Tutorial ASSIST-Cl ver. 1.3.

Versione 1.0 15 Aprile 2004 (Ciullo, Magini, Potiti, Zoccolo)
Versione 1.1 28 Novembre 2006 (Virdis)
Versione 1.2 01 Marzo 2007 (Virdis)

Indice generale

1. Introduzione

ASSIST è un ambiente di programmazione parallela fornito di un linguaggio di coordinamento molto intuitivo per la scrittura di applicazioni ad alte prestazioni. La struttura è modulare e permette di sfruttare sia codice sequenziale scritto nei comuni linguaggi di programmazione c, c++ e Fortran, sia di scrivere nuove applicazioni in uno dei linguaggi ospiti. Questo documento descrive l'ambiente e fornisce una guida per chi vuole avventurarsi nell'uso di ASSIST.
Il linguaggio ASSIST-CL mette a disposizione i seguenti costrutti fondamentali:

- *modulo sequenziale*,
- modulo parallelo (*parmod*),
- grafo di connessione dei moduli (*generic*),
- *stream* che formano le connessioni e quindi il grafo del programma,
- *proc* in cui l'utente incapsula il codice da eseguire.

La figura sottostante mostra uno schema di relazione fra le varie entità.

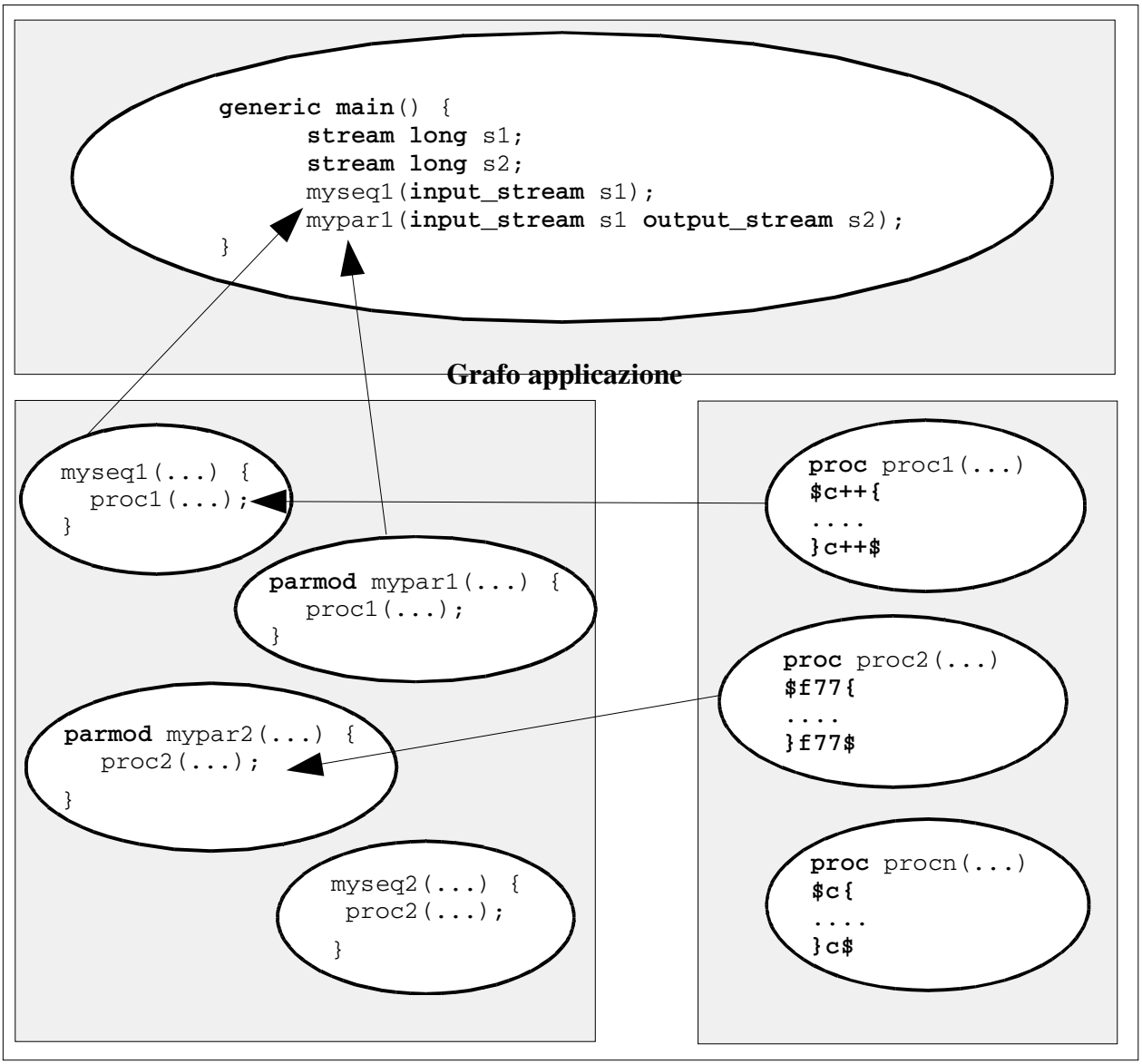


Figura 1. Schema delle relazioni tra entità in un programma ASSIST

La descrizione del linguaggio verrà effettuata *bottom-up* a partire dagli **stream** che rappresentano dei canali con cui due moduli possono comunicare, alle **proc** che descrivono i comandi fino ad arrivare alla definizione del grafo **generic** che descrive la struttura dell'intera applicazione.
Nel documento si ricorre ad un largo utilizzo di esempi e figure per rendere più chiara ed immediata la spiegazione.

2. Stream

Gli **stream** devono essere considerati come dei connettori che permettono lo scambio di

informazioni tra due moduli di ASSIST. Questi connettori hanno un *tipo* e un *nome*. I tipi che si possono utilizzare sono quelli predefiniti di ASSIST oppure un nuovo tipo definito dal programmatore.

Tipi *base* di ASSIST:

Corrispondenza tra tipi ASSIST e tipi interni alle proc nei vari linguaggi ospiti			
ASSIST(Corba)	***C***	***C++***	***Fortran***
octet	unsigned char	unsigned char	integer*1
char	char	char	character
bool	*undef*	bool	logical
short	short	short	Integer*2
long	int	int	integer
long long	long long	long long	integer*8
float	float	float	real*4
double	double	double	real*8
fcomplex (*)	fcomplex	fcomplex	complex*8
dcomplex (*)	dcomplex	dcomplex	complex*16
ref_t (*)	ref_t	ref_t	*undef*
tree_t (*)	tree_t	tree_t	*undef*

(*) non esiste l'equivalente in CORBA.

La sintassi per definire uno stream di tipo array bidimensionale di N*N elementi è la seguente

```
stream long [N][N] A;
```

Questa sintassi viene utilizzata soltanto all'interno del costrutto **generic**, che descriveremo più avanti, dove si definisce il grafo dell'applicazione. In tutti gli altri punti in cui gli stream vengono utilizzati, come nelle interfacce dei moduli e delle procedure, viene seguita la sintassi del linguaggio **c** (*es: long A[N][N]*).

La definizione di un nuovo tipo in ASSIST segue la sintassi del *typedef struct* del linguaggio *c++*. Sotto alcuni esempi di definizione di **stream**:

```
typedef struct {
        long a;
        char b;
} mytype_t;
generic main() {
   stream long str;
   stream mytype_t myst;
   ...
   ...
   ...
}
```

Una volta definiti, possono essere utilizzati per collegare i moduli.

3. Proc

La **proc** è un costrutto per incapsulare codice scritto in uno dei linguaggi ospiti di ASSIST e vengono invocate all'interno dei moduli. Ogni **proc** è formata da un' interfaccia contenente parametri di input e di output indicati rispettivamente con le parole chiave **in**, **out**. Essi descrivono i parametri di cui la **proc** ha bisogno per eseguire un'elaborazione e che, eventualmente, restituisce come risultato. La parola chiave **output_stream** viene utilizzata per passare alla **proc** il riferimento ad uno **stream** di output. Questo permette, per esempio, ad una **proc** di diventare un generatore di dati con lunghezza indefinita. Di seguito riportiamo alcuni esempi di semplici **proc** scritte in c++:

```
proc conta_fino_a_cento()
inc<"iostream">
$c++{
   int i;
   for (i=0;i<100;i++)
      std::cout << "i = " << i+1 << std::endl;
}c++$

proc fattoriale(in long n out long fat)
$c++{
   int i;
   fat = 1;
   for (i=1;i<n;i++)
      fat *=i;
}c++$
```

La prima **proc,** *conta_fino_a_cento,* ha un 'interfaccia vuota e rappresenta un esempio di una **proc** che non ha bisogno di parametri né in ingresso né in uscita. Il suo compito è quello di stampare a

video la lista dei primi cento numeri.
La seconda **proc,** *fattoriale,* possiede in ingresso il parametro *n* del quale deve essere calcolato il fattoriale ed il parametro in uscita *fat* che è il risultato dell'operazione.
Di seguito verranno mostrate altri due esempi di **proc** che hanno o solo parametri in input o solo parametri in output.

```
proc numero_random(out long random_n)
$c++{
            random_n = rand() + 1;
}c++$

proc salva_dati(in long data[N], char nomefile[200])
$c++{
            FILE *f;
            f= fopen(nomefile,"w");
            if (f) {
              for (int i=0;i < N;i++)
                 fwrite(data[i], sizeof(long), 1,f);
            }
}c++$
```

Tutte le **proc** definite sopra sono richiamabili da qualsiasi modulo sequenziale o **parmod**, basta rispettare le interfacce di chiamata.
Particolare attenzione è richiesta quando si utilizza l'**output_stream** di una **proc**. In genere questa variante viene usata quando si vogliono inviare i dati sullo **stream** direttamente dalla **proc**. Ad esempio questo si verifica quando la **proc** è invocata un'unica volta all'interno di un modulo sequenziale, ma si vogliono inviare più dati sullo **stream**. Cerchiamo di spiegarci meglio attraverso un esempio.

```
proc leggi_dati(in char nomefile[200] output_stream long data)
$c++{
            data_t d;
            FILE *f;
            fopen("prova.dat",r);
            while (!feof(f)) {
              fread(&d,sizeof(long),1,f);
              assist_out(data,d);
            }
 }c++$
```

La parola chiave **out** si è trasformata in **output_stream.** Questa trasformazione ci permette di controllare quanti e quali dati inviare in uscita. Nell'esempio, **proc** *leggi_dati,* produce un dato sullo **stream** per ogni record letto dal file. Il costrutto ci permette di inviare un dato sullo **stream** è **assist_out** dicendo al supporto di inviare un dato sullo **stream**, riottenendo il controllo dopo l'invio.

Nelle **proc** è possibile integrare del codice già scritto. Infatti, oltre alla definizione dell'interfaccia, è possibile definire delle sezioni per includere file sorgenti, oggetti e librerie. In questo modo si permette di invocare codice già scritto, oppure di linkare oggetti già esistenti.
Le parole chiavi da usare per includere codice sorgente o binario già scritto sono:

1. **path**: rappresenta una lista di directory dove il compilatore cercherà i file da includere.
2. **inc**: file sorgente che verranno usati nella proc.
3. **obj**: file oggetto da linkare al codice *(non disponibili in questa versione).*
4. **lib**: librerie esterne usate dentro il codice della proc *(non disponibili in questa versione).*

Vediamo un esempio:

```
proc stampa()
path<"/home/pippo/src/">
inc<"iostream","myobject.hpp">
$c++{
     myobject ob;
     ob.name = "ciao";
     std::cerr << "myobject name = " << ob.name << std::endl;
}c++$
```

In questa **proc** usiamo un oggetto che abbiamo definito precedentemente in un sorgente *c++*. In **path** definisco la directory dove si trova. Nella sezione indicata con **inc** viene incluso il sorgente. Ora nel codice posso definire e utilizzare tranquillamente un oggetto della mia classe c++.

4. Sequenziale

Il modulo sequenziale è costituito da un' interfaccia verso altri moduli e da una serie di chiamate a **proc.** Le chiamate alle **proc** rappresentano i comandi che il modulo sequenziale esegue quando viene chiamato.
Per attivare un sequenziale tutti gli **stream** in ingresso devono avere un dato. Questo significa che gli **stream** in ingresso ad un sequenziale si trovano in *and*. Questo comportamento del sequenziale non può essere modificato. Nel caso i cui volessimo un *or* oppure una condizione più complessa di un dato sullo **stream**, bisogna definire un altro tipo di modulo; esattamente un **parmod one** di cui parleremo più avanti.
Il sequenziale nei casi più semplici può essere visto come una **proc**, rimane comunque la differenza sostanziale a livello di linguaggio. Sotto viene mostrato un esempio di un sequenziale che chiama la **proc** *conta_fino_a_cento()* .

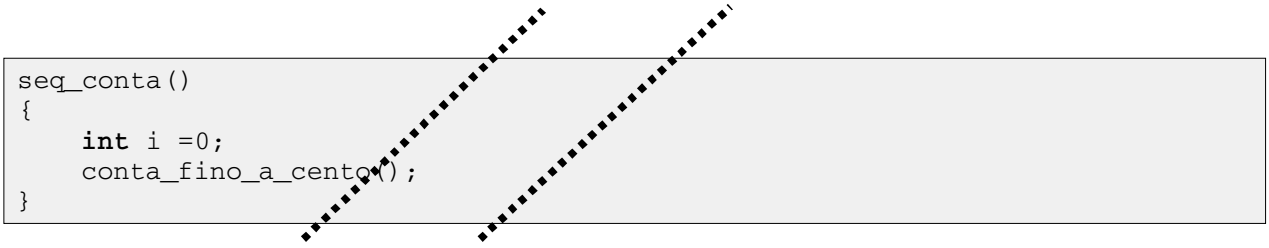
```
chiama_conta_fino_a_cento() {
          conta_fino_a_cento();
}
proc conta_fino_a_cento()
inc<"iostream">
$c++{
          int i;
          for (i=0;i<100;i++)
               std::cout << "i = " << i << std::endl;

}c++$
```

La stessa cosa in ASSIST si può scrivere in modo più compatto fondendo il sequenziale con la **proc** che chiama al suo interno. [1]

```
chiama_conta_fino_a_cento()
inc<"iostream">
$c++{
         int i;
         for (i=0;i<100;i++)
              std::cout << "i = " << i << std::endl;
}c++$
```

In questo modo noi stiamo definendo un sequenziale con il codice scritto nel linguaggio ospite tutto in uno. Comunque per ASSIST i due modi sono del tutto equivalenti, ma ciò non deve trarre in inganno. Infatti il seguente codice non è ammesso.



```
seq_conta()
{
    int i =0;
    conta_fino_a_cento();
}
```

E' errato perché questo è un sequenziale puro, dentro le parentesi { } possono essere inserite solo chiamate a **proc.** Ne vedremo più avanti l'utilizzo.

Ora, anticipando un po' i tempi sulla costruzione del grafo dell'applicazione, possiamo cimentarci a scrivere il nostro primo programma in ASSIST che conta fino a cento.

1 Il sequenziale puro ed il sequenziale con le chiamate alle **proc** sono sempre equivalenti escluso il caso in cui all'interno del sequenziale si voglia invocare esplicitamente più volte **assist_out**. In questo caso è necessario la distinzione tra sequenziale e proc da chiamare. Dentro un sequenziale puro la scrittura sullo **stream** avviene in maniera implicita una sola volta dopo la terminazione del codice del sequenziale.

```
generic main() {

          chiama_conta_fino_a_cento();
}

chiama_conta_fino_a_cento()
{
          conta_fino_a_cento();
}

proc conta_fino_a_cento()
inc<"iostream">
$c++{
          int i;
          for (i=0;i<100;i++)
            std::cout << "i = " << i << std::endl;

}c++$
```

Figura 2. Primo programma in ASSIST

Per tradizione mostriamo un esempio ancora più breve di quello appena visto: *helloworld* scritto in ASSIST.
Per il sequenziale senza la chiamata alla **proc** valgono le stesse regole della **proc** per quanto riguarda le parole chiavi **inc, path, obj, lib**.

```
generic main() {
          hello_world();
}

hello_world()
inc<"iostream">
$c++{
   std::cout << "hello world from assist" << std::endl;
}c++$
```

Figura 3. Hello World in ASSIST

Questi esempi chiamano un solo sequenziale. Spingiamoci oltre vedendo come si possa creare un *pipeline* puro semplicemente unendo i sequenziali attraverso la sezione del grafo. L'unica cosa che dobbiamo fare è dire come i moduli definiti si compongono.

```
generic main() {
   stream long signal;
   stream long n;
   seq_conta(output_stream n);
   seq_fattoriale(input_stream n output_stream signal);
   hello_world(input_stream signal);
}

proc conta(output_stream long myfat)
$c++{
   int i;
   for (i=0;i<100;i++){
     std::cout << "i = " << i << std::endl;
     assist_out(myfat,i);
   }
}c++$

seq_conta(output_stream long n) {
   conta(output_stream n);
}

seq_fattoriale (input_stream long n output_stream long fat) {
   fattoriale(in n out fat);
}

proc fattoriale (in long n out long fat)
$c++{
   int i;
   fat = 1;
   for (i=1;i<n;i++)
     fat *=i;
}c++$

hello_world(input_stream long res)
$c++{
   std::cout<<"hello world in assist con fattoriale = "<<res<< std::endl;
}c++$
```

Figura 4. Esempio di pipeline in ASSIST

Il codice del sequenziale della **proc** *fattoriale* è un semplice codice c++ per calcolare il fattoriale. Notare la differenza tra le chiamate delle due **proc**: *conta* e *fattoriale*. La prima ha come parametro di output un **output_stream**, la seconda invece ha come parametro output un **out**. Questo implica che nella **proc** *conta* per spedire fuori il risultato devo usare **assist_out**, mentre nella **proc** fattoriale il parametro *fat* è una variabile ed il risultato viene spedito fuori alla fine della **proc.**
L'altra cosa interessante da notare in questo esempio è la parte del codice non ancora spiegato identificato dalle parole chiave **generic main**.

```
generic main() {
            stream long signal;
            stream long n;
            seq_conta(output_stream n);
            seq_fattoriale(input_stream n output_stream signal);
            hello_world(input_stream signal);
}
```

Di questo codice, per adesso, basta sapere che definisce gli **stream** su cui i moduli dell'applicazione lavorano e crea il grafo ASSIST dell'applicazione.

Nell'esempio gli **stream** definiti sono:

- Lo **stream** *n* che connette *seq_conta* con *seq_fattoriale.* Viene utilizzato per scambiare i dati tra i due. Il sequenziale *seq_fattoriale* calcola il fattoriale fino ad *n* che è il numero passato da *seq_conta*.
- Lo **stream** *signal* che connette *seq_fattoriale* con *hello_world.*

5. Parmod

Ora che abbiamo conosciuto le **proc** e il modulo sequenziale, possiamo passare a conoscere uno dei costrutti più importanti: il **parmod**.
Il **parmod** viene usato per parallelizzare l'esecuzione di un determinato algoritmo. Nella parallelizzazione di un algoritmo bisogna tener conto di diversi problemi. Le strategie che si adottano dipendono molto dal tipo di algoritmo che stiamo parallelizzando. Il **parmod** nasce come costrutto generale adattabile alle diverse esigenze. Nel **parmod** esiste il concetto di processore virtuale. Ogni processore virtuale è un'entità di calcolo che esegue una sequenza di **proc** in sequenziale. L'opportuna composizione dei questi all'interno del parmod determina il parallelismo del modulo Un altro concetto esistente nel **parmod** è il concetto di stato. Lo stato di un **parmod** è dato dagli attributi che vengono definiti attraverso la parola chiave **attribute.** Questi attributi possono essere di diverso tipo con diversi significati semantici. Ogni tipo tenta di risolvere vari

problemi che un programmatore incontra durante l'implementazione di un algoritmo parallelo. Più avanti spiegheremo questi concetti attraverso degli esempi. Il **parmod** a livello funzionale deve essere visto come un *pipeline* di tre stadi: stadio di input, stadio di calcolo e stadio di output. In ogni stadio il programmatore possiede diversi strumenti per specializzare il **parmod** per il suo algoritmo.

5.1 Struttura di un parmod

Il **parmod** (*modulo parallelo*) possiede un'interfaccia per agganciarsi ad altri moduli esattamente come il modulo sequenziale, in più possiede una struttura interna più ricca che possiamo dividere in quattro sezioni principali:

- Una *sezione di definizioni*: si definiscono le variabili e il loro tipo. Vengono definite le variabili di controllo, le variabili di stato usate all'interno del **parmod** e gli **stream** interni.
- Una *sezione di input_section:* qui si definisce il comportamento del parmod quando è presente un dato sugli stream in ingresso: vengono definite le politiche di distribuzione dei dati ricevuti in ingresso, si definiscono eventuali funzioni di inizializzazione delle variabili usate, si aggiornano le variabili di controllo, ecc.
- Una *sezione di virtual_processors:* qui si specifica il calcolo che deve essere fatto, viene inserito il codice per la definizione dell'insieme di lavoro dei processori virtuali, le chiamate alle **proc,** la definizione di cicli paralleli come **for** e **while.**
- Una *sezione di output_section:* qui si definiscono le funzioni di post elaborazione: vengono definite le politiche di collezione dei dati ed eventualmente funzioni post calcolo prima di spedire i dati in uscita.

In figura 5 viene mostrata una rappresentazione schematica di queste sezioni.

```
parmod first (.....) {
        definizione della topologia
        definizione delle variabili di controllo
        definizioni degli attributi (replicated o condivise)
        definizione degli stream interni

        input_section {
                definizione delle procedure di init.
                distribuzioni degli stream in input dell'interfaccia
                aggiornamento delle variabili di controllo
                test delle condizioni di terminazione
        }

        virtual_processors {
                definizione dell'insieme di lavoro
                associazione stream in ingresso/uscita proc da chiamare
                definizione di cicli paralleli
        }

        output_section {
                definizione delle politiche di collezione dei dati,
                definizione codice post calcolo
        }
}
```

*Figura 5. schema di un **parmod***

Andiamo a vedere in dettaglio ognuna di queste sezioni.

5.1.1 Sezione di definizioni

Topologia

In un parmod dobbiamo definire il tipo di topologia, che può essere **one**, **none**, **array**.

- **one** definisce un **parmod** con un solo processore virtuale,
- **none** definisce un **parmod** con un indefinito numero di processori virtuali anonimi,
- **array** definisce un **parmod** con un numero prefissato di processori virtuali identificabili mediante indici. Il *naming* permette, durante l'esecuzione, di condividere variabili di stato tra i processori virtuali. Spiegheremo più avanti cosa significa.

La scelta della topologia deve essere fatta in base alle peculiarità del nostro algoritmo in quanto determina il tipo di processori virtuali a disposizione e il tipo di attributi utilizzabili.
In un certo senso definendo la topologia, definiamo anche il tipo di comportamento del nostro **parmod**. La possibilità di avere diverse topologie permette al **parmod** di essere modellabile, se ad esempio si vuole avere un programma parallelo che replichi l'esecuzione di una funzione pura (nessuna dipendenza tra i dati) si definirà un parmod di topologia **none**.
Se nella nostra applicazione abbiamo bisogno di uno stato che deve essere condiviso tra i vari processori virtuali o se il nostro algoritmo ha bisogno di uno stato che deve essere mantenuto indipendentemente dall'attivazione, si definirà una topologia **array**.
La topologia **one**, invece, nasce come caso particolare di un sequenziale in cui si voglia gestire il nondeterminismo sugli **stream**. La topologia **one**, infatti, permette di avere un unico modulo che esegue delle funzioni in base allo **stream** attivato. Questo permette di discriminare cosa eseguire o a chi spedire il risultato in base allo **stream** attivato.
La definizione della topologia implica una restrizione sul tipo di attributi ASSIST che si possono definire. In tabella viene mostrata tale corrispondenza.

Topologia	***Attributi definibili***
topology one	attributi singoli del **VP**
topology array	attributi partizionati e/o replicati
topology none	attributi replicati

Variabili di controllo

Le variabili di controllo definiscono variabili che verranno condivise in lettura/scrittura tra le sezioni di ingresso/uscita di un **parmod**. Esse permettono di poter definire delle ulteriori condizioni

di terminazione o di controllare il flusso sugli **stream**. Con le variabili di controllo si possono creare *loop* impliciti su **stream**. Per *loop* impliciti si intendono cicli fuori dal singolo modulo. Per esempio si può decidere che un modulo venga attivato un numero *n* volte oppure che un modulo termini quando viene verificata una determinata condizione.
Esempio:

```
parmod pippo(...) {
  attribute long conta;
  init {
    conta = 0;
  }

  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
      operation {
          conta++;
      }<use {conta}>
    }
  } while (conta < 10)
  ....
}
```

In questo esempio viene utilizzata la variabile *conta* per contare il numero di volte che il modulo parallelo riceve un dato sullo **stream**. Quando *conta* arriva al valore *10*, il modulo termina e attiva la terminazione di tutti i moduli a lui collegati. La riga di codice

```
<use {conta}>
```

serve per dire ad ASSIST quali sono le variabili di controllo utilizzate all'interno della sezione **operation**.

Definizioni degli attributi (replicated o partitioned)

Gli attributi **replicated** o *partitioned* definiscono variabili locali dei processori virtuali per il modulo parallelo. La visibilità degli attributi sono a livello di **input_section** e **virtual_processor**.
Gli attributi servono a dare uno stato ai processori virtuali di un **parmod** che rimane consistente indipendentemente dall'attivazione o no del processore virtuale.
Gli attributi *partitioned* seguono il principio dell' *Owner-Compute-Rule*, la quale implica che solo il processore virtuale proprietario può leggere e scrivere, mentre tutti gli altri possono solo leggere.
Gli attributi **replicated** sono replicati su tutti i processori virtuali presenti nella topologia. Ogni processore virtuale possiede il suo ed è l'unico che può leggere o scrivere.
Gli attributi **replicated** possono essere utilizzati sia nei **parmod** con topologia **array** che **none**, mentre gli attributi *partitioned* sono ammessi solo nel **parmod array**.

Esempio:

```
attribute long S[N][N] scatter S[*i0][*j0] onto Pv[i0][j0];
attribute bool diff replicated;
```

La prima riga definisce un attributo partizionato dove ogni elemento viene associato in maniera isomorfa ad ogni processore virtuale. La seconda riga definisce un attributo **replicated**.
Le variabili libere utilizzate per la definizione degli attributi e come vedremo in seguito per la definizione delle distribuzioni devono essere sempre nuove, ossia non devono essere mai state definite e utilizzate precedentemente nello stesso **parmod**.

Definizione degli attributi per il parmod one

Anche nel **parmod one** è possibile avere delle variabili di stato, anche se ovviamente non sono di tipo **replicated** o *partitioned*.

Esempio:

```
parmod comando (... ) {
  topology one Pv;
  attribute long S onto Pv;
.....

}
```

Nell'esempio si definisce una variabile S, di tipo long, che rappresenta una variabile di stato del **parmod one**. In questo esempio **attribute** ed **onto** sono parole chiave di ASSIST.

Definizione degli stream interni

La definizione idegli stream interni è necessaria quando il tipo dell' **output stream** del parmod è diverso dal tipo di uscita della **proc,** l'utilizzo negli altri casi è a discrezione del programmatore.
Un esempio di utilizzo si ha quando all'interno dell'elaborazione parallela si vuole utilizzare una **proc** che lavora su un elemento scalare (*es: long x*) e l'uscita del **parmod** è una matrice di N*N elementi di tipo *long*. In questo caso, si deve dichiarare uno **stream** interno del tipo di output della **proc** (*es: stream long res*) e si devono collezionare tutti i valori sullo **stream** interno per poterli inviare come una matrice sullo **stream** di uscita del **parmod**. Un esempio dell'utilizzo dello **stream** interno è riportato nella sezione riguardante l'**output_section** (vedi pag 28).

5.1.2 Input_Section

Definizione della procedura di inizializzazione i**nit**.

La definizione della procedura di **init** permette di poter inizializzare sia le variabili di stato, sia le variabili di controllo. Viene eseguita quando il parmod viene istanziato.

Esempio (inizializzazione delle variabili di stato)

```
parmod () {
  // dichiarazione di una variabile di stato replicata
  attribute bool diff replicated;
....
  init {
    VP i {
      init_false(out diff);
    }
  }
.....
}

proc init_false(out bool b)
$c++{
  b=false;
}c++$
```

Definizione della lista di guardie in ingresso

Le guardie della sezione di ingresso sono definite da un **identificatore** unico e dai seguenti parametri opzionali:

- la ***priorità***
- la ***condizione*** che abilita o disabilita la guardia
- ***espressione sugli stream*** di ingresso.(es *A*, oppure *B*, oppure *A&&B,* e cosi' via)

L'esempio sottostante riporta il caso in cui manca solamente il parametro riguardante la priorità. Nel caso in cui la variabile matrice_presente ha valore true si abilita la seconda guardia e si ricevono i dati dallo **stream** B, altrimenti si ricevono i dati dallo **stream** A.

```
parmod pippo (...) {
  attribute bool matrice_presente;
  input_section {
    guard1: on , !matrice_presente, A {
      ....
    }
    guard2: on , matrice_presente, B {
      ....
    }
  }
  ...
}
```

Distribuzioni dei dati in input sullo stream

La distribuzione di un dato dello **stream** ai processori virtuali segue alcune regole che definiscono a quale processore/i virtuale/i viene spedito il valore contenuto nello **stream** o su quali variabili di stato viene salvato il dato contenuto nello **stream**. Le distribuzioni permesse sono le seguenti:

- **scatter**: il dato viene partizionato e inviato ai processori virtuali di competenza in base alla regola della formula di **scatter**.
- **broadcast**: invia un dato a tutti i processori virtuali presenti nella topologia.
- **on_demand**: il dato viene inviato al primo processore virtuale libero.
- **scheduled**: si può indirizzare l'indice del processore virtuale a cui si vuole inviare il dato. In caso di processore virtuale occupato, la **scheduled** si blocca.

Esempi:

- distribuzione **scatter** isomorfa su un attributo.

```
parmod pippo (input_stream long A[N][N] ...) {
  topology array [i:N][j:N] my_vp;
  attribute long S[N][N];
  input_section {
    guard1: on , , A {
      distribution A[*u][*w] scatter to S[u][w];
    }
  }
  .....
}
```

- distribuzione **scatter** isomorfa sui processori virtuali.

```
parmod pippo (input_stream long A[N][N] ...) {
  topology array [i:N][j:N] my_vp;
  input_section {
    guard1: on , , A {
      distribution A[*u][*w] scatter to my_vp[u][w];
    }
  }
  ...
}
```

- distribuzione **scatter** prefissa con schiacciamento di una dimensione sui processori virtuali.

```
parmod pippo (input_stream long A[N][N] ...) {
  topology array [i:N] my_vp;
  input_section {
    guard1: on , , A {
      distribution A[*k0][] scatter to my_vp[k0];
    }
  }
  ...
}
```

- distribuzione **broadcast**

```
parmod pippo (...) {
  topology none my_vp;
  input_section {
    guard1: on , , A {
      distribution A broadcast to my_vp;
    }
  }
  ...
}
```

- distribuzione **on_demand**.

```
parmod pippo (...) {
  topology none my_vp;
  input_section {
    guard1: on , , A {
      distribution A on_demand to my_vp;
    }
  }
  ...
}
```

distribuzione **scheduled** con costante: specificata direttamente nella distribuzione

```
parmod pippo (...) {
  topology array [i:N] my_vp;
  input_section {
    guard1: on , , A {
      distribution A scheduled to my_vp[3];
    }
  }
  ...
}
```

- distribuzione **scheduled** con variabile libera, il cui valore è calcolato in una operation, in funzione del dato in ingresso e delle variabili di controllo. Nel caso di topologia **array** multidimensionali, le differenti dimensioni possono essere specificate in maniera indipendente, usando uno dei precedenti modi.

```
parmod pippo (...) {
  topology array [i:N] my_vp;
  attribute long round;
  input_section {
    guard1: on , , A {
       distribution A scheduled to my_vp[*dest];
       operation {
           dest=(A.idx+round)%7;
       }<use {round}>;
    }
  }
  ...
}
```

- distribuzione **scheduled** nel caso di topologia **none**, è possibile selezionare il Pv destinatario nell'operation assegnandone l'indice alla variabile Pv

```
distribution A scheduled to Pv;
operation {
          Pv=(A.idx+round)%7;
}<use {round}>;
```

Vincoli della distribuzione **scheduled**

- Una distribuzione **scheduled** con destinatario variabile richiede la presenza di una operation.
- Se in una stessa guardia sono presenti più distribuzioni **scheduled**, cosa attualmente non ammessa, tutte dovranno avere lo stesso destinatario (eventuali variabili libere diventano variabili legate nelle occorrenze successive).
- In una guardia non possono essere presenti sia distribuzioni **scheduled** sia di altro tipo.

Esempio: Implementazione della computazione sistolica

Una computazione sistolica su una matrice NxN è implementata utilizzando un parmod con topologia **array**[N]; su ogni Pv è mappata una riga della matrice.

Quando il parmod riceve una riga, essa viene distribuita al Pv d'appartenenza.

Quando il parmod riceve una colonna, essa viene distribuita in scatter a tutti i Pv, mentre l'indice di colonna viene distribuito in broadcast.

Esempio (scheduled)

```
// -*- C++ -*-

#define N 10

typedef struct {
  long idx;
  long row[N];
} row_t;

typedef struct {
  long i;
  long j;
  long val;
} res_t;

generic main() {
  stream row_t rows;
  stream long[N] cols;
  stream long col_idx;
  stream res_t result;

  genera1  (output_stream rows);
  genera2  (output_stream col_idx,cols);
  sisto    (input_stream rows,col_idx,cols output_stream result);
  raccogli (input_stream result);

}

raccogli(input_stream res_t r)
inc<"iostream">
$c++{
  std::cerr<<"Computed element ("<<r.i<<','<<r.j<<") = "<<r.val<<"\n";
}c++$

genera1(output_stream row_t r) {
  fgen1(output_stream r);
}


proc fgen1(output_stream row_t r)
inc<"iostream">
$c++{
  row_t t;

  for (int j=0;j<N;j++) {
    t.idx=j;
    for (int i=0;i<N;i++)
      t.row[i] = i;
    assist_out(r,t);
  }
}c++$
```

Esempio (scheduled)

```
genera2(output_stream long col_idx, long cols[N]) {
  fgen2(output_stream col_idx, cols);
```

Esempio (scheduled)

```
}


proc fgen2(output_stream long col_idx, long cols[N])
inc<"iostream">
$c++{
  for (int j=0;j<N;j++) {
    int c[N];
    for (int i=0;i<N;i++)
      c[i] = i;
    assist_out(col_idx,j);
    assist_out(cols,c);
  }
}c++$


parmod sisto(input_stream row_t r,long ixc,long cls[N]
             output_stream res_t res) {
  topology array [i:N] Pv;
  attribute short pres[N] replicated;
  attribute long data[N] replicated;

  init {
    VP i {
      init1(out pres);
    }
  }

  do input_section {
    row_g: on , , r {
      distribution r scheduled to Pv[*ix]; // necess oper
      operation {
        ix=r.idx;
      }
    }
    col_g: on , , ixc&&cls {
      distribution ixc broadcast to Pv[i];
      distribution cls[*j] scatter to Pv[j];
    }
  } while (true)

  virtual_processors {
    elab1(in row_g) {
      VP i {
         got_row(in r out pres, data output_stream res);
      }
    }
    elab2(in col_g) {
      VP i {
         got_col(in i,ixc,cls[i] out pres, data output_stream res);
      }
    }
  }

  output_section {
    collects res from ANY Pv;
  }
```

Esempio (scheduled)

```
}
```

```
proc init1(out short p[N])
$c++{
  for (int i=0;i<N;i++) p[i] = 0;
}c++$


proc got_row(in row_t r out short pres[N],long data[N] output_stream res_t rs)
$c++{
  for(int i=0;i<N;++i) {
    if(pres[i]) {
      res_t t={r.idx,i,r.row[i]*data[i]};
      assist_out(rs,t);
      pres[i]=0;
    } else {
      data[i]=r.row[i];
      ++pres[i];
    }
  }
}c++$

proc got_col(in long i, long ix, long col out short pres[N], long data[N]
             output_stream res_t rs)
$c++{
  if(pres[ix]) {
    res_t t={i,ix,data[ix]*col};
    pres[ix]=0;
    assist_out(rs,t);
  } else {
    data[ix]=col;
    ++pres[ix];
  }
}c++$
```

Aggiornamento delle variabili di controllo

La lettura e la scrittura delle variabili di controllo può essere fatta solo dentro la sezione **operation**. Questa sezione viene chiamata subito dopo il test della *guardia* e le variabili di controllo vengono bloccate in scrittura. All'uscita della **operation** vengono aggiornati i valori che sono stati modificati e le variabili vengono sbloccate in scrittura. Questo comportamento è necessario perché le variabili di controllo sono condivise tra la **input_section** e l'**output_section**.

Esempio:

```
parmod pippo (...) {
  attribute bool matrice_presente;
  input_section {
    guard1: on ,, A {
      ....
      operation {
        matrice_presente = true;
      }<use {matrice_presente}>
    }
  }
  ...
}
```

Test delle condizioni di terminazione

La terminazione di un modulo può avvenire essenzialmente in due modi. In modo implicito, quando vengono chiusi tutti gli stream in ingresso, oppure esplicitamente quando viene verificata la condizione di terminazione.

Esempio (terminazione implicita)

```
parmod pippo (...) {
  ...
  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
    }
  } while (true)
  ...
}
```

Esempio (terminazione esplicita)

```
parmod pippo (...) {
  ...

  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
      operation {
         conta++;
      }<use {conta}>
    }
  } while (conta < 10)
  ...
}
```

Nel primo caso il modulo termina quando viene chiuso lo **stream** *A,* nel secondo caso il modulo termina quando la variabile di controllo *conta* raggiunge il valore *10* o nel caso che lo stream *A* venga chiuso. L'eventualità in cui il modulo termina e sullo **stream** ci sono ancora dati non letti è

da considerarsi errata, infatti il programma ASSIST potrebbe bloccarsi o dare errore a tempo di esecuzione.

5.1.3 virtual_processors

Definizione dell'insieme di lavoro:

Una volta definito il comportamento e le distribuzioni che devono essere effettuate nella **input_section** si passa alla sezione dei **virtual_processors** dove si definiscono le associazioni tra gli **stream** di input e le **proc** da chiamare. Un esempio di associazione viene proposto nell'esempio seguente:

```
parmod pippo (input_stream long A[N][N] ...) {
   ...
   do input_section {
    guard1: on , , A {
         distribution A[*u][*w] scatter to S[u][w];
     }
   } while (true)

   virtual_processors {
     elab1 (in guard1 out ris) {
       VP i,j {
         .....
       }
     }
   }
   ....
}
```

Le etichette cerchiate servono al compilatore per poter associare agli **stream** di ingresso l'elaborazione da far partire.
L'etichetta *elab1* rappresenta un nome arbitrario dato dall'utente, *guard1* rappresenta l'etichetta per associare l'evento che avviene nella **input_section**. L'etichetta *guard1* fornisce anche l'ambiente delle variabili che sono visibili all'interno dell'elaborazione *elab1*.
Nell'esempio sopra la variabile *A* è uno **stream** definito nell'interfaccia del modulo.
L'etichetta **VP** è una parola chiave di ASSIST. In caso di **parmod** con topologia **array** la parola chiave **VP** deve essere seguita dalla definizione dell'insieme dei processori virtuali che partecipano all'elaborazione. Questo permette di discriminare l'insieme dei processori virtuali che devono lavorare o di discriminare insiemi diversi di processori virtuali che eseguiranno una determinata **proc**. Si pensi per esempio al caso di algoritmi che lavorano sugli elementi dei bordi di una matrice in maniera diversa dagli elementi all'interno di una matrice. Sotto viene presentato un esempio:

```
parmod pippo (input_stream long A[N][N] ...) {
  ...
  virtual_processors {
    elab1 (in guard1 out risultato) {
       VP i=0..0 ,j=0..N-1 {
          proc_bordo(in i,j out S[i][j]);
       }
       VP i=N-1..N-1 ,j=0..N-1 {
          proc_bordo(in i,j out S[i][j]);
       }
       VP i=1..N-2 ,j=0..0 {
          proc_bordo(in i,j out S[i][j]);
       }
       VP i=1..N-2 ,j=N-1..N-1 {
          proc_bordo(in i,j out S[i][j]);
       }
       VP i=1..N-2 ,j=1..N-2 {
          proc_interno(in i,j out S[i][j]);
       }
    }
  }
  ....
}
```

L'esempio precedente mostra la definizione di un'elaborazione *elab1* che fa lavorare i processori virtuali dei bordi di una matrice in maniera diversa rispetto a quelli interni. Il seguente, invece mostra la definizione di un'elaborazione *elab2* che fa lavorare tutti i processori virtuali della topologia.

```
parmod pippo (...) {
  ...
  virtual_processors {
    elab2 (in guard2 out risultato) {
      VP i,j {
         proc_matrice(in S[i][j]  output_stream risultato);
      }
    }
  }
...
}
```

Definizione di cicli paralleli:

A volte può essere utile iterare un'elaborazione. Per esempio supponiamo che dobbiamo calcolare qualche criterio di convergenza. ASSIST mette a disposizione i cicli paralleli all'interno di un **parmod**. Sotto si vede un esempio di un ciclo **for** parallelo. Notare che la variabile *S* è un attributo di stato *partitioned* del **parmod**. ASSIST ci garantisce che ogni volta che inizia un'iterazione lo stato viene aggiornato in modo che i processori virtuali vedano lo stato degli altri processori virtuali consistente con l'iterazione in corso.

```
parmod pippo (...) {
  ...
  virtual_processors {
    elab1 (...) {
      VP i,j {
         for (h=0; h<N; h++) {
           Felab (in S[i][h], S[h][j], S[i][j] out S[i][j]);
         };
         assist_out (ris, S[i][j]);
      }
    }
  }
...
}
```

Sotto riportiamo un altro esempio di ciclo parallelo, questa volta non è un **for**, ma un **while**. La condizione del **while** esegue il test della variabile **replicated** *diff*. Tutti i valori della variabile *diff* vengono valutati in *or*, e se almeno un valore della variabile è uguale a **true**, il ciclo continua. *diff* è un attributo definito nella sezione dichiarazioni in questo modo:

```
attribute bool diff replicated;
```

```
parmod pippo (...) {
  ...
  virtual_processors {
    elab1 (...) {
      VP i,j {
        do {
          calcola (in S[i][], S[i-1][], S[i+1][] out S[i][], diff);
        } while (reduce (diff, ||) == true);
      }
    }
  }
}
```

5.1.4 output_section

Nell'**output_section** vengono definite le politiche di collezione dei dati ricevuti dai processori virtuali e il loro invio all'esterno del **parmod**. Le politiche messe a disposizione di ASSIST sono due: **ALL** e **ANY**. La politica **ANY** permette di prendere un dato dal primo processore virtuale che spedisca un risultato. La politica **ALL** invece permette la collezione da tutti i processori virtuali. La combinazione tra politiche di collezione ed eventuale codice sequenziale da eseguire permettono di definire una moltitudine di comportamenti. Più avanti verranno mostrati alcuni modi di operare nella **output_section**.

```
parmod pippo (...output_stream long B[N][N]) {
  stream long ris;
  ...
  output_section {
    collects ris from ALL Pv[i][j] {
      int el;
      int B_[N][N];
      AST_FOR_EACH(el) {
        B_[i][j] = el;
      }
      assist_out (B, B_);
    }<>;
  }
  ...
}
```

L'esempio sopra mostra un collezione di valori singoli da parte dei processori virtuali. La **collects** viene chiamata quando tutti i processori virtuali hanno generato un dato. Qui viene mostrato come si recuperano i dati da ogni processore virtuale. Il costrutto **AST_FOR_EACH** serve ad iterare sugli elementi inviati dai processori virtuali. Le variabili i e j, vengono impostate dal supporto con i valori del processore virtuale corrente. Non si deve ipotizzare nessun ordinamento sui valori che possono assumere i e j. In questo esempio si definisce una matrice locale che viene riempita con i valori ricevuti e poi viene spedita all'esterno del parmod.[2]
Di seguito mostriamo un **output_section** senza nessuna elaborazione. Viene chiamata ogni qualvolta un processore virtuale produce un risultato.

```
parmod pippo (...output_stream long C) {
  ...
  output_section {
    collects C from ANY Pv;
  }
}
```

Ora che abbiamo descritto il parmod in generale realizzeremo tre parmod con topologia diversa cercando, per ogni esempio, di enfatizzare le peculiarità di ogni parmod unito alla topologia.

5.2 Parmod con topologia one

Il nostro esempio sarà formato da un sequenziale che manda il tipo di comando e un **parmod one** che a seconda dello **stream** dal quale riceve esegue il comando associato. In figura 6 è mostrato lo schema della nostra applicazione.

2 Il codice nella **collects** è codice *c++*, pertanto valgono le stesse limitazioni che si applicano nel codice c++ in una qualsiasi funzione; in particolare, le strutture dati non possono essere troppo grandi (causerebbero un overflow dello stack); per ovviare a tale problema si può utilizzare, ad esempio, l'allocazione dinamica.

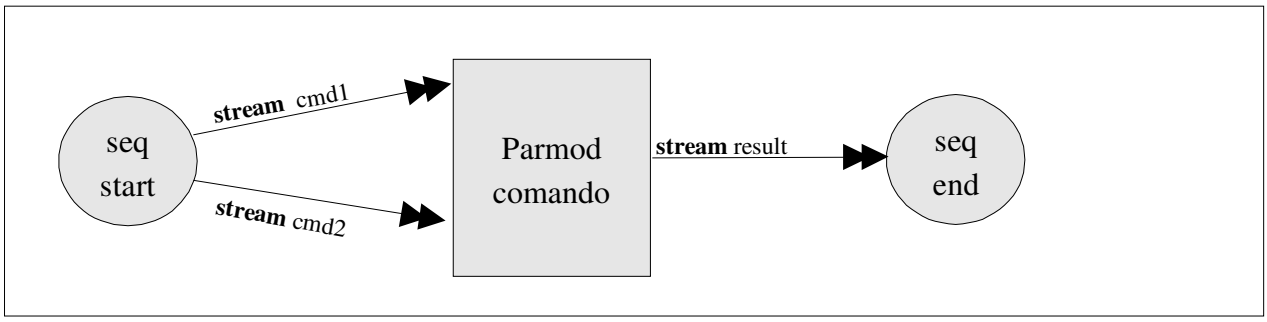


*Figura 6. Grafo applicazione di esempio **parmod one***

Definiamo prima di tutto l'interfaccia del nostro **parmod** e la topologia. Il nostro **parmod** riceve i dati da due **stream** di tipo **long**, *cmd1, cmd2,* lo **stream** di output è ancora di tipo **long** chiamato *result*. Il risultato può essere prodotto sia dal comando *1* che dal comando *2*. La topologia definita è **one** dal nome *my_vp*.

```
parmod comando(input_stream long cmd1, long cm2 output_stream long result) {
   topology one my_vp;
   ...
}
```

Passiamo ora a definire l'**input_section** del nostro **parmod**.

```
parmod comando(input_stream long cmd1, long cmd2 output_stream long result) {
   topology one my_vp;

   do input_section {
     guard1: on , , cmd1 {
       distribution cmd1 broadcast to my_vp;
     }
     guard2: on , , cmd2 {
       distribution cmd2 broadcast to my_vp;
     }
   } while (true);
   ...
}
```

Nella **input_section** sono state definite due *guardie* associate alla ricezione dei dati sugli **stream** *cmd1* e *cmd2*. Alla ricezione di *cmd1* viene distribuito in **broadcast** il dato sullo **stream** *cmd1* ad ogni processore virtuale. Ricordiamoci ovviamente che siamo in topologia **one**, quindi la **broadcast** spedisce il dato all'unico processore virtuale presente nel **parmod**.
Alla ricezione di *cmd2* viene effettuata una distribuzione dei dati sullo **stream** *cmd2* in **broadcast** ad ogni processore virtuale.
Il ciclo

```
do input_section { } while(true)
```

indica che il modulo finirà quando tutti gli **stream** in ingresso saranno chiusi. Questa terminazione viene chiamata terminazione implicita. Passiamo ora a definire la sezione **virtual_processors**.

```
parmod comando(input_stream long cmd1, long cmd2 output_stream long result) {
     topology one vp;

     do input_section {
          guard1: on , , cmd1 {
               distribution cmd1 broadcast to Pv;
          }
          guard2: on , , cmd2 {
               distribution cmd2 broadcast to Pv;
          }
     } while (true);

     virtual_processors {
          elab1 (in guard1 out result) {
               VP {
                    proc_cmd1(in cmd1 output_stream result);
               }
          }

          elab2 (in guard2 out result) {
               VP {
                    proc_cmd2 (in cmd2 output_stream result);
               }
          }
     }
     ....
}
```

Come detto precedentemente, scrivendo la sezione **virtual_processors**, forniamo l'associazione tra **stream** dal quale si è ricevuto e l'elaborazione da effettuare. Questa associazione è data dalle etichette *guard1*, *guard2*. Le associazioni permettono di discriminare comandi diversi in base alla guardia attivata.
Ora espandiamo ulteriormente il nostro parmod con il codice dell'**output_section**.

```
parmod comando(input_stream long cmd1, long cmd2 output_stream long result) {
     topology one my_vp;

     do input_section {
          guard1: on , , cmd1 {
               distribution cmd1 broadcast to my_vp;
          }
          guard2: on , , cmd2 {
               distribution cmd2 broadcast to my_vp;
          }
     } while (true);

     virtual_processors {
          elab1 (in guard1 out result) {
               VP {
                    proc_cmd1(in cmd1 output_stream result);
               }
          }
          elab2 (in guard2 out result) {
               VP {
                    proc_cmd2 (in cmd2 output_stream result);
               }
          }
     }
     output_section {
          collects result from ANY my_vp;
     }
}
```

Abbiamo cosi' scritto il nostro primo **parmod one**. Esso deve essere agganciato ad altri moduli attraverso i suoi **stream** in ingresso ed in uscita. Il **parmod one** non fornisce un modulo che si possa parallelizzare, la sua funzione è quella di poter realizzare un modulo che abbia comportamenti diversi in base allo **stream** da cui riceve. Questa possibilità in ASSIST viene definita "*nondeterminismo su stream"*.

5.3 Parmod con topologia none

Con il termine topologia *none* indichiamo che il **parmod** possiede una topologia con processori virtuali completamente anonimi. In genere questa topologia viene usata quando si ha bisogno di replicare unità di calcolo che eseguano un calcolo puramente funzionale. Questa proprietà ci permette di poter fare a meno del *naming* dei processori virtuali. In questo caso, in fase di esecuzione, non conosciamo quanti processori virtuali abbiamo né abbiamo modo per riferirci a loro. Facciamo un esempio.

Supponiamo di volere un **parmod** che esegua il fattoriale in parallelo; si può benissimo creare un **parmod none**, in quanto il fattoriale ha bisogno solo del parametro *n* che è il numero di cui si vuole il fattoriale. Una volta che il processore virtuale ha ricevuto il parametro *n* non ha bisogno di collaborare con nessun altro processore virtuale della topologia. Lo schema della nostra

applicazione si può pensare come un *farm* che riceve il numero *n* di cui bisogna calcolare il fattoriale e un raccoglitore dei risultati. Schematicamente l'applicazione si presenterebbe come in figura 7.

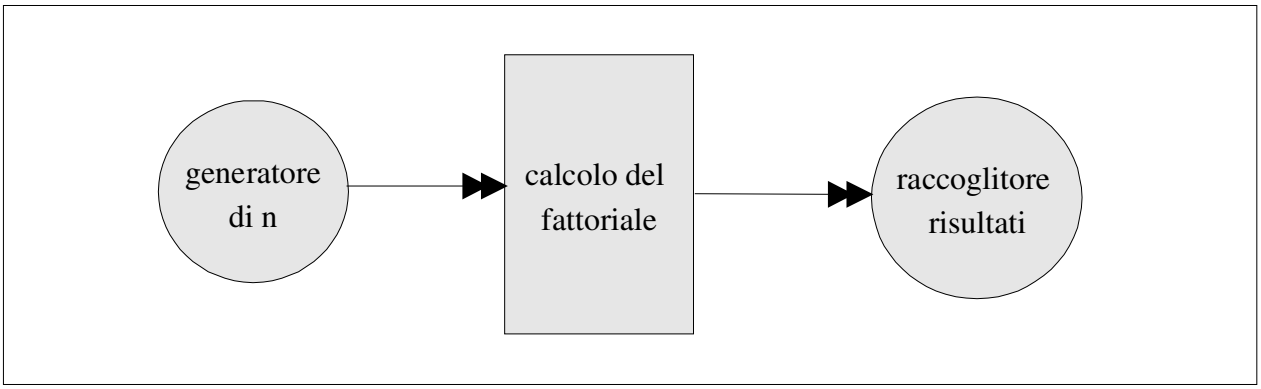


*Figura 7. grafo applicazione di esempio **parmod none***

Tralasciando per ora i due sequenziali, iniziamo a scrivere l'interfaccia del **parmod**. Il **parmod** avrà come parametro di ingresso il numero *n* di cui deve calcolare il fattoriale e *result* come parametro in uscita che rappresenta il calcolo del fattoriale. Definiamo la prima parte del nostro **parmod** insieme alla topologia.

```
parmod par_fattoriale(input_stream long n output_stream long result) {
     topology none vp;
     input_section {
     }
     virtual_processors {
     }
     output_section {
     }
}
```

Abbiamo definito le interfacce verso l'esterno ed il tipo di topologia. Affrontiamo la definizione dell'**input_section**. Sicuramente nel calcolo del nostro fattoriale non è importante quale VP si prende l'onere di effettuarlo, non a caso la topologia che abbiamo definito è di tipo **none**!
Con una topologia di tipo **none** le uniche distribuzioni permesse sono: **on_demand**, **broadcast**. In questo caso useremo la distribuzione **on_demand**, perché a noi interessa trovare un esecutore del calcolo prima possibile. Andiamo ad arricchire il nostro parmod, cercando di scrivere la sezione di ingresso.

```
parmod par_fattoriale(input_stream long n output_stream long result) {
     topology none my_vp;

     input_section {
          guard1: on , , n {
               distribution n on_demand to my_vp;
          }
     }
     virtual_processors {
     }
     output_section {
     }
}
```

La parte cerchiata è il codice che bisogna scrivere per poter creare la distribuzione **on_demand** degli elementi sullo **stream** *n.* In pratica il codice esprime il fatto che ogni volta che sullo **stream** di ingresso arriverà un dato, esso verrà distribuito (inviato) al primo processore virtuale libero. La parola *guard1* rappresenta un'etichetta arbitraria definita dal programmatore, che serve per creare l'associazione con la parte di calcolo che verrà definita nella sezione **virtual_processors**.
In questo modo ASSIST sarà in grado di creare l'associazione tra ricezione **stream** e **proc** da chiamare. Dopo la parola *guard1* la riga:

```
guard1: on , , n
```

caratterizza le regole di attivazione della guardia. In generale l'istruzione è formata da:

```
nome_guardia: on priorità, condizione, stream
```

- *nome_guardia*: nome arbitrario scelto dal programmatore
- *priorità*: rappresenta la sequenza in cui gli eventi dell'input_section verranno analizzati, di default le condizioni sugli **stream** vengono elaborate in sequenza rispettando l'ordine di scrittura. (non implementata in questa versione).
- *condizione*: rappresenta invece l'abilitazione o la disabilitazione della guardia. Il **parmod** infatti, durante l'esecuzione, può disabilitare e abilitare la ricezione su alcuni **stream**, disattivando la guardia con cui sono associati.
- *stream*: rappresenta il nome dello **stream** di ingresso, o un'espressione booleana di stream, a cui associamo questa guardia.

La riga:

```
distribution n on_demand to my_vp
```

definisce che il dato ricevuto dallo **stream** deve essere inviato al primo processore virtuale libero.
Ora che abbiamo risolto e definito il comportamento che deve avere il **parmod** quando sono presenti dati in ingresso, andiamo a definire quale **proc** deve chiamare quando il dato viene

distribuito.

```
parmod par_fattoriale(input_stream long n output_stream long result) {
     topology none my_vp;

     input_section {
          guard1: on , , n {
               distribution n on_demand to my_vp;
          }
     }

     virtual_processors {
          elab1 (in guard1 out result) {
               VP {
                    fattoriale(in n out result);
               }
          }
     }

     output_section {
     }
}
```

La parte cerchiata è il codice che dobbiamo scrivere per poter chiamare la **proc** che effettua il fattoriale. Analizziamo il codice:

- *elab1*: rappresenta una *label* arbitraria definita dall'utente. Ogni volta che vogliamo definire un tipo di elaborazione dobbiamo darle un nome. Tra parentesi viene richiamato il nome della guardia usata nella **input_section**. In questo modo abbiamo creato l'associazione tra **stream** ricevuto ed elaborazione da effettuare. Oltre alla guardia in **in** abbiamo un parametro di **out** *result*, in questo caso specifico l'uscita dalla proc corrisponde con l'uscita di tutto il parmod. Ovviamente non è obbligatorio avere un parametro di uscita, ed in tal caso si sarebbe scritto solo:

  ```
  elab1 (in guard1)
  ```

- **VP**: è una parola chiave che serve per indicare al compilatore la sezione dove descriviamo le proc da chiamare. Qui non ci soffermeremo sull'utilizzo della parola chiave **VP**, ma più avanti verrà chiarita.
- la riga:

  ```
  fattoriale(in n out result);
  ```

  rappresenta il codice per la chiamata della **proc** con i parametri di **in** e **out**.

```
parmod par_fattoriale(input_stream long n output_stream long result) {
     topology none my_vp;

     input_section {
          guard1: on , , n {
               distribution n on_demand to my_vp;
          }
     }
     virtual_processors {
          elab1 (in guard1 out result) {
               VP {
                    fattoriale(in n out result);
               }
          }
     }
     output_section {
      collects result from ANY my_vp;
     }
}
```

Abbiamo definito fino ad adesso il comportamento per la ricezione del dato, per la **proc** da chiamare per elaborarlo, manca da definire come viene spedito all'esterno il risultato.
Descriviamo ora l'**output_section**.
L'**output_section** che vediamo qui, è molto semplice.

- **collects** è una parola chiave,
- *result* rappresenta il nome dello **stream** d'uscita,
- **ANY** è un attributo della **collects** e rappresenta il tipo di collezione. Ricordiamo che esistono due tipi di collezioni **ALL** e **ANY**,
- *my_vp*: rappresenta il nome della topologia del **parmod** definita dal programmatore.

Ora che abbiamo scritto il nostro primo parmod cerchiamo di collegarlo con gli altri moduli sequenziali per creare un'applicazione completa.

```
generic main() {
    stream long d;
    stream long ris;
    invia_n        (                output_stream d);
    par_fattoriale (input_stream d   output_stream ris);
    salva          (input_stream ris);
}

invia_n(output_stream long n) {
  f_invia(output_stream n);
}

proc f_invia(output_stream long n)
inc<"stdlib.h", "stdio.h">
$c++{
  int k;
  FILE * f = fopen("filedati", "r");
  while (feof(f) == 0) {
    fread(&k, sizeof(int), 1, f);
    assist_out(n, k);
  }
  fclose(f);
}c++$

parmod par_fattoriale(input_stream long n output_stream long result) {
  topology none my_vp;

  do input_section {
     guard1: on , , n {
       distribution n on_demand to my_vp;
     }
  } while (true)

  virtual_processors {
    calcola_jacobi (in guard1 out result) {
      VP {
          fattoriale(in n out result);
      }
    }
  }
  output_section {
    collects result from ANY my_vp;
  }
}

salva (input_stream long n)
inc<"stdlib.h", "stdio.h">
$c++{
  FILE * f = fopen("dati_out", "w+");
  fwrite(&n, sizeof(int), 1, f);
}c++$

proc fattoriale(in long n out long fat)
$c++{
  fat = 1;
  for (int i=1;i<n;i++)
    fat *=i;
}c++$
```

Il **parmod** con **collects ANY** non mantiene l'ordinamento nei dati, ossia i dati inviati dal sequenziale *invia_n* non è detto che vengano mantenuti nello stesso ordine dal **parmod**. Questo è causato proprio dalla politica di collezione **ANY** associata alla politica di distribuzione **on_demand** sulla topologia **none**. A volte però è necessario rispettare un ordinamento che deve essere gestito direttamente dal programmatore. Proviamo a dare un semplice suggerimento. Sullo **stream** facciamo viaggiare non un semplice dato di tipo **long**, ma un dato strutturato costruito appositamente con un indice ed il valore vero e proprio.

Il programma precedente si modifica in questo modo: cambiare il tipo **long** con la struttura:

```
typedef struct {
      long source;
      long result;
} fat_t;
```

In questo modo, grazie alla possibilità di far viaggiare su stream dati strutturati, possiamo recuperare l'ordinamento nonostante che la politica di collezione e la topologia non lo permettevano.

Da notare come ASSIST permette l'introduzione di tipi di dato definiti dall'utente e la dichiarazione di **stream** con i nuovi tipi di dato. L'unica cosa che per adesso non è permessa sono i dati dinamicamente variabili. Non è possibile per esempio mandare una **string**, un **array** di dimensione variabile, ecc. Questo sarà un *feature*delle prossime versioni.

5.4 Parmod con topologia array

La topologia **array** permette di avere un *naming* dei processori virtuali che rappresentano l'unità di calcolo. La possibilità di avere una funzione di *naming* dei processori virtuali permette anche di poter accedere in maniera univoca ad una determinata variabile di stato. Infatti una volta decisa la topologia **array** ASSIST permette di effettuare il *mapping* di una variabile di stato su tale topologia. Questa variabile diventa, in maniera automatica, visibile in sola lettura a tutti gli altri processori virtuali. Un'altra possibilità fornitaci dal *naming* dei processori e quella di creare partizioni del calcolo. Per capire meglio questa affermazione si pensi, ad esempio, a tutti i problemi in cui si deve calcolare una funzione diversa in zone diverse di una matrice. In figura 8 viene mostrato un piccolo esempio.

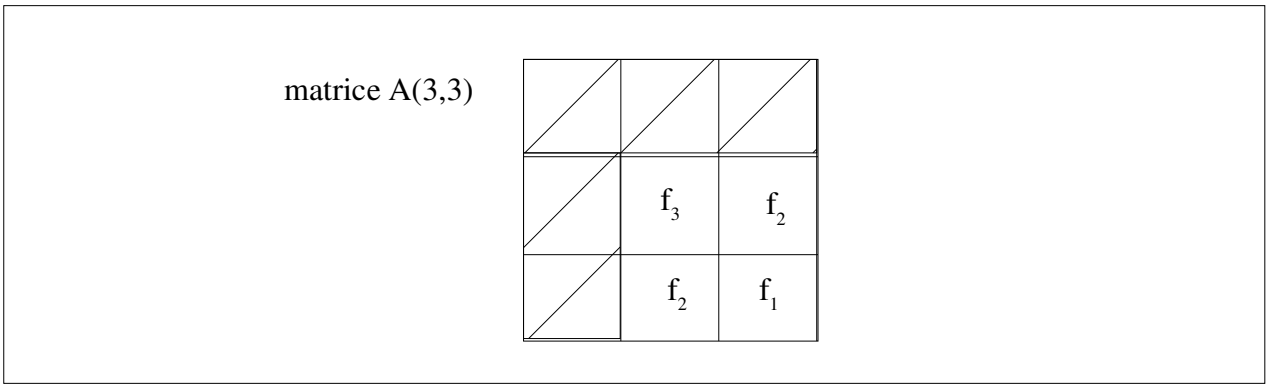


Figura 8. Applicazione di funzioni diverse su elementi diversi di una matrice

Vediamo come si possono creare in ASSIST partizioni disgiunte nell'insieme dei processori virtuali su cui eseguire funzioni di calcolo diverse. Diamo un primo anticipo per poi riprendere la sintassi su un esempio vero.

```
parmod pippo (...) {
  ...
  virtual_processors {
    elab1 (.....) {
      VP i=0..2, j=0..0 {
          nop();
      }
      VP i=0..0, j=1..2  {
          nop();
      }
      VP i=1..1, j=1..1 {
          f3();
      }
      VP i=1..1, j=2..2 {
          f2();
      }
      VP i=2..2, j=1..1 {
          f2();
      }
      VP i=2..2, j=2..2 {
          f1();
      }
  }
  ...
}
```

Da notare che *nop()* è una **proc** vuota definita dal programmatore.

Esempio:

```
proc nop()
$c++{
}c++$
```

Questo codice esprime esattamente lo schema mostrato in figura. Le ipotesi in cui si possono applicare queste regole sono:

- la dimensione della matrice deve essere statica.
- le partizioni dei processori virtuali non devono avere intersezioni.
- devono essere definite tutte le possibili partizioni.

Ma cosi' abbiamo anticipato i tempi, iniziamo per gradi a costruire il nostro **parmod** con topologia **array**.
La topologia **array** si usa in genere quando bisogna scrivere un'applicazione *data parallel*. Questo permette di ragionare rispetto ai dati che bisogna elaborare. Un vincolo forte per questa versione di ASSIST è che la matrice della topologia insieme a quella dei dati non può cambiare dinamicamente durante l'esecuzione.
Andiamo a vedere quali problemi bisogna porsi nell'utilizzo di un **parmod** a topologia **array**. Prima di tutto dobbiamo definire quali sono i dati di proprietà di ogni processore virtuale presente nella topologia. Per spiegarci meglio immaginiamo la matrice descritta prima. Era implicito nella descrizione che ogni processore virtuale era proprietario di un solo elemento della matrice. Questo tipo di associazione (dato <-> processore virtuale) viene definito dal *mapping*. Nella figura 8 si aveva un *mapping* isomorfo. Però non è sempre cosi'. Supponiamo di voler associare un'intera riga o un'intera colonna ad un processore virtuale, schiacciando quindi una dimensione. Questo schiacciamento si può effettuare in due modi:

- indiretto costruendo una struttura dati opportuna che permetta lo schiacciamento di una dimensione. Questo implica però che la singola unità visibile diventa l'intera riga o colonna e non l'elemento.
- diretto utilizzando una mapping prefisso. In questo caso si può indirizzare il singolo elemento della struttura dati.

Detto questo andiamo a definire il nostro **parmod** mostrando alcuni esempi su come si effettua il *mapping* di una variabile di stato sui processori virtuali.
Supponiamo di avere una matrice su cui vogliamo applicare qualche tipo di filtro. Si pensi per esempio ad algoritmi di elaborazione dell'immagini. La matrice *A* rappresenta i dati da elaborare , mentre la matrice *B* rappresenta la maschera degli elementi che servono per applicare la funzione su ogni singolo punto. In figura 9 viene schematizzato il tutto.

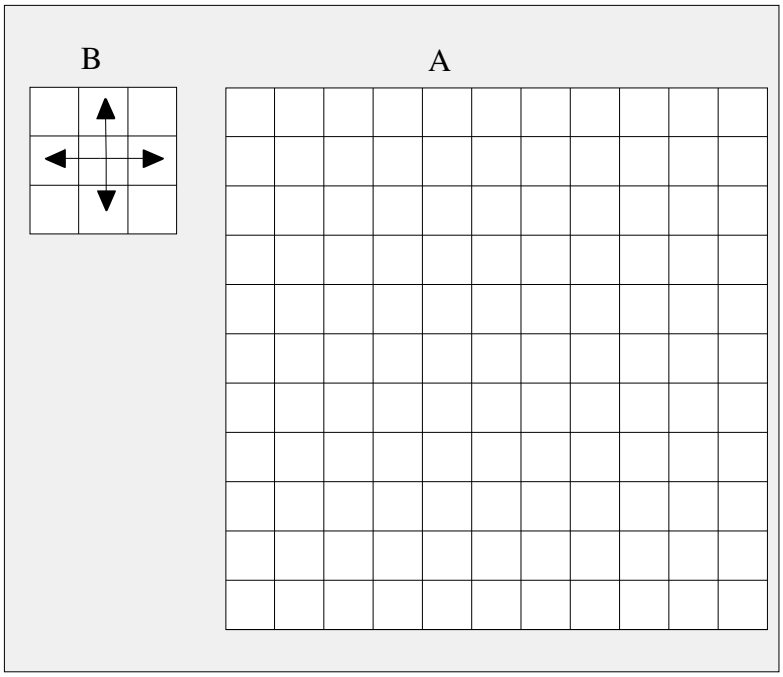


Figura 9. Rappresentazione del calcolo da effettuare su una matrice

Se dobbiamo applicare la matrice *B* per ogni punto della matrice *A*, si può pensare che il nostro processore virtuale possa essere associato ad ogni elemento della matrice. Per fare questo definiamo nel nostro **parmod** una topologia bidimensionale della dimensione della matrice ed effettuiamo un *mapping* isomorfo della matrice *A*. La matrice *A* arriverà al **parmod** da uno **stream**.
Definiamo l'interfaccia del nostro **parmod.**

```
parmod stencil(input_stream long A[N][N] output_stream long result[N][N]) {
   topology array[i:N][j:N] vp;

   attribute long S[N][N] scatter S[*i0][*j0] onto vp[i0][j0];
   ....
}
```

La parte cerchiata rappresenta la definizione di una variabile di stato che più d' essere condivisa in sola lettura da tutti i processori virtuali del **parmod**. La scrittura degli elementi di tale attributo può essere effettuata solo dal processore virtuale proprietario. Analizziamo la riga per sezioni:

- **attribute long S**[N][N] : questo definisce l'attributo di stato che vogliamo condividere.
- **scatter:** parola chiave che definisce il modo in cui viene effettuato il *mapping*.
- S[*i0][*j0] **onto** vp[i0][j0]; questa parte dell'istruzione crea l'associazione tra elemento dell'attributo *S* e la topologia *vp*. Le variabili *i0*, *j0* sono variabili arbitrarie che vengono chiamate **variabili libere** e servono solo per definire l'associazione tra gli elementi di *S* e la topologia. La variabile **i0* rappresenta una variabile che assumerà valori tra *0* e *N*, mentre la variabile *i0* rappresenta la sua istanza. La stessa regola vale per **j0.*

Se avessimo voluto creare lo schiacciamento di una dimensione in modo indiretto si sarebbe dovuto scrivere cosi':

```
typedef struct {
    long a[N];
} rig_t;

parmod pippo (...) {
  topology array[i:N] vp;
  attribute rig_t S[N] scatter S[*i0] onto vp[i0];
  ...
}
```

In modo diretto invece si sarebbe dovuto scrivere cosi':

```
parmod pippo (...) {
  topology array[i:N] vp;
  attribute long S[N][N] scatter S[*i0][] onto vp[i0];
  ...
}
```

Ritorniamo ora al nostro semplice esempio.
Una volta che è arrivata la matrice sullo **stream**, bisogna assegnarla allo stato interno del **parmod** rappresentato dall'attributo *S,* quindi bisogna decidere come distribuire i dati dello **stream** sullo stato.

```
parmod stencil(input_stream long A[N][N] output_stream long result[N][N]) {

    topology array[i:N][j:N] vp;

    attribute long S[N][N] scatter S[*i0][*j0] onto vp[i0][j0];
    stream long res;

    input_section {
        guard1: on , , A {
            distribution A[*i1][*j1] scatter to S[i1][j1];
        }
    }
    virtual_processors {
    }
}
```

Nella parte cerchiata è scritto il codice per effettuare questa distribuzione.
In questo **parmod** abbiamo associato ogni elemento della matrice ad un processore virtuale, questo implica come detto prima che l'unico che può scrivere nell'elemento generico A[i,j] è vp[i,j].
La definizione :

```
stream long res;
```

definisce uno **stream** interno per poter effettuare la **collects ALL** nell'**output_section**. Questo è

necessario in quanto come si vede, l'uscita generata dal processore virtuale non è della stessa dimensione dello **stream** di interfaccia di uscita del **parmod**.

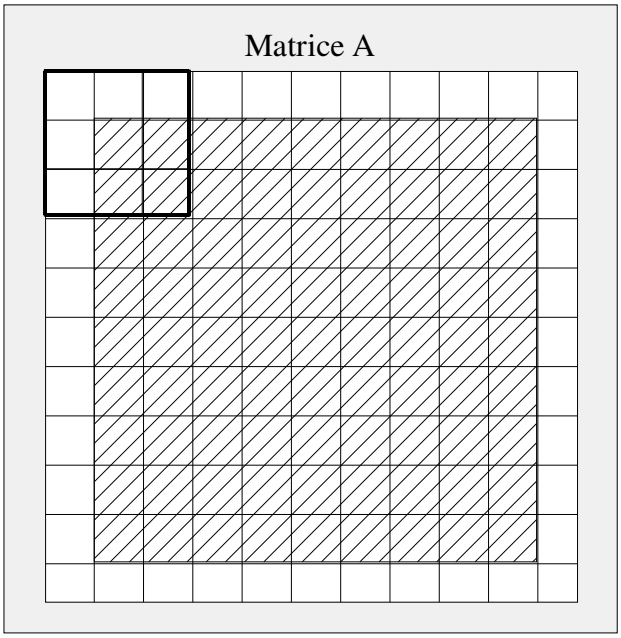


Figura 10 Schema di calcolo

Applicare la matrice B significa che i processori virtuali del confine non devono effettuare nessun calcolo, ma sono solo dei contenitori di dati. In figura 10 viene mostrato il confine dei processori virtuali che effettueranno il calcolo. Sicuramente nel nostro **parmod** dobbiamo definire dei sottoinsiemi di processori virtuali per discriminare quelli che effettueranno dei calcoli. Nella parte **virtual_processors** dobbiamo definire in che modo i nostri processori virtuali elaboreranno la matrice. Sicuramente tutti i processori virtuali della prima riga, ovvero i processori virtuali di coordinate [0,k], [N,k] e quelli di coordinate [k,0] [k,N] con 0<= k< N non effettueranno nessun calcolo, mentre quelli interni, per calcolare il loro elemento, avranno bisogno di avere tutti i valori rappresentati dalla matrice *B*.
Un processore virtuale di coordinate *i, j* avrà bisogno dei seguenti elementi per eseguire il proprio calcolo:

```
A[i-1,j], A[i,j-1], A[i, j+1], A[i+1,j]
```

La nostra **proc** avrà in entrata nove parametri, in uscita un unico parametro risultato.

```
proc calcola(in long me, long up, long sx, long dx, long down out long res)
$c++{
          res = me + (up + sx + dx + down) / 8;
}c++$
```

Ora associamo il calcolo all'attivazione, ricordando che i processori virtuali che non partecipano al

calcolo devono comunque spedire il valore a loro assegnato. I processori virtuali dei bordi si limiteranno ad eseguire un forward del valore della matrice a loro assegnato.

```
parmod pippo (...) {
  ...
  virtual_processors {
    VP i=0..0, j=0..N-1 {
         assist_out(res,S[i][j]);
    }
    VP i=N-1..N-1, j=0..N-1 {
         assist_out(res,S[i][j]);
    }
    VP i=1..N-2, j=N-1..N-1 {
         assist_out(res,S[i][j]);
    }
    VP i=1..N-2, j=1..N-2 {
         assist_out(res,S[i][j]);
    }
    VP i=1..N-2, j=1..N-2 {
         calcola (in S[i][j], S[i-1][j], S[i][j-1], S[i][j+1], S[i+1][j]
                  out S[i][j]);
         assist_out(res,S[i][j]);
  }
  ...
}
```

Mostriamo ora la parte **output_section**

```
parmod pippo (...output_stream long B[N][N]) {
  ...
  output_section {
    collects res from ALL Pv[i][j] {
      long el;
      long localB[N][N];
      AST_FOR_EACH(el) {
        localB[i][j] = el;
      }
      assist_out (B, localB);
    }<>;
  }
  ...
}
```

Questo codice serve ad implementare una **collects ALL**, in questa sezione uno può anche decidere di applicare un'ulteriore funzione sugli elementi che riceverà prima di spedire la matrice su **stream**. Notiamo che la **collects** lavora sullo **stream** interno *res.* Per il motivo citato prima, la matrice di uscita *B* è la collezione dello **stream** interno *res*.

Quando si utilizza la **collects ALL**, i parametri di uscita della proc associata possono essere solo di tipo **out.** Di conseguenza non può essere utilizzato il costrutto **assist_out**. Questi vincoli non sono però controllati, attualmente a tempo di compilazione, ma si ottiene un comportamento indefinito al momento della esecuzione del programma ASSIST.

Di seguito verrano mostrati due esempi. Il primo mostra l'utilizzo della distribuzione sullo stato. Il

secondo la distribuzione sia sullo stato sia sui processori virtuali.

Esempio (uso dello stencil)

```
#define N 10

generic main() {
  stream long [N][N] d;
  stream long [N][N] h;
  invia (output_stream d);
  stencil (input_stream d output_stream h);
  salva (input_stream  h);
}


invia (output_stream long n[N][N]) {
  f_invia (output_stream n);
}

proc f_invia (output_stream long n[N][N])
$c++{
  long ntmp[N][N];

  for(unsigned int i=0; i<N;i++)
    for(unsigned int j=0;j<N;j++)
      ntmp[i][j] = i*j;

  assist_out (n, ntmp);
}c++$

salva (input_stream long n[N][N])
inc<"stdlib.h", "stdio.h">
$c++{
        FILE *f;
        f = fopen("risultato", "w+");
        for(unsigned int i=0; i<N;i++)
          for(unsigned int j=0;j<N;j++)
            fwrite(&(n[i][j]), sizeof(int), 1, f);
        fclose(f);
}c++$


proc calcola (in long me, long up, long sx, long dx, long down
              out long res)
$c++{
    res = me + (up + sx + dx + down) / 8;
}c++$
```

```
parmod stencil (input_stream long A[N][N] output_stream long result[N][N]) {
  topology array[i:N][j:N] Pv;
  attribute long S[N][N] scatter S[*i0][*j0] onto Pv[i0][j0];
  stream long res;

 do input_section {
    guard1: on, , A {
      distribution A[*i1][*j1] scatter to S[i1][j1];
    }
  } while (true)

   virtual_processors {
    elab (in guard1 out res) {
```

Esempio (uso dello stencil)

```
      VP i=0..0, j=0..N-1 {
       assist_out(res, S[i][j]);
    }
    VP i=N-1..N-1, j=0..N-1 {
       assist_out(res, S[i][j]);
    }
    VP i=1..N-2, j=N-1..N-1 {
       assist_out(res, S[i][j]);
    }
    VP i=1..N-2, j=1..N-2 {
       assist_out(res, S[i][j]);
    }

      VP i=1..N-2,j=1..N-2 {
       calcola (in S[i][j], S[i-1][j], S[i][j-1], S[i][j+1], S[i+1][j]
                 out S[i][j]);
       assist_out(res, S[i][j]);
      }
    }
  }

  output_section {
    collects res from ALL Pv[i][j] {
      int el;
      int local_result[N][N];
      AST_FOR_EACH(el) {
        local_result[i][j]=el;
      }
      assist_out(result, local_result);
    }<>;
  }
}
```

Esempio 2 (moltiplicazione matrici)

```
  #define N 4
  #define M 4
  #define L 4

  generic main() {
    stream long[N][M] Matrix1;
    stream long[M][L] Matrix2;
    stream long[N][L] Matrix_ris;
    genera1           (output_stream Matrix1);
    genera2           (output_stream Matrix2);
    prodotto_matrici (input_stream Matrix1, Matrix2  output_stream
       Matrix_ris);
    fine              (input_stream Matrix_ris);
  }

  genera1(output_stream long Matrix1[N][M]) {
    fgen1(output_stream Matrix1);
  }

  proc fgen1(output_stream long Matrix1[N][M])
  $c++{
```

Esempio 2 (moltiplicazione matrici)

```
  long a[N][M];
  for (int i=0;i<N;i++)
    for (int j=0;j<M;j++)
      a[i][j] = i;
  assist_out(Matrix1, a);
}c++$

genera2(output_stream long Matrix2[N][M]) {
  fgen2(output_stream Matrix2);
}

proc fgen2(output_stream long Matrix2[M][L])
$c++{
  long a[M][L];
  for (int i=0;i<M;i++)
    for (int j=0;j<L;j++)
      if (i==j) a[i][j] = 1;
      else a[i][j] = 0;
  assist_out(Matrix2,a);
}c++$

fine(input_stream long Matrix_ris[N][L])
inc<"iostream">
$c++{
  int ok = 0;
  // verifica del risultato
  for (int i=0;i<N;i++)
    for (int j=0;j<M;j++)
      if (Matrix_ris[i][j] != i) ok = -1;

}c++$
```

```
parmod prodotto_matrici (input_stream long Matrix1[N][M], long Matrix2[M][L]
                         output_stream long Zion[N][L]) {

  topology array [i:N][j:L] morpheus;
  attribute long tank[N][M] scatter neo[*ia][*ja] onto  morpheus [ia][ja];
  attribute long neo[M][L] scatter neo[*ib][*jb] onto  morpheus [ib][jb];
  stream long trinity;
  do input_section {
    guard1: on , , Matrix1 && Matrix2 {
      distribution Matrix1[*i0][*j0] scatter to tank[i0][j0];
      distribution Matrix2[*i1][*j1] scatter to neo[i1][j1];
    }
  } while (true)


  virtual_processors {
    elab1 (in guard1 out trinity) {
      VP i, j {
        f_mul (in Matrix1[i][], neo[][j] out trinity);
      }
    }
  }

  output_section {
    collects trinity from ALL morpheus[i][j] {
      int elem;
      int local_Matrix[N][L];
```

Esempio 2 (moltiplicazione matrici)

```
      AST_FOR_EACH(elem) {
        Matrix_ris_[i][j]=elem;
      }
      assist_out(Zion, local_Matrix);
    }<>;
  }
}

proc f_mul(in long A[M], long B[M] out long C)
inc<"iostream">
$c++{
  register long r=0;
  for (register int k=0; k<M; ++k)
    r += A[k]*B[k];
  C = r;
}c++$
```

Nell'esempio *2,* il codice dell'**input_section** del **parmod** *prodotto_matrici*

```
guard1: on , , Matrix1 && Matrix2
```

esprime il fatto che l'elaborazione elab1 verrà eseguita solo quando saranno disponibili i dati sugli **stream** *Matrix1*, *Matrix2*

Quando si lavora con un attributo partizionato a volte può essere necessario eseguire più **proc** sul singolo attributo, ma l'esecuzione di una **proc** deve poter vedere le modifiche effettuate dalla prima **proc**. Per questo caso ASSIST mette a disposizione l'utilizzo di una parola chiave **sync**. La parola chiave **sync** serve per sincronizzare lo stato del **parmod**. Dopo la parola **sync** ASSIST assicura che lo stato del **parmod** visibile nelle **proc** successive alla **sync** sarà aggiornato con tutte le modifiche effettuate dalle **proc** precedenti. Cerchiamo di spiegarci meglio con un esempio.

Esempio 3 (utilizzo delle sync)

```
parmod more_proc(input_stream long A[N][N] output_stream long result[N][N]) {
   topology array[i:N][j:N] vp;
   attribute long S[N][N] scatter S[*i0][*j0] onto vp[i0][j0];
   stream long res;
   input_section {
      guard1: on , , A {
         distribution A[*i1][*j1] scatter to S[i1][j1];
      }
   }

   virtual_processors {
     elab0(in guard1 out res) {
      VP i,j {
         filtro1(in S[i][j], S[j][i] out S[i][j]);
         sync;
         filtro2(in S[i][j], S[j][i] out S[i][j]);
         assist_out(res, S[i][j]);
      }
   }
  }

  output_section {
      collects res from ALL vp[i][j] {
         long el;
         long B[N][N];
         AST_FOR_EACH(el) {
         B[i][j] = el;
         }
         assist_out (result, B);
      }<>;
  }
}
```

Nell'esempio sopra i valori con cui la **proc** *filtro2* lavorerà saranno modificati dalla **proc** *filtro1*. Se avessimo scritto un codice di questo tipo:

```
  virtual_processors {
      VP i,j {
         filtro1(in S[i][j], S[j][i] out S[i][j]);
         filtro2(in S[i][j], S[j][i] out S[i][j]);
         assist_out(res, A[i][j]);
      }
  }
```

L'esecuzione di *filtro2* avrebbe lavorato sugli stessi valori di *filtro1* e i valori modificati visibili sarebbero stati solo quelli effettuati da *filtro2*.
Questi esempi possono essere utilizzati per algoritmi che non lavorano su matrici di grosse dimensioni. Per matrici di grosse dimensioni gli stessi esempi si possono scrivere utilizzando la memoria condivisa fornita da ASSIST che permette di non sovraccaricare le sezioni di **input_section** e **output_section**.

6. Oggetti esterni in ASSIST

L'introduzione di una memoria condivisa o di oggetti esterni accedibili dai vari moduli ASSIST è un altro dei punti forti. La memoria condivisa esiste ortogonalmente ai moduli e permette di creare uno stato comune a tutta l'applicazione o di creare oggetti che possono essere utilizzabili da piu' moduli o di avere uno spazio di memoria per allocazioni dinamiche. Ci sono diversi tipi di memoria condivisa differenziate soprattutto per il loro utilizzo:

- variabili **shared**
- libreria **Reference**
- libreria **Shared_tree**
- Oggetti CORBA

6.1 Variabili shared

Gli oggetti **shared** permettono di estendere il concetto di stato del modulo parallelo e risultano utili in particolare quando le strutture dati sono troppo grandi per essere passate sugli stream. La sincronizzazione su questi oggetti è a carico del programmatore.
Una variabile shared viene dichiarata fuori da ogni modulo, con la seguente sintassi:

```
shared long A[10];
```

Con questa riga di codice si definisce una variabile condivisa visibile da tutti i moduli paralleli dell'applicazione ASSIST. Per utilizzare la variabile A all'interno dei moduli basta utilizzare il nome della definizione. Le variabili shared non sono visibili e quindi utilizzabili nei moduli sequenziali.
Una volta definito, l'attributo **shared** può essere utilizzato nell'interfaccia di una **proc** come un attributo partizionato. Di seguito mostriamo un esempio sull'uso di questo tipo di memoria esterna.

Nell'esempio per inizializzare i valori delle due variabili **shared** viene utilizzato un parmod one, esattamente il **parmod** *crea_shared.* Questa scelta è obbligatoria per la limitazione sul sequenziale.
Il **parmod** *somma_shared,* invece, rappresenta il modulo che esegue la somma in parallelo accedendo alle variabili shared inizializzate dal **parmod** *crea_shared.*
Notiamo anche che per garantire che il **parmod** *somma_shared* esegua la somma quando il **parmod** *crea_shared* ha finito l'inizializzazione viene utilizzato lo **stream** *pronto*. Questo perché ASSIST non inserisce nessuna limitazione sulla lettura/scrittura delle variabili **shared**. Qualsiasi semantica è possibile ed è a carico del programmatore.

Esempio di uso di una variabile shared

```
// -*- C++ -*-

#define N 10
```

Esempio di uso di una variabile shared

```
shared long A[N];
shared long B[N];

generic main() {
  stream long pronto;
  stream long[N] C;
  crea_shared   (output_stream pronto);
  somma_shared  (input_stream pronto output_stream C);
  stampa        (input_stream C);
}

parmod crea_shared (output_stream long pronto) {

  topology one Pv;

  do input_section {
    guard1: on , , {
    }
  } while (false)

  virtual_processors {
    elab_crea_s (in guard1 out pronto) {
      VP {
        crea_s (out A[], B[], pronto);
      }
    }
  }

  output_section {
    collects pronto from ANY Pv;
  }
}


proc crea_s (out long A[N], long B[N], long pronto)
inc<"iostream">
$c++{
  /* inizializzazione dei vettori condivisi */
  for (int i=0; i<N; i++) {
    A[i] = i;
    B[i] = N-i;
  }
  pronto=1;
}c++$


stampa (input_stream long C[N])
inc<"iostream">
$c++{

  int ok = 0;
  // verifica del risultato
  for (int i=0; i<N; i++)
    if (C[i] != (i + (N-i))) ok = -1;

}c++$
```

```
parmod somma_shared (input_stream long pronto output_stream long C[N]) {
```

Esempio di uso di una variabile shared

```
  topology array [i:N] Pv;
  stream long ris;
    do input_section {
    guard1: on , , pronto {
      distribution pronto broadcast to Pv;
    }
  } while (true)

  virtual_processors {
    elab_somma_s (in guard1) {
      VP i {
        somma_s (in A[i], B[i] out ris);
      }
    }
  }

  output_section {
    collects ris from ALL Pv[i] {
      int elem;
      int C_[N];
      AST_FOR_EACH(elem) {
        C_[i] = elem;
      }
      assist_out(C, C_);
    }<>;
  }
}


proc somma_s (in long a, long b  out long ris)
inc<"iostream">
$c++{
  ris = a + b;
}c++$
```

6.2 Libreria Reference

La libreria dei Reference è una libreria che realizza l'astrazione di una memoria in cui l'allocazione di buffer può avvenire dinamicamente. L'accesso ai buffer allocati avviene attraverso un riferimento opaco. La libreria mette a disposizione dell'utente primitive di creazione, scrittura e lettura dell'intero buffer o parti di questo, lock e unlock, e infine di distruzione.
I moduli di un programma ASSIST possono condividere l'area di memoria allocata semplicemente scambiandosi il riferimento attraverso gli **stream**. Questo risulta molto utile quando si trattano problemi dinamici e con strutture dati irregolari e di grandi dimensioni. In ASSIST non esiste nessuna limitazione sull'uso dei Reference, possono essere utilizzati tanto nei parmod quanto nei sequenziali. Di seguito mostriamo un esempio di uso della libreria: il primo modulo crea il buffer (`REF_New`) e ci scrive, il secondo modulo, una volta ricevuto il riferimento, legge e stampa a video i valori letti.

Esempio di uso della libreria Reference)

```
// -*- C++ -*-

#define N 10
#define K 1

generic main() {
  stream ref_t A_rif;
  genera_A     (output_stream A_rif);
  end          (input_stream A_rif);
}

genera_A (output_stream ref_t A_rif) {
  proc_genera_A (output_stream A_rif);
}

proc proc_genera_A (output_stream ref_t A_rif)
inc<"iostream">
$c++{
  int displ;
  int dim = N*N*sizeof(long);  // dimensione area allocata con i reference
  int size = N*sizeof(long);
  REF_Reference_t ref_matrice;
  long riga_A[N];

  /* allocazione dell'area in memoria condivisa */
  REF_New (&ref_matrice, dim, REF_Default, 0);


  for (int i=0; i<N; i++) {
    for(int j=0; j<N; j++)  riga_A[j] = j*i;
    displ = i * N * sizeof(long);


    /* scrittura in memoria condivisa della riga i-esima */
    REF_D_Write (&ref_matrice, riga_A, displ, size);

  }
  assist_out (A_rif, ref_matrice);
}c++$

end (input_stream ref_t C2_rif)
inc<"iostream">
$c++{
  long riga_C[N];
  int displ;
  int size = N * sizeof(long);
  int ok = 0;

  for(int i=0; i<N; i++) {
    displ = i * N * sizeof(long);


  /* lettura della riga i-esima dalla memoria condivisa */
    REF_D_Read (&(C2_rif), riga_C, displ, size);


    for(int j=0; j<N; j++)
      if (riga_C[j] != j*i) ok =-1;
```

Esempio di uso della libreria Reference)

```
    REF_delete (&C2_rif);
  }
}c++$
```

Le parte cerchiate sono le funzioni proprie della libreria che permettono di interagire con la memoria di tipo Reference.

6.3 Libreria Shared_tree

Un'altra libreria integrata nell'ambiente ASSIST è quella degli **Shared_Tree** . Questa libreria permette ad un'applicazione parallela di condividere istanze della struttura dati albero. L'interfaccia verso l'utente è costituita da un oggetto opaco (di tipo ShT_Tree) e da un insieme di librerie. L'oggetto *ShT_Tree* che riferisce i nodi di un albero condiviso è una struttura locale, e quindi la condivisione avviene tramite la comunicazione di questi oggetti.

Esempio di uso della libreria Shared_Tree

```
#define N_NODI 3

generic main() {
  stream tree_t      t;
  stream long    somma;
  crea_albero   (                 output_stream t);
  visita_one    (input_stream t   output_stream somma);
  end           (input_stream somma);
}

crea_albero (output_stream tree_t t) {
  crea_a (output_stream t);
}

proc crea_a (output_stream tree_t t)
inc<"iostream">
$c++{
  int numFigli = 0;
  int databuf = 0;
  ShT_Tree * vetFigli = NULL;
  ShT_Tree * padre = NULL;
  ShT_Tree t_tmp;
  ShT_Tree figlio_dx, figlio_sx;

  /* creazione dei nodi dell'albero */
  ShT_Create (&t_tmp, &databuf, sizeof(databuf), padre, vetFigli, numFigli);
  databuf++;
  ShT_Create(&figlio_dx,&databuf,sizeof(databuf),&t_tmp,vetFigli,numFigli);
  databuf++;
  ShT_Create(&figlio_sx,&databuf,sizeof(databuf),&t_tmp,vetFigli,numFigli);

  /* inserimento dei nodi nell'albero di radice t_tmp */
  ShT_SetNumFigli (&t_tmp, 2);
  ShT_SetFiglio (&t_tmp, 0, &figlio_sx);
  ShT_SetFiglio (&t_tmp, 1, &figlio_dx);
  assist_out (t, t_tmp);
```

Esempio di uso della libreria Shared_Tree

```
}c++$

parmod visita_one (input_stream tree_t t  output_stream long somma) {
  topology one Pv;
  do input_section {
    guard1: on , , t {
      distribution t broadcast to Pv;
    }
  } while (true)

  virtual_processors {
    elab_visita (in guard1 out somma) {
      VP {
        Felab_visita (in t output_stream somma);
      }
    }
  }
  output_section {
    collects somma from ANY Pv;
  }
}
```

```
proc Felab_visita (in tree_t t  output_stream long somma)
inc<"iostream">
$c++{
  int somma_tmp = 0;
  int databuf;
  int numFigli;
  ShT_Tree figlio;
  ShT_GetNumFigli (&t, &numFigli);

  /* pezzo su cui fare la ricorsione */
  for(int i=0; i<numFigli; i++)
    {
      ShT_GetFiglio (&t, i, &figlio);
      ShT_GetInfo (&figlio, &databuf);
      somma_tmp += databuf;
    }
  ShT_GetInfo (&t, &databuf);
  somma_tmp += databuf;

  assist_out (somma, somma_tmp);
}c++$

end (input_stream long somma)
inc<"iostream">
$c++{
  int ris = 0;

  // verifica del risultato
  for (int i=0; i<N_NODI; i++)
    ris +=i;
}c++$
```

6.4 Uso di oggetti remoti tramite CORBA

L'ultimo tipo di oggetti esterni disponibili per il programmatore ASSIST sono gli oggetti invocabili tramite interfacce CORBA. Un oggetto CORBA può essere invocato all'interno delle procedure sequenziali (**proc**) di ASSIST e non ci sono particolari costrutti o parole chiavi da utilizzare all'interno del codice del programma. I passi da eseguire per il funzionamento corretto sono gli stessi che si sarebbero dovuti eseguire all'interno di un programma sequenziale c++ dove si vuole utilizzare un oggetto CORBA.
Come primo passo bisogna procurarsi un ORB. L'ORB (Object Request Broker), elemento fondamentale dell'intera architettura di CORBA, è il canale di comunicazione degli oggetti nell'ambiente distribuito. Gli esempi riportati in questo tutorial utilizzano l'ORB TAO fornito con la libreria ACE sulla quale ASSIST si appoggia. Naturalmente ciò non è obbligatorio e qualsiasi altro può essere utilizzato. La libreria con la specifica dell'ORB, nel nostro caso TAO, va inserita nel file di configurazione descritto nel paragrafo successivo.

- Nelle proc dove si vuole utilizzare l'oggetto remoto devono essere inclusi i file generati dalla compilazione del file **.idl** dove c' è la specifica delle funzionalità dell'oggetto. Solitamente il nome dei file è lo stesso di quello .IDL, ma cambia l'estensione. Ad esempio se il file è ***time.idl*** riportato nella tabella sottostante, allora i file generati che devono essere inclusi sono ***timeC.cpp***, ***timeS.cpp.***

Esempio:

Esempio Interfaccia IDL

```
struct TimeOfDay {

  short hour;       // 0 - 23
  short minute;     // 0 - 59
  short second;     // 0 - 59
};

interface Time {
  TimeOfDay get_gmt ();   // metodo esportato
};
```

Esempio di file inclusi

```
seq_start (...)
path<"/home/pippo/corba_assist">
inc<"timeC.h", "iomanip.h">
....
```

- Ci dobbiamo procurare lo IOR che definisce l'oggetto del quale vogliamo invocare le funzionalità. Lo IOR è una stringa che contiene tutte le informazione per identificare in un ambiente distribuito l'oggetto remoto. Lo IOR può essere recuperato in diversi modi,

nell'esempio riportato nell'esempio sottostante viene scambiato attraverso il file system condiviso. Il sistema più elegante è l'utilizzo del naming service fornito dall'architettura CORBA. Il naming service è un repository in cui i server pubblicano i propri oggetti con un nome mnemonico, la chiave del repository, mediante la quale i client possono accedere e farsi restituire lo IOR.

```
$c++{
      // lettura IOR da file
      ifstream f;
      char str[601];
      while (f.open("file_ior"),!f) sleep (1);
      f.getline(str, 601);
      f.close();
....
```

- Ottenere a partire dallo IOR il riferimento all'oggetto su cui chiamare i metodi pubblicati.

```
...
   // Initialize orb
   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

   // Destringify
   CORBA::Object_var obj = orb->string_to_object (str);

   // Narrow
   Time_var tm = Time::_narrow (obj.in ());

   // Validating connection
   CORBA::PolicyList_var pols;
   tm->_validate_connection (pols.out ());
```

Invocare il metodo dell'oggetto sul riferimento all'oggetto.

```
....
   // Get time
   TimeOfDay tod = tm->get_gmt ();

   cout << "Time in Greenwich is "
   << setw (2) << setfill ('0') << tod.hour << ":"
   << setw (2) << setfill ('0') << tod.minute << ":"
   << setw (2) << setfill ('0') << tod.second << endl;
 }
....
....
}c++$
```

Bisogna fare attenzione al fatto che una **proc** ASSIST può essere invocata più volte, mentre alcuni dei passi precedentemente descritti devono essere eseguiti una sola volta, per ragioni di efficienza o di correttezza. Per questo motivo, suggeriamo di definire alcune variabili e metodi utilizzati come ***static***.

Di seguito riportiamo un esempio del caso in cui la procedura (seq_loop) viene invocata più di una volta e quindi alcune delle variabili necessarie al funzionamento di ASSIST con CORBA sono state definite statiche. L'oggetto pubblicato attraverso CORBA è un' interfaccia grafica che permette la visualizzazione del risultato restituito da ASSIST.
L'esempio ASSIST, non completamente riportato qui sotto, è l'n-body esatto. I dati da elaborare vengono passati ad ASSIST attraverso il metodo esportato dall'oggetto CORBA ***body_t get_dataset()*** mentre la visualizzazione avviene invocando il metodo ***oneway void display (in body_t B)***.

Esempio:

Esempio GUI.idl

```
#define N 11

struct Tdata_ {
  double m;
  double x;
  double y;
  double vx;
  double vy;
};

struct Tbody_ {
  Tdata_ el[N];
};

typedef Tbody_ body_t[N];

interface GUI {
  body_t get_dataset();
  oneway void display (in body_t B);
};
```

n-body

```
/* n_body con interfaccia grafica connessa tramite corba */

typedef struct {

  double m;   // massa
  double x;   // posizione sulle ascisse
  double y;   // posizione sulle ordinate
  double vx;  // velocita' sulle ascisse
  double vy;  // velocita' sulle ordinate
} Tdata;      // body

typedef struct {
  Tdata el[N];
} Tbody;

generic main() {
   stream Tbody[N] A;
   stream Tbody[N] B;
```

n-body

```
   genera    (output_stream A);
   n_body_r  (input_stream A output_stream B);
   seq_loop  (input_stream B output_stream A);
}

proc proc_loop(in Tbody A[N] output_stream Tbody B[N])
path</home/pippo/corba/n_body">
inc<"GUIC.cpp","cmath","iostream","cstdio", "unistd.h">
$c++{

  static long conta=0;
  static int argc = 1;
  static char * argv[]= {"nome",0};
  static CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
  static GUI_var * gui=0;

....
....

}c++$
```

6.5 ASSIST come oggetto CORBA

Un programma ASSIST può essere incapsulato in un oggetto CORBA ed integrato in una più grande applicazione usando i meccanismi standard di invocazione CORBA. Esistono due modi distinti per far interconnettere un programma ASSIST ad un'applicazione distribuita. Le due modalità affrontano due diverse classi di problemi e sono:

1. Invocazione sincrona (**RMI-like**)**:** si realizza mediante l'invocazione del metodo esportato dall'oggetto CORBA con parametri in ingresso e ritorno dei risultati. Questa modalità è utile quando i task che devono essere eseguiti in parallelo sono ben definiti ed isolati.
2. Invocazione asincrona (**stream-like data-passing**): si realizza mediante il meccanismo degli event-channel di CORBA. Questo meccanismo risulta utile ad esempio quando i dati da produrre sono molti e quindi la comunicazione può essere sovrapposta al calcolo, o quando le strutture dati sono troppo grandi per essere comunicate all'applicazione in un' unica volta.

Un programma ASSIST, per essere esportato come oggetto CORBA, deve essere una composizione di moduli, nel caso particolare anche un unico modulo, nelle quali un input stream ed un output stream sono disconnessi e quindi scelti come punti di ingresso e di uscita della componente esportata. Inoltre nel caso **RMI-like** si ha l'ulteriore vincolo che per ogni dato ricevuto in ingresso si deve produrre uno ed un solo dato in uscita.
Il processo di esportazione del programma ASSIST attraverso CORBA è stato automatizzato: il programma che rispetti i vincoli descritti sopra viene sottoposto ad una prima fase di analisi e poi viene trasformato aggiungendo il codice necessario all'interazione con il run-time di CORBA. Anche le interfacce IDL vengono generate in modo automatico a partire dal programma.

6.5.1 Invocazione sincrona del metodo

Nel caso di invocazione sincrona un programma ASSIST è esportato come un oggetto con un singolo metodo **`execute`** con argomenti e tipi di ritorno uguali rispettivamente al tipo dell'input e dell'output stream del programma iniziale. La figura 11 mostra lo schema del programma ASSIST arricchito con le parti necessarie all'interazione con CORBA.

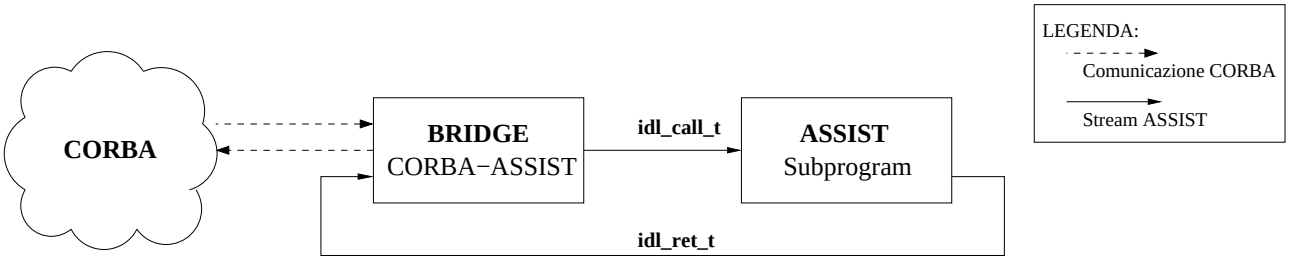


Figura 11 Grafo programma ASSIST nel caso di invocazione sincrona

Nella figura sottostante invece riportiamo il codice con l'interfaccia e le definizioni dei tipi del programma ASSIST e dello IDL generato a partire da quest'ultimo.

Codice ASSIST	**Interfaccia IDL generata**

`#define N 20`
 	
`generic main() {`
 	
`  stream long[N][N] Aaa;`
`  stream long[N][N] Bbb;`
`  spt( input_stream Aaa`
`  output_stream Bbb );`
`}`
 	
`parmod spt (input_stream long A [N][N]`
`  output_stream long B [N][N])`
`{`
`  //parallel code`
`}`
	`#define N 20`
	`#define bool boolean`
	
	`typedef long idl_ret_t[N][N];`
	`typedef long idl_call_t[N][N];`
	
	`interface sptG {`
	`  idl_ret_t execute(in idl_call_t`
	`  _var);`
	`};`

Per l'utilizzo del tool automatico, denominato ***assist2corba*** bisogna eseguire i seguenti passi.

1. *$ASTCC_ROOT/ASSIST2CORBA/analizzatore/assist2corba nomefile.ast nomefileG.ast*, per lanciare l'analizzatore del programma utente e creare i file necessari al funzionamento con CORBA Il primo parametro è il nome del file ASSIST utente di partenza, il secondo è il nome del file ASSIST che verrà generato in automatico.
2. *$TAO_ROOT/TAO_IDL/tao_idl nomefileG.idl,* per la compilazione del file IDL generato automaticamente.
3. *astCC -c nomefileG.ast,* per la compilazione del programma ASSIST in cui è stato inserito il

codice per la gestione di CORBA.

Prima di lanciare gli eseguibili ASSIST, deve essere attivo il servizio di naming. Noi abbiamo scelto di utilizzare il *File System* per comunicare lo IOR che identifica il servizio di naming; il file creato dall'attivazione si chiama *ns.ior*. Per attivare il servizio si deve eseguire il seguente comando:

- $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -o ns.ior

A questo punto è possibile lanciare il server ASSIST. Nel caso si utilizzi la ***clam*** per lanciare il programma ASSIST, il file *ns.ior* si deve trovare (eventualmente copiare se il comando è stato lanciato in una directory diversa) nella directory in cui viene mandata in esecuzione la ***clam*** stessa. Altrimenti, se gli eseguibili ASSIST vengono lanciati senza ***clam***, il file *ns.ior* si deve trovare all'interno della directory in cui si lanciano gli eseguibili.

6.5.2 Interconnessione attraverso event-channel

Per esprimere pienamente la computazione *stream-parallel,* abbiamo sviluppato un secondo meccanismo basato sugli event channel di CORBA per implementare le comunicazioni asincrone.
Gli eventi CORBA possono avere un tipo qualsiasi, e la loro ricezione è governata dal tipo, perciò abbiamo scelto di incapsulare il dato in una struttura "nominata" con un solo campo. per semplificare la discriminazione dei messaggi in ingresso ed in uscita. La figura 12 mostra la struttura del programma ASSIST finale ottenuta dall'applicazione del tool *assist2corba* ad un sottoprogramma ASSIST iniziale.

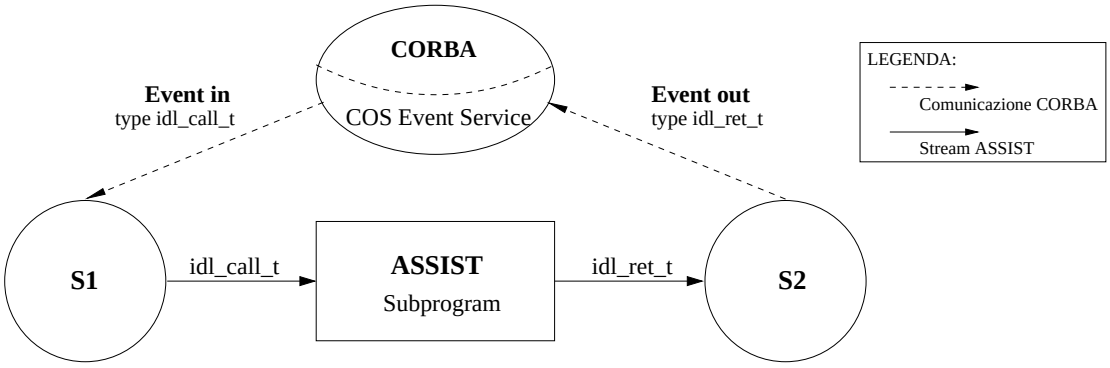


Figura 12 Grafo ASSIST nel caso di invocazione asincrona

Per l'utilizzo del tool in questo caso i passi da seguire sono:

1. *$ASTCC_ROOT/ASSIST2CORBA/analizzatore/assist2corba nomefile.ast nomefileG.ast*

nomeventoingresso nomeventouscita, per lanciare l'analizzatore del programma utente e creare i file necessari al funzionamento con CORBA. Il primo parametro è il nome del file ASSIST utente di partenza, il secondo è il nome del file ASSIST che verrà generato in automatico, mentre *nomeventoingresso* e *nomeventouscita* sono i nomi che verranno dati rispettivamente agli eventi di input e di output. è la presenza di questi due ultimi parametri che discrimina il caso con o senza eventi.

2. *$TAO_ROOT/TAO_IDL/tao_idl nomefileG.idl,* per la compilazione del file IDL generato automaticamente.
3. *ln -s $ASTCC_ROOT/ASSIST2CORBA/codice_event/Ast_Event_** . Per creare nella directory di lavoro un link ai file *Ast_Event_Consumer.cpp, Ast_Event_Consumer.h*, *Ast_Event_Producer.cpp, Ast_Event_Producer.h* che si trovano nella directory ASSIST2CORBA*/codice_event.*
4. *astCC -c nomefileG.ast,* per la compilazione del programma ASSIST in cui è stato inserito il codice per la gestione di CORBA.

Prima di lanciare gli eseguibili ASSIST, deve essere attivo il servizio di naming e il servizio degli eventi. Noi abbiamo scelto di utilizzare il *File System* per comunicare lo IOR che identifica il servizio di naming; il file creato dall'attivazione si chiama *ns.ior*. Per attivare i due servizi basta lanciare lo script

- *$ASTCC_ROOT/ASSIST2CORBA/script/namingeventservice*

oppure se si vogliono lanciare singolarmente, i due comandi da eseguire sono:

1. $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -o ns.ior
2. $TAO_ROOT/orbsvcs/CosEvent_Service/CosEvent_Service
   -ORBInitRef NameService=file://ns.ior

Il secondo comando deve essere lanciato nella stessa directory in cui si trova il file ns.ior creato dall'esecuzione del primo comando.
A questo punto è possibile lanciare il server ASSIST. Nel caso si utilizzi la ***clam*** per lanciare il programma ASSIST, il file *ns.ior* si deve trovare (eventualmente copiare se il comando è stato lanciato in una directory diversa) nella directory in cui viene mandata in esecuzione la ***clam*** stessa. Altrimenti, se gli eseguibili ASSIST vengono lanciati senza ***clam***, il file *ns.ior* si deve trovare all'interno della directory in cui si lanciano gli eseguibili.
Una volta lanciato, il server resta attivo in attesa di eventi da gestire. Per far terminare correttamente il server, si deve eseguire il comando

- *$ASTCC_ROOT/ASSIST2CORBA/terminazione/termina*
  -ORBInitRef NameService=file://ns.ior

Il servizio degli eventi e il servizio di naming restano attivi.

6.5.2.1Generazione e compilazione mediante script

Per semplificare il processo di generazione e compilazione del programma ASSIST incapsulato in CORBA abbiamo scritto uno script che esegue i passi descritti sopra (fino alla compilazione di ASSIST compresa). Lo script deve essere eseguito nella directory di lavoro e discerne il caso da eseguire (sincrono/asincrono) dal numero di parametri che gli vengono passati. Lo script si trova nella directory *ASSIST2CORBA/script.*
Il comando da eseguire e', nel caso di invocazione sincrona

- *$ASTCC_ROOT/ASSIST2CORBA/script/creacorba.sh nomefile.ast nomefileG.ast*

mentre nel caso di invocazione asincrona

- *$ASTCC_ROOT/ASSIST2CORBA/script/creacorba.sh nomefile.ast nomefileG.ast eventoingresso eventouscita*

A questo punto il programma ASSIST incapsulato in un oggetto CORBA è pronto per essere mandato in esecuzione come descritto nella parte finale dei paragrafi precedenti.

7. Utilizzo di ASSIST

7.1 Installazione ASSIST

L'installazione può essere fatta da un qualsiasi utente, tuttavia è consigliabile effettuare un'unica installazione come root in una directory visibile dai vari utenti, in seguito definita INSTALLAZIONE (ad esempio */usr/local)*.
Per la comunicazione tra i processi ASSIST e' possibile utilizzare la libreria di comunicazione ACE o la libreria di comunicazione GLOBUS. In fase di configurazione e' possibile scegliere se installare il supporto per un'unica libreria di comunicazione o entrambe. Non esistono dipendenze tra il supporto alla libreria ACE e il supporto alla libreria GLOBUS.

7.1.1 Prerequisiti

Prima di installare ASSIST e' necessario installare i seguenti prerequisiti:

- ACE 5.4.4 o superiore http://www.cs.wustl.edu/~schmidt/ACE.html
  e / o
- GLOBUS Toolkit 4.0.3 http://www.globus.org

- libxml2 -2.6 o superiore (dovrebbe essere presente nella release linux)

- SWIG 1.3 o superiore http://www.swig.org/download.html
- Xerces 2_6 o superiore http://xml.apache.org/xerces-c/download.cgi
- Java jdk http://java.sun.com
- ANT 1.5.4 o superiore http://ant.apache.org/
- gperf, autoconf, automake

7.1.1.1 Fedora Core 5

Per la release di Linux Fedora Core 5 e' possibile installare tutti i prequisiti tramite RPM. I comandi devono essere eseguiti come utente root. Al momento non e' presente un rpm per il GLOBUS Toolkit relativo a Fedora core 5.

ACE

- Scaricare dal sito http://dist.bonsai.com/ken/ace_tao_rpm/#RPM i pacchetti:
  - ace-5.5.4-1.FC5.i386.rpm
  - ace-devel-5.5.4-1.FC5.i386.rpm

  situati nella apposita sezione di Fedora Core 5 / i386/ 5.5.4-1
- Installare i pacchetti con i comandi seguenti:
  - `=> rpm --import http://dist.bonsai.com/ken/RPM-GPG-KEY.ken.txt`
  - `=> rpm --install ace-devel-5.5.4-1.FC5.i386.rpm ace-5.5.4-1.FC5.i386.rpm`

libxml2

- Digitare il comando:
  - `=> yum install libxml2 libxml2-devel`

Swig

- Digitare il comando:
  - `=> yum install swig`

Xerces

- Digitare il comando:
  - `=> yum install xerxes-c xerces-c-doc xerces-c-devel`

Ant

- Digitare il comando:
  - `=> yum install ant ant-javadoc ant-manual`

gperf

- Digitare il comando:
  - `=> yum install gperf`

Java

- Digitare il comando:
  - `=> yum install jdk`
- Una volta installato java 1.5 e' necessario verificare che il sistema lo riconosca:
  - `=> alternatives --display java`
- Nel caso non lo riconosca con i comandi seguenti viene aggiunto alla lista e viene definito

quale utilizzare tra quelli disponibili:

- `=> alternatives --install /usr/bin/java java /usr/java/jdk1.5.0_10/bin/java 2`
- `=> alternatives --config java`

- Le eventuali variabili d'ambiente da settare sono:

  BASH

  - `=> export J2RE_HOME=/usr/java/jdk1.5.0_10`
  - `=> export PATH=$J2RE_HOME/bin:$PATH`

  TCSH

  - `=> setenv J2RE_HOME /usr/java/jdk1.5.0_10`
  - `=> setenv PATH $J2RE_HOME/bin:$PATH`

7.1.2 Operazioni preliminari

- Decomprimere il pacchetto astCC1_3.tgz nella directory INSTALLAZIONE

```
(BASH):andromeda [/usr/local/ASSIST]
=> tar xzvf astCC1_3.tgz
```

- Settare le variabili d'ambiente.
  Modificare il file di configurazione ast_env.sh (ast_env.csh per shell di tipo tcsh) inserendo i path corretti rispetto all'installazione sulla macchina locale.
  Eseguire tramite shell il comando:
  `=> source ast_env.sh` (oppure `ast_env.csh`)

- (SOLO x GLOBUS) Nel caso di installazione del supporto alle librerie GLOBUS e' necessario eseguire i seguenti comandi.

  `=> cd INSTALLAZIONE/astCC1_3/include/Globuslib3.2`

  (es. `=> cd /usr/local/ASSIST/astCC1_3/include/Globuslib3.2`)

  ```
  => globus-makefile-header globus_io globus_common --flavor=gcc32dbgpthr >
  globus_definitions
  ```

7.1.3 Compilazione libHOCUTIL

- Posizionarsi nella directory INSTALLAZIONE/astCC1_3/ADHOC

  `=> cd INSTALLAZIONE/astCC1_3/ADHOC`

  (es. `=> cd /usr/local/ASSIST/astCC1_3/ADHOC`)

- Eseguire il configure di ADHOC

  `=> ./configure --enable-assist-setup --with-ext-execute-object=INSTALLAZIONE /astCC1_3/streamlib3/libsrc/libCComm.a`

  (es. `=> ./configure --enable-assist-setup --with-ext-execute-object=$HOME/ASSIST/astCC1_3/streamlib3/libsrc/libCComm.a`)

- Posizionarsi nella directory INSTALLAZIONE/astCC1_3/ADHOC/hoc

```
=> cd hoc
```

- Eseguire il make
```
=> make libhocutil.a
```

7.1.4 Compilazione streamlib3

- Posizionarsi nella directory INSTALLAZIONE/astCC1_3/streamlib3
```
=> cd INSTALLAZIONE/astCC1_3/streamlib3
```
(es. `=> cd /usr/local/ASSIST/astCC1_3/streamlib3`)

- Eseguire l'autogen
```
=> ./autogen.sh
```

- Eseguire il configure
```
=> ./configure
```

- Eseguire il make
```
=> make
```

7.1.5 Compilazione astCC1_3

- Posizionarsi nella directory INSTALLAZIONE/astCC1_3
```
=> cd INSTALLAZIONE/astCC1_3
```
(es. `=> cd /usr/local/ASSIST/astCC1_3`)

- Eseguire aclocal
```
=> aclocal
```

- Eseguire automake
```
=> automake
```

- Eseguire autoconf
```
=> autoconf
```

- Eseguire il configure

Come default la libreria di comunicazione e' ACE. Utilizzando l'opzione --enable-globus viene compilato anche il supporto alla libreria di comunicazione GLOBUS.
Con l'opzione --enable-noace non viene compilato il supporto per la libreria di comunicazione ACE.

  - Verificare le installazioni dei prerequisiti ACE e libXML2

Se ACE e libXML2 sono stati installati correttamente utilizzando il comando "make install" devono essere presenti le seguenti directory:

$ACE_ROOT/lib
$ACE_ROOT/include
$LIBXML2_ROOT/lib
$LIBXML2_ROOT/include

Se questo e' vero eseguire il configure semplice altrimenti il configure specificando i prefix necessari

- Configure semplice
  `=> ./configure --prefix=INSTALLAZIONE/astCC1_3/`
  (es. `=> ./configure --prefix=$HOME/ASSIST/astCC1_3/`)

- Configure specificando i prefix necessari
  Digitare:
  `=> ./configure --help`
  per visualizzare i prefix disponibili
  (es. specificando una directory diversa per gli include di ACE
  `=> ./configure -prefix=/usr/local/ASSIST/astCC1_3/ --with-ACE-lib-prefix=$HOME/pippo`

- Eseguire il make
  `=> make`

- Copia degli eseguibili
  `=> make install`

7.1.6 Compilazione ADHOC

- Posizionarsi nella directory INSTALLAZIONE/astCC1_3/ADHOC
  `=> cd INSTALLAZIONE/astCC1_3/ADHOC`
  (es. `=> cd /usr/local/ASSIST/astCC1_3/ADHOC`)

- Eseguire il make
  `=> make`

7.2 Configurazione di ASSIST

Per configurare ASSIST è possibile modificare il file di configurazione. Un file di configurazione standard e' presente nella directory di INSTALLAZIONE/astcc1_3 con il nome di ast_rc. Come default il compilatore ASSIST cerca il file di configurazione .ast_rc posizionato in $HOME, è, però, possibile specificare per una certa compilazione un file diverso da quello di default, in modo da inserire tutte le configurazioni relative ad un progetto in un file ad hoc. Le più importanti

impostazioni che possono essere modificate sono:

Sequenza di TAG prefisso	*Sintassi del TAG*	*Spiegazione*
	<compiledir name="/home/capraia/maginis/compileAssist" />	Directory in cui vengono generati i sorgenti e gli eseguibili
<libraries>	<lib path="/usr/local/lib" file="rit" />	Path e nome di librerie da linkare ai binari generati da ASSIST
<includes>	<include path="${ACE_ROOT}" />	Directories in cui si vogliono cercare i file di include di supporto
<sysincs>	<sysinc path="/usr/include/c++/3.2.2/" />	Directories in cui si vogliono cericare i file di include di sistema
<machines>	<machine name="andromeda" addr="131.114.2.104" kind="smp"/>	Lista delle macchine su cui i processi possono essere allocati
<debflags><flags>	<flag value="-g -Wall -pedantic"/>	Flag con cui viene invocato il compilatore in caso di compilazione con debug
<optflags><flags>	<flag value="-O"/>	Flag con cui viene invocato il compilatore in caso di compilazione ottimizzata

7.3 Compilazione di un programma ASSIST

Il comando ***astCC -h*** , eseguito da *shell,* permette di avere la lista delle opzioni del compilatore.

Opzioni compilatore	
-I <path>	add an include path
-c <file_src>	compile a source file
-o <file_out>	specify the output file
-f <rc_file>	specify the Assist resource file
-g	generate Globus Makefile

Opzioni compilatore	
-h	print this help
-L	use DVSA-based Reference
-i	incremental compile
-k	keep generated rts files
-t	generate intermediate code
-T <format>	Trace execution <xcel/csv>
-b	don't build binaries
-d	debug
-D	dynamic don't generate <loading> section
-G	generate GNU makefile
-X	generate XDR code
-x <file>	generate IDL code
-P	disable profile
-O [level]	optimization level 0..3
-S [complex]	max complexity to swap optimizing
-U [u-num]	unroll loop optimization
-M	makeflags more flags to be passed to make (e.g. -j2)

Per la compilazione *standard* del nostro primo programma ASSIST si deve eseguire il seguente comando:

```
astCC -c esempio.ast
```

7.4 Esecuzione di un programma ASSIST

Per l'esecuzione di un programma ASSIST si veda la documentazione del Loader.