :: n = 5
  integer :: ierr, rank, size, i
  integer :: local_array(n), global_array(n)

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

  ! Inizializza array locale con valori dipendenti dal rank
  do i = 1, n
    local_array(i) = rank + i
  end do

  ! Esegue somma degli array in global_array sul processo 0
  call MPI_Reduce(local_array, global_array, n, MPI_INTEGER, MPI_SUM, 0,
    MPI_COMM_WORLD, ierr)

  ! Solo il processo 0 stampa il risultato
  if (rank == 0) then
    print *, 'Somma globale degli array:'
    do i = 1, n
      print *, 'Elemento', i, ':', global_array(i)
    end do
  end if

  call MPI_Finalize(ierr)
end program reduce_sum_example
```

Listing 5.3: Terzo Esercizio

In questo esercizio vediamo come raccogliere dati distribuiti tra processi in un'unica operazione collettiva, l'output atteso è :

Somma globale degli array:
Elemento 1 : 3
Elemento 2 : 6
Elemento 3 : 9
Elemento 4 : 12
Elemento 5 : 15

5.4 Broadcast di un Valore

Nel prossimo esercizio, il primo processo genera un numero intero e lo invia a tutti gli altri processi tramite la chiamata `MPI_BCAST`. Al termine ogni processo stampa il valore che ha ricevuto.

```
1     program broadcast_example
2   use mpi
3   implicit none
4
5   integer :: ierr, rank, size
6   integer :: value
7
8   call MPI_Init(ierr)
9   call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
10  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
11
12  ! Solo il processo 0 inizializza il valore
13  if (rank == 0) then
14    value = 99
15  end if
16
17  ! Broadcasting del valore a tutti i processi
18  call MPI_Bcast(value, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
19
20  print *, 'Processo', rank, 'ha ricevuto il valore:', value
21
22  call MPI_Finalize(ierr)
23 end program broadcast_example
```

Listing 5.4: Quarto Esercizio

Ecco l'output atteso :
Processo 0 ha ricevuto il valore: 99
Processo 1 ha ricevuto il valore: 99
Processo 2 ha ricevuto il valore: 99
Processo 3 ha ricevuto il valore: 99

Notiamo che in questo programma ogni processo ha ricevuto il valore che è stato inviato tramite la chiamata `MPI_BCAST`, mostrando come un processo possa condividere dati con tutti gli altri processi efficentemente.

5.5 Sincronizzazione con MPI_BARRIER

Riprendendo MPI_BARRIER, si ricordi che è una funzione utilizzata per sincronizzare tutti i processi nel Communicator. Questo esercizio è implementato in modo da misurare il tempo totale tra le due chiamate MPI_BARRIER e vedere come i processi si arrestano prima di proseguire.

```
1     program barrier_example
2   use mpi
3   implicit none
4
5   integer :: ierr, rank, size
6   double precision :: start_time, end_time
7   integer :: i
8
9   call MPI_Init(ierr)
10  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
11  call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
12
13  ! Simulazione: ogni processo fa un lavoro diverso (sleep fittizio con
      loop)
14  do i = 1, 10000000 * (rank + 1)
15    ! loop vuoto per simulare carico computazionale
16  end do
17
18  ! Sincronizzazione prima di misurare il tempo
19  call MPI_Barrier(MPI_COMM_WORLD)
20  start_time = MPI_Wtime()
21
22  ! Tutti i processi si sincronizzano qui
23  call MPI_Barrier(MPI_COMM_WORLD)
24  end_time = MPI_Wtime()
25
26  print *, 'Processo', rank, ' - Tempo dopo la barriera:', end_time -
      start_time, 'secondi'
27
28  call MPI_Finalize(ierr)
29 end program barrier_example
```

Listing 5.5: Quinto Esercizio

L'output, dopo la seconda barriera, mostrerà tempi molto simili. Ciò è dovuto al fatto che tutti i processi aspettano gli altri che raggiungano quel punto.

Processo 0 - Tempo dopo la barriera: 0.000010000 secondi
Processo 1 - Tempo dopo la barriera: 0.000009000 secondi
Processo 2 - Tempo dopo la barriera: 0.000009000 secondi
Processo 3 - Tempo dopo la barriera: 0.000009000 secondi

Utilizziamo la prima barriera per assicurarci che tutti i processi partano allo stesso momento, cosi da calcolare e misurare il tempo tra le due barriere, dove la seconda serve per sincronizzare il termine. L'uso di MPI_Wtime() mostra quanto duri effettivamente l'attesa, utile in ambienti reali per benchmarking o debugging.

Chapter 6

Guida alla Compilazione ed Esecuzione

Una volta scritto il codice, è importante testarlo e valutare le sue prestazioni. Ma come si compila ed esegue un programma in linguaggio `FORTRAN` che utilizza MPI? In questa sezione forniamo una breve guida su come farlo in ambiente Linux.

6.1 MPI su Distribuzioni Linux

Primo Passo: Installazione
Aggiorniamo il sistema e installiamo le librerie necessarie tramite terminale (per sistemi Debian/Ubuntu):

- `sudo apt update`
- `sudo apt install openmpi-bin libopenmpi-dev`

Nota: Su altre distribuzioni (es. Fedora, CentOS) i comandi possono variare, ad esempio:

- `sudo dnf install openmpi openmpi-devel`

Secondo Passo: Compilazione
Compiliamo il file utilizzando i wrapper forniti da MPI. I comandi cambiano in base al linguaggio:

- **Fortran**
  `mpif90 nomefile.f90 -o nomeprogramma`
- **C**
  `mpicc nomefile.c -o nomeprogramma`
- **C++**
  `mpic++ nomefile.cpp -o nomeprogramma`

Terzo Passo: Esecuzione
Per eseguire il programma con, ad esempio, 4 processi:

- `mpirun -np 4 ./nomeprogramma`

6.2 Errori Comuni e Soluzioni

- Verificare che il file sia eseguibile: `chmod +x nomeprogramma`
- In caso di errore "`command not found`", verificare che `mpif90` o `mpirun` siano inclusi nel `PATH`.
- Per salvare l'output su file: `mpirun -np 4 ./nomeprogramma > output.txt`

6.3 Altre Informazioni su MPI

- Esiste una FAQ su MPI disponibile tramite ftp anonimo presso:
  -Mississippi State University.
  L'indirizzo è ftp.erc.msstate.edu, e il file si trova in pub/mpi/faq.

- Ci sono anche numerosi siti web dedicati a MPI. Alcuni di questi sono:
  - [http://www.epm.ornl.gov/ walker/mpi]
  -(http://www.epm.ornl.gov/ walker/mpi).

- La pagina web di MPI del Oak Ridge National Lab.
  - [http://www.erc.msstate.edu/mpi]
  -(http://www.erc.msstate.edu/mpi).

- La pagina web di MPI della Mississippi State University. - [http://www.mcs.anl.gov/mpi]
  -(http://www.mcs.anl.gov/mpi).

  Ognuno di questi siti contiene una vasta quantità di informazioni su MPI. In particolare, la pagina della Mississippi State University contiene una bibliografia di articoli su MPI, mentre la pagina di Argonne contiene una raccolta di programmi di test MPI.

Lo Standard MPI Message Passing Interface Forum (1994) è attualmente disponibile su ciascuno dei siti sopra indicati. Questa è, naturalmente, la dichiarazione definitiva di cosa sia MPI. Quindi, se c'è un dubbio riguardo qualcosa, questa è la fonte finale di riferimento. Contiene anche numerosi esempi ben strutturati di utilizzo delle varie funzioni MPI, quindi è molto più di una semplice guida di riferimento.

Attualmente, diversi membri del Forum MPI stanno lavorando su una versione annotata dello standard MPI Otto et al. (1998).

Il libro Gropp et al. (1994) è un'introduzione tutorial a MPI. Fornisce numerosi esempi completi di programmi MPI.

Il libro Pacheco (1997) contiene un'introduzione tutorial a MPI (su cui questa guida si basa). Contiene anche un'introduzione più generale al parallelismo e alla programmazione su macchine di passaggio messaggi.

Referenze

Gropp, W., Lusk, E. and Skjellum, A. (1994), *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

Message Passing Interface Forum (1994), 'Mpi: A message-passing interface standard', *International Journal of Supercomputer Applications* **8**(3–4). Also available as Technical Report CS-94-230, University of Tennessee, Knoxville.

Otto, S. et al. (1998), *MPI Annotated Reference Manual*, MIT Press, Cambridge, MA. To appear.

Pacheco, P. S. (1995), *A User's Guide to MPI*, University of San Francisco. University of San Francisco.

Pacheco, P. S. (1997), *Programming Parallel with MPI*, Morgan Kaufmann, San Francisco, CA.

Pacheco, P. S. and Ming, W. C. (1995), *MPI User Guide in FORTRAN*, University of San Francisco. Revised 1997.