MPI:
MESSAGE PASSING INTERFACE

Lorenzo Simionato
lorenzo@simionato.org

Dicembre 2008

Questo documento vuol essere una brevissima introduzione ad MPI. Per un approfondimento dettagliato, si rimanda invece alla documentazione ufficiale o ad un libro sull'argomento.
Il documento è stato redatto nell'ambito di una presentazione per il corso di “Sistemi Distribuiti”, all'università Ca' Foscari di Venezia.
Nel testo non si escludono errori od inesattezze, pertanto eventuali segnalazioni (errori, suggerimenti, domande) possono essere inviate a lorenzo@simionato.org.

Lorenzo Simionato
03/12/2008

Indice

Introduzione

MPI è l'acronimo di Message Passing Interface. Si tratta di una libreria per lo scambio di messaggi tra diversi processi. Per essere più precisi MPI è solamente la specifica di una interfaccia, non una implementazione. Esistono invece numerose implementazioni di MPI, alcune fra le più note sono Open MPI[1] ed MPICH[2]. L'utilizzo principale di MPI è nel campo della computazione parallela e distribuita (parallel computing & distributed computing).

Per realizzare una elaborazione in maniera distribuita (utilizzando quindi più serventi), è in generale necessaria una comunicazione tra i vari processi. Esistono sostanzialmente due tecniche per effettuarla: shared-memory e message passing.

Nella tecnica shared-memory, il programma non si deve preoccupare dell'esistenza degli altri processi (collocati anche su altri serventi) e può accedere ad una memoria che è idealmente condivisa tra tutti i processi. Naturalmente la componente che realizza l'architettura di shared-memory comunicherà tramite messaggi per realizzare questa illusione.

Nella tecnica di message-passing, invece, il programma deve farsi carico della comunicazione tra i vari processi e non c'è un concetto di memoria condivisa, ma ogni processo può accedere unicamente alla propria memoria privata.

Esistono vantaggi e svantaggi dei due approcci, tuttavia l'analisi di questo punto non è lo scopo di questo documento.

MPI è lo standard de-facto per il message-passing e nelle proprie specifiche, prevede la definizione di una serie di primitive per la comunicazione tra processi. Si tratta sostanzialmente di uno stato middelware, che si colloca tra il linguaggio di programmazione ed il sistema operativo, consentendo appunto lo scambio di messaggi.

Il progetto MPI inizia nel 1992, con lo scopo di realizzare un sistema di message passing formato dalle migliori funzionalità presenti negli altri sistemi dell'epoca. Si voleva inoltre creare un'interfaccia standard, focalizzando gli sforzi su un'unica soluzione, inoltre un requisito importante doveva essere la portabilità. Nel processo di standardizzazione furono coinvolte moltissime università ed aziende, per un totale di 40 organizzazioni e 60 persone. Si costituì inoltre un working group e l'MPI Forum[3].

Un primo draft dello standard si ebbe già per la fine del 1992, seguito poi da varie correzioni e chiarimenti, per arrivare ad MPI 1.0, nel 1994. Nel 1995 seguì MPI 1.1, che conteneva unicamente lievi modifiche rispetto alla versione 1.0. Seguì poi MPI 1.3 ma soprattutto, MPI 2 nel 1997 che introduceva novità significative. Furono introdotte infatti diverse nuove funzionalità e possibilità (si veda il capitolo “Altre Funzionalità”).

Seguirono poi ulteriori versioni ed errate, fino ad arrivare alla versione corrente (MPI 2.1) rilasciata nel Giugno 2008.

Le specifiche di MPI descrivono una serie di funzioni, subroutine e metodi per realizzare la comunicazione (e molte altre cose). La descrizione avviene tramite un linguaggio astratto, tuttavia per C, C++ e Fortran vengono descritti con precisione i prototipi dei metodi, delle funzioni, ecc. In questo modo, per questi linguaggi, tutte le funzioni di MPI saranno consistenti per tutte le implementazioni. Per altri linguaggi, come ad esempio Java, non sono invece stati definiti con precisione questi aspetti. Infatti, sebbene esistano delle implementazioni di MPI anche per altri linguaggi, si sarà più legati alla specifica implementazione della libreria che si utilizza.

Concetti di base

Prima di presentare le funzioni MPI più importanti per lo scambio di messaggi, è utile fare una panoramica dei concetti base di MPI.

Consideriamo un sistema distribuito costituito da una serie di nodi, connessi tramite una rete di comunicazione. Su ogni nodo sono presenti uno o più processi, che vengono eseguiti in maniera concorrente. Ciascun processo può essere considerato come un programma e, sebbene nelle specifiche di MPI non sia richiesto, supponiamo che il programma sia sempre lo stesso (figura 1).

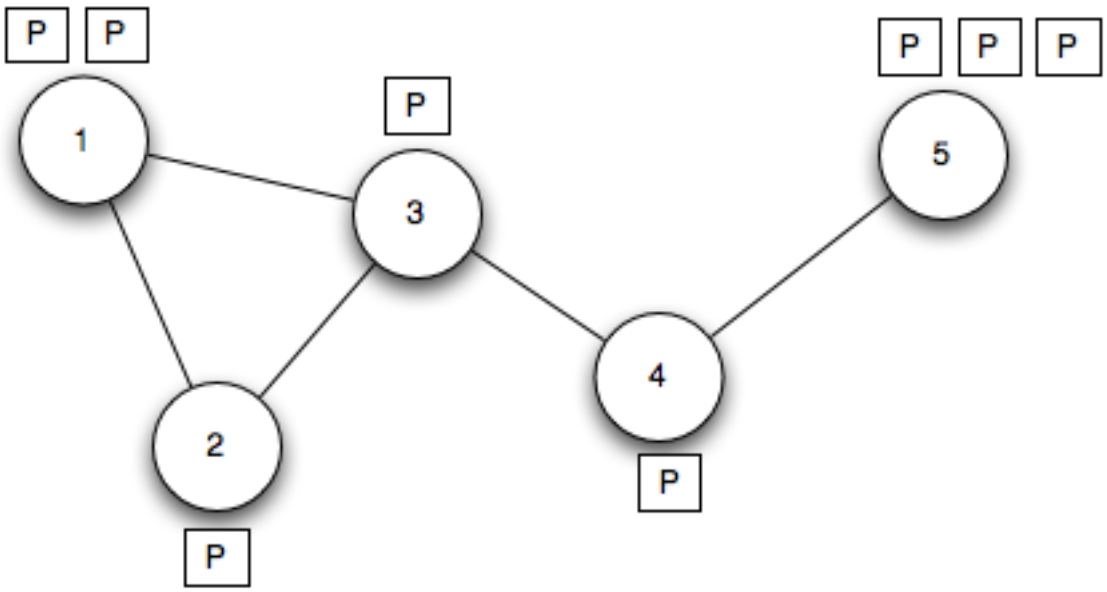

Figura 1: Esempio di scenario per l'esecuzione un programma con MPI

MPI consente la comunicazione tra i diversi processi. Deve quindi esserci innanzitutto un modo per identificare ciascuno processo. Bisogna quindi avere una sorta di etichettamento, che nello specifico viene chiamato **ranking**,

In MPI, ciascun processo che fa parte di un gruppo (per semplicità considereremo un unico gruppo, formato da tutti i processi) ha associato un particolare ranking. Il ranking è un numero naturale, che va da 0 ad N-1, con N numero di processi del gruppo.

In tantissime funzioni MPI, il ranking sarà di fondamentale importanza per specificare mittenti e destinatari dei messaggi, distinguere il ruolo delle entità ed altro.

Consideriamo ora un semplice programma MPI, per chiarire meglio i concetti finora esposti. Per questo esempio, e per tutti quelli che seguiranno, utilizzeremo in linguaggio C. Tuttavia la scelta del linguaggio non riveste un ruolo importante in questo contesto.

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

int main (int argc, char **argv) {
  int myrank;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  printf("Il mio ranking e' : %d\n", myrank);

  MPI_Finalize();
  return 0;
}
```

Come si può notare, è necessario inizializzare l'ambiente tramite la funzione MPI_Init e terminare tramite la funzione MPI_Finalize. Per quanto riguarda invece il ranking:

MPI_Comm_rank (Comunicator, int myrank)*

Colloca nella variabile myrank il valore numerico del rank del processo che esegue. Il comunicator è un concetto più articolato, che non tratteremo. Basti sapere che la costante MPI_COMM_WORLD può essere ipotizzata come l'insieme di tutti i i processi.

Se eseguiamo dunque il programma specificato sopra, otterremo un output di questo tipo:

```
Il mio ranking e' : 0
Il mio ranking e' : 1
Il mio ranking e' : 2
Il mio ranking e' : 3
...
```

Naturalmente il funzionamento è abbastanza semplice se tutti i processi si trovano sullo stesso nodo ma, come abbiamo già introdotto, in generale si possono trovare in nodi diversi. Sarà dunque evidente che la libreria MPI in uso effettua delle comunicazioni tra i vari nodi per stabile un etichettamento di questo tipo. Come ciò avvenga tuttavia non fa parte della specifica di MPI, ma è lasciato all'implementazione della singola libreria. Su questo punto torneremo nell'ultimo capitolo.

Comunicazione punto-punto

Una delle funzioni più importanti tra quelle messe a disposizione da MPI, è la comunicazione punto-punto. In maniera più specifica l'invio e la ricezione di un messaggio tra due processi diversi.

Per compiere questa operazione, vengono fornite le seguenti funzioni (utilizziamo ancora le primitive del linguaggio C):

MPI_Send(buf, count, datatype, dest, tag, comm)

MPI_Recv(buf, count, datatype, source, tag, comm, status)

Per quanto riguarda la funzione send:

- *buf* è un puntatore ad un'area di memoria contente il valore da spedire
- *count* è il numero di elementi da spedire
- *datatype* il tipo di dato da spedire
- *dest* è il ranking del destinatario
- *tag* è l'identificatore del messaggio
- *comm* è il communicator da utilizzare

Per la funzione recv invece:

- *buf* è un puntatore ad un'area di memoria dove salvare il valore ricevuto
- *count* è il numero di elementi da salvare
- *datatype* il tipo di dato che si riceve
- *source* è il ranking del mittente
- *tag* è l'identificatore del messaggio
- *comm* è il communicator da utilizzare
- *status* è un puntatore ad una variabile dove viene salvato lo stato

Esistono molti dettagli relativi a queste due funzioni, per i quali si rimanda alle specifiche di MPI. Tuttavia è bene chiarire alcuni punti. In primo luogo appare un po' limitativo il fatto di dover indicare il sorgente, nella funzione MPI_Recv. Tuttavia è possibile specificare che si intende ricevere da un mittente qualsiasi. Anche per quanto riguarda la dimensione del messaggio, si possono utilizzare alcuni accorgimenti per evitare di specificarla nella funzione di ricezione.

Di importanza maggiore è la semantica di esecuzione di queste due funzioni. Innanzitutto si tratta di due funzioni **bloccanti**. Ovvero l'esecuzione del programma prosegue, solamente quando un messaggio è stato inviato o ricevuto.

Se per la funzione Recv il funzionamento è chiaro (quando termina si sono ricevuti dei dati), per la funzione Send la questione è più delicata. Sebbene infatti l'esecuzione sia bloccante, non è detto che il messaggio sia stato consegnato. Infatti in MPI si distinguono due modalità:

- Modalita buffered
- Modalità sincrona

Nella modalità buffered, il controllo ritorna al programma appena il dato da inviare è stato copiato in un buffer. Ciò quindi non significa assolutamente che sia stato già inviato e ricevuto. Nella modalità sincrona, invece, la funzione termine solamente quando una funzione corrispondente funzione Receive ha iniziato la ricezione.

Il funzionamento di default lascia libera implementazione alla specifica libreria, a seconda delle situazioni. Se si vuole invece forzare una particolare modalità, si possono usare le seguenti funzioni:

MPI_Bsend, MPI_Ssend

Rispettivamente per modalità Buffered e Sincrona (gli argomenti solo gli stessi della funzione Send). Esiste poi un'ulteriore modalità (Ready) che non verrà discussa.

Facciamo ora un esempio di comunicazione tra due processi:

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

int main (int argc, char **argv) {
  int myrank;
  char message[20];
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  printf("Il mio ranking e' : %d\n", myrank);

  if(myrank==0) {
    //Manda un messaggio al processo 1
    strcpy(message,"PROVA");
    MPI_Send(message,strlen(message)+1,MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    printf("%d) Ho inviato: '%s'\n", myrank,message);
  } else if(myrank==1) {
    //Riceve il messaggio
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("%d) Ho ricevuto: '%s'\n", myrank,message);
  }

  MPI_Finalize();
  return 0;
}
```

In questo caso, il processo con rank 0, invia una stringa al processo con rank 1 (che lo riceva). Un possibile (si possono naturalmente avere più interleaving) output con 5 processi è il seguente:

```
Il mio ranking e' : 3
Il mio ranking e' : 1
Il mio ranking e' : 2
Il mio ranking e' : 4
Il mio ranking e' : 0
0) Ho inviato: 'PROVA'
1) Ho ricevuto: 'PROVA'
```

Si supponga ora di eliminare la MPI_Recv(...) e di sostituire la MPI_Send con una MPI_Bsend. Quello che accade è che verrà comunque stampato il messagio “Ho inviato: PROVA”. Infatti come è stato visto, nel caso buffered la funzione ritorna una volta che ha copiato il messaggio nel buffer. Se invece utilizzassimo la funzione MPI_Ssend, il processo 0 resterebbe bloccato per sempre (sempre nel caso in cui rimuovessimo la recv).

MPI mette poi a disposizione delle chiamate **non bloccanti**. In questo caso le funzioni ritornano immediatamente, sebbene i dati non siano ancora stati inviati o ricevuti. Il vantaggio delle chiamate non bloccanti è l'incremento di performance. E' infatti possibile, durante lo svolgimento della funzione di invio/ricezione effettuare computazione. Visto lo scopo del presente documento, non descriveremo ulteriormente questo punto, per il quale si rimanda alla documentazione ufficiale.

E' naturale chiedersi a questo punto quale sia la semantica di comunicazione delle funzioni send/receive. In altre parole, cosa viene effettivamente inviato sui canali di comunicazione?

La risposta è che dipende dalla specifica implementazione di MPI, in quanto lo standard non definisce in dettaglio in funzionamento. Tuttavia ciascuna implementazione di MPI deve rispettare l'ordine dei messaggi. Ovvero se il mittente invia prima il messaggio 1 e poi il messaggio 2, allora il destinatario deve ricevere prima 1 e poi 2.

Nelle specifiche di MPI, viene tuttavia descritto un **possibile** protocollo per la comunicazione di tipo sincrono, vista prima. Il funzionamento è illustrato in figura 2.

In questo caso, una volta che il mittente ha invocato la funzione Send, viene inviato al destinatario una richiesta per l'invio del messaggio. Il destinatario invierà il permesso di inviare i dati, solamente se ha eseguito una chiamata alla funzione Receive. Quando il mittente ha ricevuto l'autorizzazione, inizia ad inviare i dati.

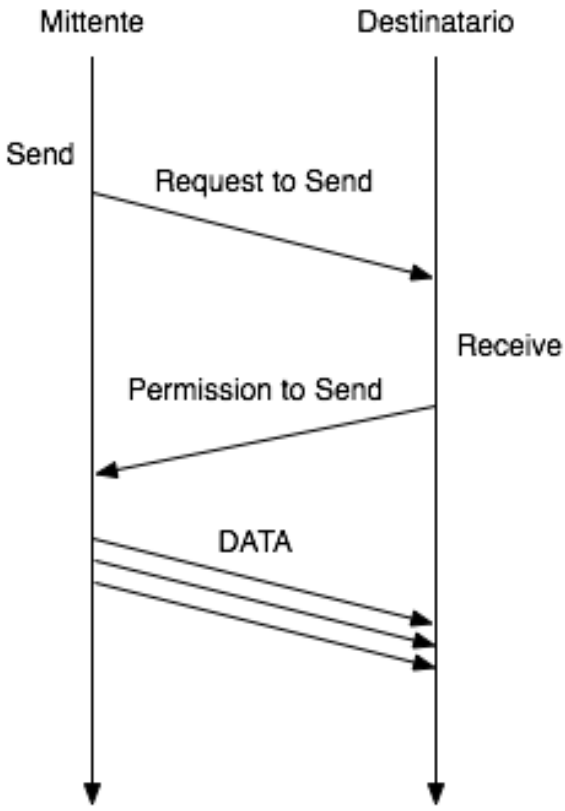

Figura 2: Possibile protocollo di comunicazione tramite Send/Receive

Comunicazione collettiva

E' stato mostrato come inviare dei dati ad un unico destinatario, tuttavia nell'ambito di una comunicazione parallela è spesso utile inviare un messaggio a tutti i membri del gruppo.

Spesso tuttavia quello che si vuole realizzare non è una semplice comunicazione broadcast, ma qualcosa di più articolato. In molte situazioni infatti si vogliono inviare dati diversi a seconda del destinatario. Ancora è utile che tutti i nodi inviino un messaggio (diverso) ad un nodo radice.

Per chiarire meglio questi punti, consideriamo un esempio. Si supponga di voler effettuare una computazione su una matrice. In particolare, si vorrà applicare una funzione su ogni elemento della matrice. Sarà quindi utile inviare (dal nodo root) una parte della matrice (e.g. N righe) ad ogni processo. Successivamente ogni processo effettuerà le computazioni necessarie e dovrà rispedire i dati elaborati alla radice.

MPI mette a disposizione delle funzioni per rispondere a tutte queste esigenze. Verranno ora illustrate le funzioni principali, tuttavia ne esistono molte altre (per le quali si rimanda alla documentazione ufficiale).

Broadcast

Per iniziare, vediamo come effettuare una semplice comunicazione broadcast. In questo caso un nodo vuole inviare un messaggio a tutti gli altri. Il messaggio è lo stesso per tutti i nodi. La funzione fornita da MPI è la seguente:

MPI_Bcast(buf,count,datatype,root,comm)

Molti parametri sono uguali alla funzione send. In dettagli abbiamo:

- *buf* è un puntatore ad un'area di memoria contente il valore da spedire
- *count* è il numero di elementi da spedire
- *datatype* il tipo di dato da spedire
- *root* è il ranking del mittente
- *comm* è il communicator da utilizzare

Si nota subito la presenza del parametro root (il mittente se vogliamo). Ciò è necessario in quanto la funzione Bcast va eseguita su tutti i nodi, non c'è una funzione receive.

Un esempio di codice dunque, che sarà lo stesso per tutti i nodi, può essere il seguente:

```
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, MPI_COMM_WORLD);
```

Naturalmente si sono omesse tutte le funzioni di inizializzazione, ecc.

E' importante notare che la comunicazione inizia, solamente quando tutti i processi hanno eseguito MPI_Bcast. In questo modo si evita di inviare dei dati solo ad alcuni processi (non si considera il caso di perdita di messaggi/crash dei processi).

Scatter

La funzione scatter è analoga a quella broadcast, tuttavia in questo caso c'è una importante differenza: il messaggio inviato non è lo stesso. Quello che si vuole fare è infatti inviare un messaggio diverso per ogni nodo. Il prototipo della funzione è:

MPI_Scatter(sendbuf,sendcount, sendtype,recvbuf,recvcount,recvtype,root,comm)

In dettaglio:

- *sendbuf* è un puntatore ad un'area di memoria contente il valore da spedire
- *sendcount* è il numero di elementi da spedire ad ogni processo
- *sendtype* il tipo di dato da spedire
- *recvbuf* è un puntatore all'area di memoria dove salvare i dati
- *recvcount* è il numero di elementi da salvare

- *recvtype* il tipo di dato da ricevere
- *root* è il ranking del mittente
- *comm* è il communicator da utilizzare

La funzione è un po' diversa rispetto alle precedenti, in quanto i messaggi inviati sono diversi. Come la funzione Bcast, la stessa riga va eseguita su tutti i processi.

La semantica di esecuzione è invece differente. Ovvero è equivalente ad eseguire n (con n numero di processi) operazioni Send, ciascuna ad un destinatario diverso. Inoltre la posizione iniziale del buffer avanza ad ogni invio di *sendcount* elementi. Per i riceventi, l'operazione è invece analoga a Recv.

Ad esempio, eseguendo il seguente codice:

```
int array[100];
int array2[10];
int root=0;
...
MPI_Scatter(array, 10, MPI_INT, array2, 10, MPI_INT, root, MPI_COMM_WORLD);
```

Vengono inviati 10 numeri ad ogni processo. Nello specifico il processo 0 riceverà:

array[0],arry[1],....array[9]

Il processo 1:

array[10],arry[11],....array[19]

Il processo k:

array[k*10],arry[k*10+1],....array[k*10+9]

In questo caso quindi il codice funziona correttamente solo 10 processi! Bisogna quindi porre molta attenzione all'uso di questa funzione, per evitare di inviare dati inconsistenti. Per usarla correttamente è dunque necessario **conoscere il numero di processi** (una funzione MPI si occupa di questo: MPI_Comm_size).

In molte situazioni è un po' limitativo inviare la stessa quantità di dati ad ogni processo. Tramite la funzione MPI_ScatterV (e l'opposta MPI_GatherV) è possibile inviare un numero differente di elementi per ogni processo. Tuttavia, per queste funzioni si rimanda al manuale.

Gather

La funzione Gather, compie l'operazione inversa della funzione Scatter.

In questo caso infatti tutti i processi inviano un risultato alla root. Il prototipo è:

MPI_Gather(sendbuf,sendcount, sendtype,recvbuf,recvcount,recvtype,root,comm)

Il significato dei parametri è del tutto analogo al precedente, avendo cura di scambiare le entità mittente e destinatario. In questo caso la root memorizza i dati in base all'ordine di

ranking. In questo modo si ha la stessa suddivisione della funzione Scatter. Ciò consente di eseguire con più facilità l'invio e la successiva ricezione dei dati.

Esempio

E' utile a questo punto fornire un piccolo esempio di Scatter e Gather, per chiarirne le potenzialità. Supponiamo di voler realizzare delle computazioni su ogni riga di una matrice. Un possibile esempio potrebbe essere sottrarre ad ogni elemento la media dei valori della corrispondente riga.

Per risolvere questo problema, possiamo dividere la matrice orizzontalmente in n parti, ed inviarne ciascuna ad un processo. Terminata la computazione, ogni processo invierà alla radice la riga con i dati elaborati. Uno schema della comunicazione è riportato in figura 3 (i processi P1 e P3 hanno già terminato e inviato i dati, P2 e P4 non ancora..

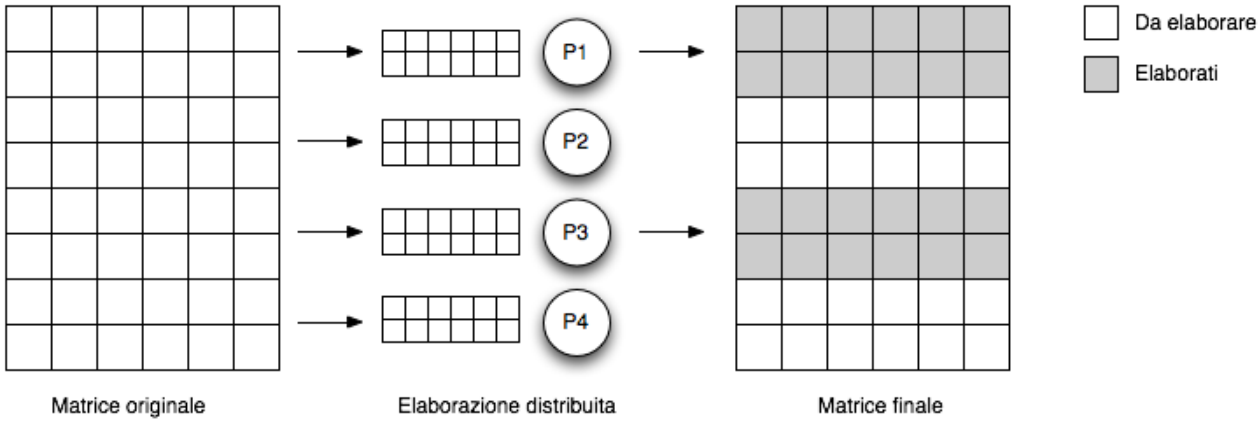

Figura 3: Esempio di elaborazione per il calcolo su matrice

A questo punto, per compiere la fase di distribuzione possiamo usare MPI_SCATTER, poi effettuare la computazione ed infine riottenere i dati con MPI_GATHER.

Segue un esempio di codice (non è un codice completo). Si noti che la costante N deve corrispondere al numero di processi (in questo esempio viene inviata una riga ad ogni processo). Per un esempio più realistico, si dovrebbero usare le funzioni MPI_SCATTERV e MPI_GATHERV.

```
  int matrix[N][N];
  int row[N];
  int myrank;
  int root=0;
  int i,j;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  MPI_Status status;

  if(myrank==root)
      //***Inizializza la matrice***

  //Invio dei dati
  MPI_Scatter(matrix,N,MPI_INT,row,N,MPI_INT,root,MPI_COMM_WORLD);

  //***Esegui una elaborazione sulla variabile row***

  //Invio risposta a root
  MPI_Gather(row,N,MPI_INT,matrix,N,MPI_INT,root,MPI_COMM_WORLD);

  //Root ha ora la matrice elaborata

  MPI_Finalize();
  return 0;
}
```

Tipi di dato

Come è stato visto finora, per la comunicazione è necessario specificare un tipo di dato. Negli esempi fatti finora, abbiamo utilizzato sempre MPI_INT. Approfondiamo ora brevemente la questione relativa ai tipi.

MPI fornisce alcuni tipo di base. Per citarne alcuni abbiamo: MPI_CHAR per caratteri, MPI_INT per interi, MPI_FLOAT e MPI_DOUBLE per i numeri con virgola singola e doppia precisione. Naturalmente i tipi base non sono sempre sufficienti, ed è quindi utile prevedere dei costruttori di nuovi tipi.

Uno dei costruttori più semplice è:

MPI_Type_contiguous(int count, oldtype,newtype)

Dove:

- *count* è il numero di volte in cui replicare il tipo
- *oldtype* è il tipo da replicare

- *newtype* è il nuovo tipo.

Questo costruttore consente in sostanza di inviare una sorta di array. Una volta costruito il tipo, per poterlo utilizzare, è necessario “committarlo”. Ciò viene fatto tramite la chiamata ad un'altra funzione MPI. Vediamo direttamente un esempio:

```
MPI_Datatype intArray;
MPI_Type_contiguous(10, MPI_INT, &intArray);
MPI_Type_commit(&intArray);
```

In questo caso si crea un nuovo tipo, formato da 10 interi in sequenza.

Se si ripensa alla funzione MPI_Send, si scopre in realtà che si sta utilizzando in maniera automatica proprio un costruttore di tipo continuo. Infatti, come è stato descritto, la funzione invia una serie di elementi (specificati dal parametro count). Quello che succede in realtà è proprio la creazione di un tipo come quello specificato nell'esempio.

Questo costruttore di tipo è abbastanza limitativo, in quanto non ci consente di inviare dati di tipo diverso in una volta sola. A questo proposito MPI fornisce tantissime funzioni per costruire tipi più complessi. Descriveremo ora un altro costruttore più potente di quello appena visto, per gli altri costruttori invece si rimanda invece al manuale.

MPI_Type_create_struct(count,blocklengths,displacements,types[],newtype)

Dove:

- *count* è il numero di blocchi (elementi) che compongono il tipo
- *blocklengths* è un array contenente, per ogni blocco, la lunghezza
- *displacements* è un array contenente, per ogni blocco, la posizione in cui inserirlo
- *types* è un array contenente i tipi dei vari dati che compongono la struttura
- *newtype* è un puntatore al nuovo tipo che si vuole creare

Questo costruttore ci consente di creare un tipo analogo ad una struttura (**struct**), del linguaggio C. Si possono infatti specificare tutti i tipo che compongono la struttura, con le relative lunghezze. Il campo displacements, invece, serve a specificare la posizione di inizio di ogni blocco. Sebbene possa essere dedotta utilizzando l'array blocklengrhs, displacement consente un controllo maggiore.

Può essere interessante sapere come avvenga effettivamente il passaggio dei dati tramite un'operazione Send/Receive. In sostanza che tipo marshalling si effettua? Che codifica hanno i dati quando passano nel canale?

Ancora una volta MPI lascia spazio all'implementazione e non definisce in dettaglio questi punti. Ciò che viene detto è che, in un ambiente omogeneo, i dati possono essere inviati senza alcuna conversione. Se l'ambiente è invece eterogeneo (e.g. macchine little-endian e big-endian) allora si possono avere varie soluzioni:

- Il destinatario converte, una volta ricevuto il messaggio, dalla rappresentazione del mittente alla propria
- Il mittente converte dalla propria rappresentazione a quella del destinatario, prima

di inviare il messaggio

- Si utilizza una codifica indipendente, come XDR (eXternal Data Representation)

Topologie virtuali

Un interessante funzione messa a disposizione da MPI, riguarda le topologie virtuali. Come è stato descritto, tutte le funzioni di comunicazione (punto-punto o collettive) si riferiscono ad un gruppo di processi. Nello specifico, abbiamo sempre utilizzato il gruppo MPI_COMM_WORLD che comprende tutti i i processi. Per essere più precisi, si parla di **communicator**, una struttura definita da MPI che contiene altre informazioni rispetto al semplice gruppo. E' stato poi visto che MPI assegna un rank da 0 a n-1, ad ogni processo che appartiene ad un communicator di grandezza n.

MPI consente di assegnare una topologia ad un communicator (ecco perché, ad esempio, è qualcosa di più articolato di un gruppo). Si tratta di una topologia virtuale, che è ben distinta da una topologia reale.

Con topologia reale infatti si intende come i vari nodi sono collegati fisicamente, mentre con topologia virtuale, si definisce unicamente un particolare assegnamento di etichette ai vari nodi. In sostanza, assegnare una topologia virtuale in MPI, implica unicamente un diverso assegnamento del ranking ai vari processi.

Un meccanismo di questo tipo può essere utile in vari casi. In particolare se si ha una topologia reale particolare, può essere utile assegnare il ranking in maniera da riflettere questa topologia. Ciò può essere utile per incrementare le performance. Si supponga, ad esempio, che i nodi siano collegati tramite una rete Token-ring (ad anello). Se si costruire una topologia virtuale adeguata, allora ogni nodo comunicherà unicamente con il vicino, ottimizzando le prestazioni. Se infatti si assegnassero invece i rank casualmente, un messaggio potrebbe essere costretto a transitare per molti altri nodi prima di raggiungere la destinazione.

Al di la della questione prestazionale, una topologia virtuale rende di sicuro il codice più chiaro e leggibile. Se ad esempio si applica un algoritmo di elezione su un anello, sarà conveniente creare come topologia virtuale un anello, al di la dell'effettiva topologia fisica dei nodi.

MPI mette a disposizione due costruttori di topologie. Il primo costrutto, consente di creare strutture cartesiane, mentre il secondo topologie qualsiasi. Nello specifico nel secondo caso bisogna fornire la matrice di adiacenza del grafo che si vuole costruire. Ci occuperemo solamente di topologie cartesiane (non discuteremo il secondo costruttore). Tramite esse è infatti possibile costruire parecchie strutture molto utilizzate: mesh, anelli, tori, ecc. La funzione per creare un nuova topologia cartesiana è la seguente:

MPI_Cart_create(comm_old,ndims,dims,periods,reorder, comm_cart)

In dettaglio:

- *comm_old* è il communicator da cui partire
- *ndims* il numero di dimensioni della griglia
- *dims* è un array di interi, contenente il numero di processi per ogni dimensione
- *periods* è un array che specifica per ogni dimensione, se la topologia è periodica

- *reoder* specifica se si vogliono riordinare le etichette già assegnate
- *comm_cart* è un puntatore alla nuova topologia creata

Vediamo subito alcuni esempi. In particolare costruiamo delle topologie fra le più note: un anello, una mesh, un toro.

Per costruire un anello (Figura 5) con N nodi, si usano i seguenti parametri:

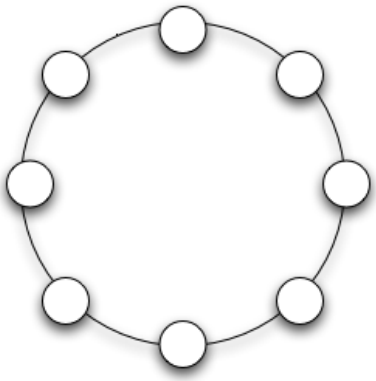

Figura 4: Anello

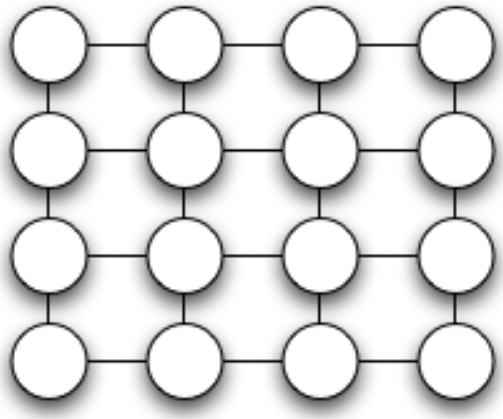

Figura 5: Una mesh

```
MPI_Comm ring;
int dims[]={N};
int periods[]={1};
MPI_Cart_create(MPI_COMM_WORLD,1,dims,periods,1,&ring);
```

Per costruire invece una mesh (Figura 4), con NxM nodi (bidimensionale):

```
MPI_Comm mesh;
int dims[]={N,M};
int periods[]={0,0};
MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,1,&mesh);
```

Per concludere, ecco il caso del toro MxNa due dimesioni (Figura 6):

```
MPI_Comm torus;
int dims[]={N,M};
int periods[]={1,1};
MPI_Cart_create(MPI_COMM_WORLD,2,dims,periods,1,&torus);
```

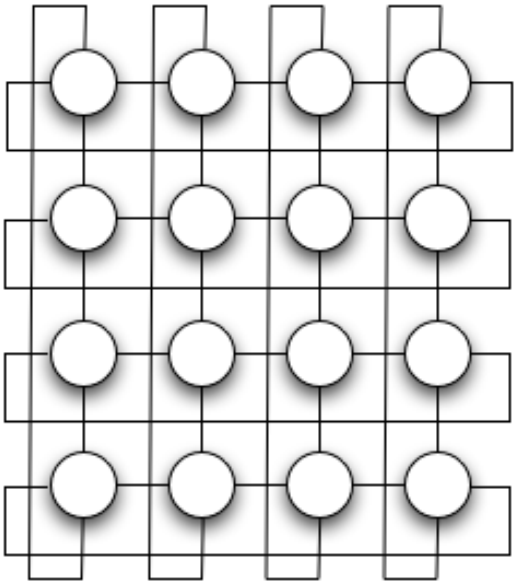

Figura 6: Un toro

A questo punto MPI fornisce delle funzioni per avere informazioni sulla topologia. Consente in sostanza di effettuare una conversione tra coordinate cartesiane e ranking.
In particolare è utile la funzione:

MPI_Cart_rank(comm,coords,rank)

Dove:

- *comm* è il communicator (con topologia cartesiana)
- *coords* è un array con le coordinate
- *rank* è un puntatore ad un intero, con il ranking voluto

Esiste anche una funzione inversa, per convertire da ranking a coordinate cartesiane. Molto utile invece la funzione si shifting:

MPI_Cart_shift(comm,direction,disp,rank_source,rank_dest)

Dove:

- *comm* è il communicator (con topologia cartesiana)
- *direction* è la dimensione da prendere in considerazione
- *disp* è lo “spazio” da lasciare
- *rank_source* è un puntatore ad un intero in cui inserire la partenza
- *rank_dest* è un puntatore ad un intero in cui inserire la destinazione

Tramite essa possiamo, tra l'altro, conoscere i nostri vicini (orizzontalmente e

verticalmente). In questo modo possiamo fare varie operazioni (ad esempio propagare un token) senza conoscere il ranking degli altri processi.

In dettaglio, se ci troviamo ad esempio su un toro, in direction specificheremo se vogliamo il rank dei vicini in “verticale” od in “orizzontale”. Nel parametro disp, usando il valore 1, otterremo quelli a distanza 1 (i vicini appunto).

Esempio per ottenere i propri vicini su una topologia ad anello (supponiamo esista già e si chiami ring):

```
int pred[1];
int next[1];
MPI_Cart_shift(ring,0,1,pred,next);
```

Altre funzionalità

Ci siamo limitati nella descrizione delle funzionalità principali di MPI, con particolare riferimento a quelle già presenti nella versione 1. Vediamo ora due nuove funzionalità introdotte da MPI-2, che possono risultare di grande importanza nella realizzazione di software paralleli.

La prima funzionalità è detta “**One-sided Commumication**”. In tutti gli esempi visti, la comunicazione coinvolgeva sempre due (o più entità). Infatti un'operazione di Send, doveva sempre avere una corrispondente operazione di Receive (discorso analogo per le comunicazioni collettive). Si imponeva in sostanza una sorta di sincronizzazione: i due processi dovevano invocare due funzioni complementari.

Con la funzionalità di One-sided Communication, invece, la comunicazione coinvolge unicamente un'entità. Vengono messe a disposizione infatti delle chiamate che consentono di leggere e scrivere nella memoria di un altro processo. Naturalmente, affinché ciò sia possibile, tale processo dovrà aver condiviso inizialmente una quantità di memoria.

Un'altra funzionalità interessante è la gestione dell' I/O. Si parla nello specifico di **parallel-I/O**. E' possibile infatti gestire dei normali file in maniera distribuita. In particolare, un file può essere aperto e gestito da un insieme di processi. Questo è molto utile, quando si devono effettuare delle elaborazioni a partire da un file. Per esempio, si vuole che ogni processo elabori delle informazioni presenti in varie parti del file. In maniera simile, più processi potranno scrivere in contemporanea un file (naturalmente in punti diversi). MPI-2 fornisce delle funzioni per rispondere a queste esigenze. Tuttavia c'è da dire che si tratta di funzioni abbastanza astratte, è lasciata molta libertà di implementazione (e.g. Il nome del file è solo una stringa senza alcuna semantica associata).

Implementazione

Attraverso tutte le discussioni effettuate finora, è emerso chiaramente come moltissime cose dipendano dall'implementazione di MPI. In effetti, le specifiche sono molto generali, lasciando quindi ampio spazio ad implementazioni differenti. Sarà ora descritta brevemente la libreria **Open MPI** (una delle più utilizzate), allo scopo di capire quali possono essere le scelte implementative principali, per realizzare le funzioni che abbiamo visto. E' bene precisare però che Open MPI utilizza un'architettura a componenti, altamente personalizzabile. Vengono messi a disposizione una serie di moduli (ed è possibile crearne di nuovi), tra i quali l'utente può scegliere in base alle proprie esigenze. In questo modo il comportamento di Open MPI può essere diverso a seconda del modulo in uso.

Lanciare un programma

Supponiamo di aver realizzato un programma che fa uso di MPI e di volerlo eseguire. Supponiamo inoltre che il programma sia lo stesso per tutti i nodi, paradigma SPMD (Single Program Multiple Data).

In primo luogo bisogna effettuare la compilazione, Open MPI fornisce dei wrapper. Ad esempio, per compilare un programma scritto in C è sufficiente eseguire:

```
mpicc -o prog prog.c
```

Una volta compilato il programma, ci si scontra subito con un primo problema non definito nelle specifiche. Come distribuire il programma a tutti i nodi? Esistono fondamentalmente due tecniche: effettuare una copia manuale (ed eventualmente una ricompilazione se in ambiente eterogeneo) o utilizzare un *Resource Manager*. La prima modalità è quella più semplice: una volta effettuata la copia dell'eseguibile si crea un file con tutti gli host e si esegue:

```
mpirun -hostfile hosts -np N prog
```

Dove la variabile N indica quanti processi si vogliono utilizzare. Nel file host è poi possibile specificare in modo più dettagliato quanti processi creare per ogni host (e.g. a seconda del numero di core/processori presenti). Open MPI si connette dunque ad ogni host e lancia il programma (per la connessione si utilizza tipicamente SSH).

Questa modalità tuttavia è abbastanza limitativa. L'uso di un Resource Manager offre invece molte possibilità in più. A tal proposito si rimanda alla documentazione di questi applicativi, come ad esempio: Sun Grid Engine[4], SLURM[5], Apple Xgrid[6], ecc. Per dare tuttavia un'idea generale, diciamo che questi strumenti forniscono (tra l'altro) direttamente alla libreria MPI il numero di processi da eseguire e su quali nodi. In questo modo si possono ottenere sofisticati metodi di distribuzione del carico e molto altro.

Ranking

Come è stato descritto, per effettuare le comunicazioni è fondamentale avere a disposizione il ranking dei vari processi. A questo punto è interessante chiedersi come avvenga l'assegnamento del ranking e come si associ ad ogni processo un particolare rank.

Vediamo quindi in maniera più dettagliata la procedura di “avvio” di una computazione con Open MPI. Il funzionamento è abbastanza complesso e coinvolge molte componenti,

tuttavia daremo una visione d'insieme molto semplificata. Un esempio dell'architettura è riportato in figura 7.

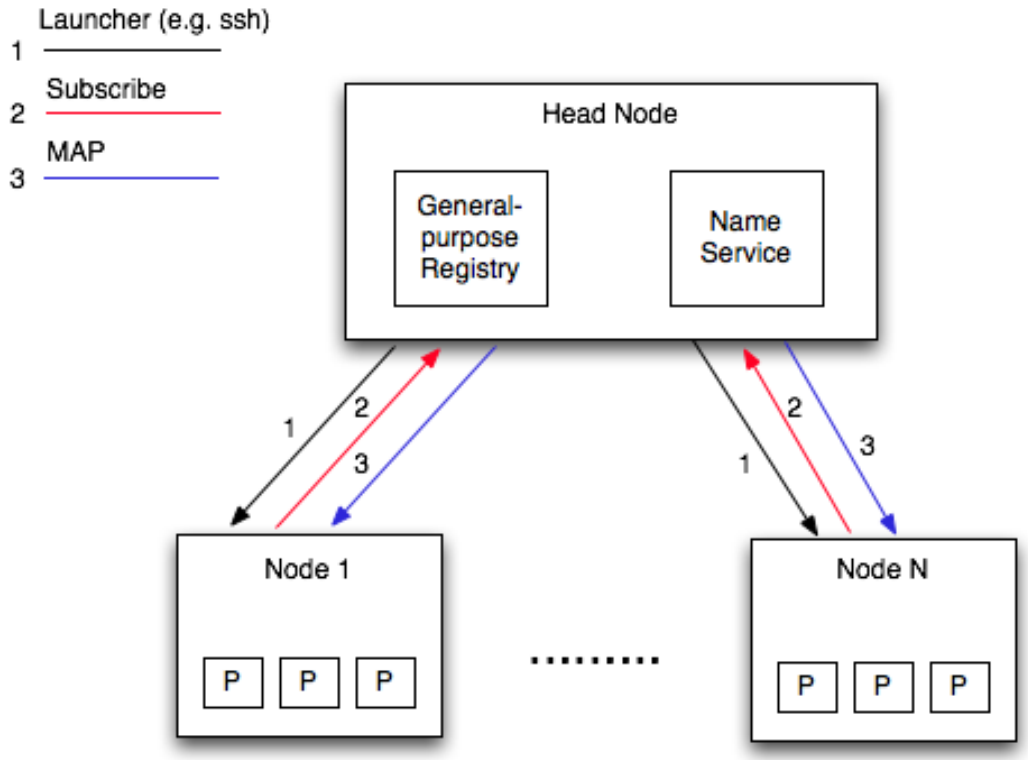

Figura 7: Inizializzazione dei processi con Open MPI

Inizialmente l'Head Node (il nodo coordinatore), si connette a tutti i nodi. Come abbiamo visto nella sezione precedente, questo coordinatore sarà la macchina su cui lanciamo il comando mpirun. Per connettersi ai vari nodi, si utilizza la procedura vista prima (e.g. hostfile e quindi SSH, Resource Manager). Per risolvere il nome dei nodi si utilizza un name service (nel caso dell'hostfile, sarà tipicamente il server DNS).

A questo punto il nodo si sottoscrive ad un notificatore di eventi presente nell'head node e fornisce una serie di informazioni su di se. Il notificatore di eventi, tramite il general-purpose registry, memorizza una mappa delle allocazioni dei vari nodi con i relativi processi. Inoltre attribuisce il rank ad ogni processo. Quando la mappatura è stata completata, il gestore eventi notifica (inviando la mappa) a tutti i nodi (che avevano quindi fatto il subscribe). In questo modo tutti conoscono la collocazione e il rank degli altri processi (e ovviamente anche il proprio). In situazioni articolate, lo stesso gestore eventi può notificare ai nodi l'aggiunta di nuovi processi (caso dinamico).

Comunicazione

Per quanto riguarda la comunicazione, tramite due ordinarie operazioni Send-Receive, restano da chiarire molti aspetti. In particolari cosa succede nel caso di perdita di un messaggio?

L'architettura modulare di Open MPI, non consente di dare un'unica risposta. C'è infatti molta dipendenza a seconda dal modulo che si utilizza. E' poi da notare che la scelta del modulo dipende da come i nodi sono interconnessi. Ad esempio esiste un modulo

specifico per connessioni Infiniband [7], il cui funzionamento è molto diverso rispetto a quello che si può utilizzare su una rete ethernet.

Se ci poniamo in un caso specifico, supponendo che i nodi siano connessi tramite una rete Ethernet, allora il modulo di default è tcp. Come dice il nome, viene stabilita una connessione TCP tra i nodi che vogliono comunicare. In questo modo si suppone che la trasmissione sia affidabile (ci si affida in questo senso a TCP). Pertanto il protocollo in questo caso non prevede acknowledge o timeout.

Sempre in merito al modulo TCP, per l'invio dei messaggi viene adottata la seguente scelta (di tipo Rendezvous). Se la dimensione del messaggio da inviare è inferiore a 64KB, allora lo si spedisce immediatamente. Se invece è superiore a 64 Kb, viene inviato un messaggio al destinatario, che informa della presenza di un messaggio. Quando il processo (del destinatario) esegue una chiamata alla funzione Receive, allora viene inviata una notifica al mittente, indicando di spedire i dati.

Per quanto riguarda invece la comunicazione collettiva, anche in questo caso esistono diversi moduli. Il più semplice, si limita ad eseguire una serie di operazioni di Send/Receive punto-punto. Tuttavia, di default Open MPI adotta delle soluzioni molto più articolate e complesse, per le quali si rimanda alla documentazione ufficiale.

Bibliografia

[1] http://www.open-mpi.org/
[2] http://www.mcs.anl.gov/research/projects/mpich2/
[3] http://www.mpi-forum.org/
[4] http://gridengine.sunsource.net/
[5] https://computing.llnl.gov/linux/slurm/
[6] http://www.apple.com/server/macosx/technology/xgrid.html
[7] http://www.infinibandta.org