Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Skeleton di componenti master-slave per la parallelizzazione di moduli legacy

A. Machì, F. Collura

RT-ICAR-PA-05-02 **Marzo 2005**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Skeleton di componenti master-slave per la parallelizzazione di moduli legacy

A. Machì, F. Collura

CNR/MIUR Legge 449/97 (5% 1999)
Piattaforma abilitante complessa ad oggetti distribuiti e ad alte prestazioni

Task 1 : Ambiente di programmazione portabile a componenti parallele
Livello L1: Supporti ad Alte prestazioni

Rapporto Tecnico N.: **RT-ICAR-PA-05-02**

Data: **Marzo 2005**

Indice

1. INTRODUZIONE

Obiettivi generali dell'attività nell'anno conclusivo del Progetto "***MIUR 5% - PIATTAFORMA DISTRIBUITA AD ALTE PRESTAZIONI***, *Task A: Ambiente di programmazione portabile a componenti parallele*" erano la realizzazione della versione Beta dell'ambiente, con benchmarking finale e valutazione delle prestazioni su casi di studio di particolare complessità. In particolare per il Gruppo ICAR di Palermo erano previste la reingegnerizzazione del progetto esecutivo di skeleton master-slave con capacità di autobilanciamento del carico in funzione di indici di performance delle risorse computazionali su cui è mappato il grafo di esecuzione parallela, sviluppato nell'anno precedente, e la effettuazione di benchmark delle sue prestazioni.

La reingegnerizzazione ha effettivamente riguardato le classi di controllo dello skeleton per permettere la parallelizzazione, secondo il modello d'integrazione di moduli enucleati da codice legacy, sviluppato sinergicamente dallo stesso gruppo nel progetto MIUR FIRB Grid.it.[19].

Infatti, nella integrazione a partire da codice oggetto di moduli applicativi legacy originariamente sviluppati secondo un paradigma di programmazione procedurale, è in generale difficile enucleare elementi software pre-strutturati cui attribuire ruoli di coordinamenti. In particolare i ruoli di producer, di emitter, collector e consumer che sono tipicamente utilizzati negli skeleton di programmazione parallela nativa, e nei linguaggi di coordinamento per applicazioni distribuite quali ASSIST basati sul modello dataflow.

I kernel computazionali si presentano, invece, generalmente più strutturati dentro il modulo, e sono facilmente identificabili, intercettabili ed importabili agevolmente tramite wrapping nei processi worker.

Lo skeleton effettivamente implementato nel secondo anno del progetto unifica i ruoli di emitter/collector (nella versione map) e di producer/consumer (nella versione farm) e simula un meccanismo di RPC parallelo fra master e slave. Si elimina così il ricorso a memoria condivisa esterna al modulo parallelo, generalmente richiesto nel codice utente per la riconduzione al paradigma dataflow del controllo del codice legacy procedurale.

Le prestazioni dello skeleton sono state valutate attraverso estensivi test di performance parallela su un modulo applicativo di image processing specializzato

nella individuazione di irregolarità (difetti) in una sequenza di immagini, particolarmente complesso e parallelizzabile al massimo al 50%.

L'overhead imposto dallo skeleton, per il controllo e la manipolazione dei dati, è risultato inferiore al 5% del tempo di esecuzione, su tutto il data set di sequenze di immagini tipiche assegnate come input nelle prove.

Per le prove di verifica delle capacità di bilanciamento del carico sono stati utilizzati diversi indici di potenza nominale statica (bogomips, dhrystone,whetstones etc), il più adeguato dei quali si è rivelato essere l'indice Bogomips. Lo skeleton è stato in grado di bilanciare efficacemente il carico di lavoro dei workers, mantenendo la oscillazione del loro tempo di esecuzione entro il 10%.

Lo speed-up medio ottenuto è stato pari al 96% del valore massimo atteso.

2. Modello d'integrazione di Codice Legacy

L'integrazione in applicazioni di calcolo ad alte prestazioni di codice generico non nativamente sviluppato in un particolare framework di riferimento, richiede una metodologia rigorosa che, a partire da elementi software di varia natura (codice sorgente, codice oggetto, eseguibili) e realizzati in vari linguaggi di programmazione (C, C++, Fortran), permetta di creare un componente software ospitabile nel framework di riferimento.

Nell'ambito delle attività del Progetto MIUR FIRB Grid.it, l'Unità di ricerca ICAR-CNR di Palermo sta sviluppando un ambiente software CAE che, in maniera automatica e/o assistita, segua l'utente durante il processo d'integrazione [19]. La soluzione adottata si basa sulla definizione rigorosa dei ruoli e delle caratteristiche degli elementi software (kernel, moduli) da utilizzare come risultati intermedi e/o finali del processo di integrazione. L'obiettivo è quello di pervenire alla definizione di un modello di meta-componente controllabile e (ri)configurabile i cui gli elementi costitutivi (kernel, moduli) siano, direttamente o tramite adattamento e/o sfoltimento, importabili dal Codice Legacy di partenza.

Il meta-componente è il punto di partenza per l'integrazione del Codice Legacy nell'applicazione desiderata e rappresenta un'astrazione del componente integrato, da

completare con il supporto applicativo (adapters) e architetturale (ports) del framework di riferimento.

2.1 Elementi software del processo d' integrazione

Il Codice Legacy, prima di essere integrato, deve essere ricondotto in uno dei seguenti elementi:

- kernel
- modulo

Un ***kernel*** è un'unità atomica di computazione "in memory" senza stato interno, privo di operazioni di I/O dati ma con meccanismi di output testuale formattato. Un kernel non emette eccezioni, né esplicitamente né implicitamente, e termina il suo flusso d'esecuzione esattamente al ritorno dalla sua chiamata di attivazione.

Un kernel presenta e s'identifica in un unico metodo di chiamata. Tale metodo ha la seguente interfaccia:

```
extern "C" int kernel-call(param-list);
```

ove *kernel-call* rappresenta il nome della chiamata e *param-list* è la lista dei parametri richiesti.

Un **modulo** è un'unità d'elaborazione applicativa, ossia è orientato al soddisfacimento di un requisito funzionale di una determinata applicazione o classe d'applicazioni, che può effettuare operazioni di I/O su risorse di memorizzazione esterne o può richiedere funzionalità esterne.

Un modulo presenta uno o più **metodi di chiamata**, ciascuno con la seguente interfaccia:

```
int module-call(param-list)
```

dove *module-call* rappresenta il nome della chiamata del modulo e *param-list* è la lista dei parametri richiesti.

Kernel e Moduli sono organizzati in un meta-componente secondo l'architettura evidenziata in figura 1:

Il meta-componente è costituito da due moduli principali: un modulo applicativo, ottenuto dal processo di integrazione del Codice Legacy, con interfacce specifiche di servizio e di utilizzo di memoria applicativa e d'ambiente, ed un modulo di controllo, con interfacce di introspezione e ad eventi.
Il modulo applicativo contiene internamente i kernel computazionali, mentre il modulo di controllo mantiene e gestisce i *relevant-data*, ossia i dati di controllo condivisi tra il modulo applicativo ed il Workflow dell'applicazione complessiva, attraverso lo scambio di eventi.

Il componente implementa le interfacce secondo il framework di riferimento, attraverso l'utilizzo di opportune ports.

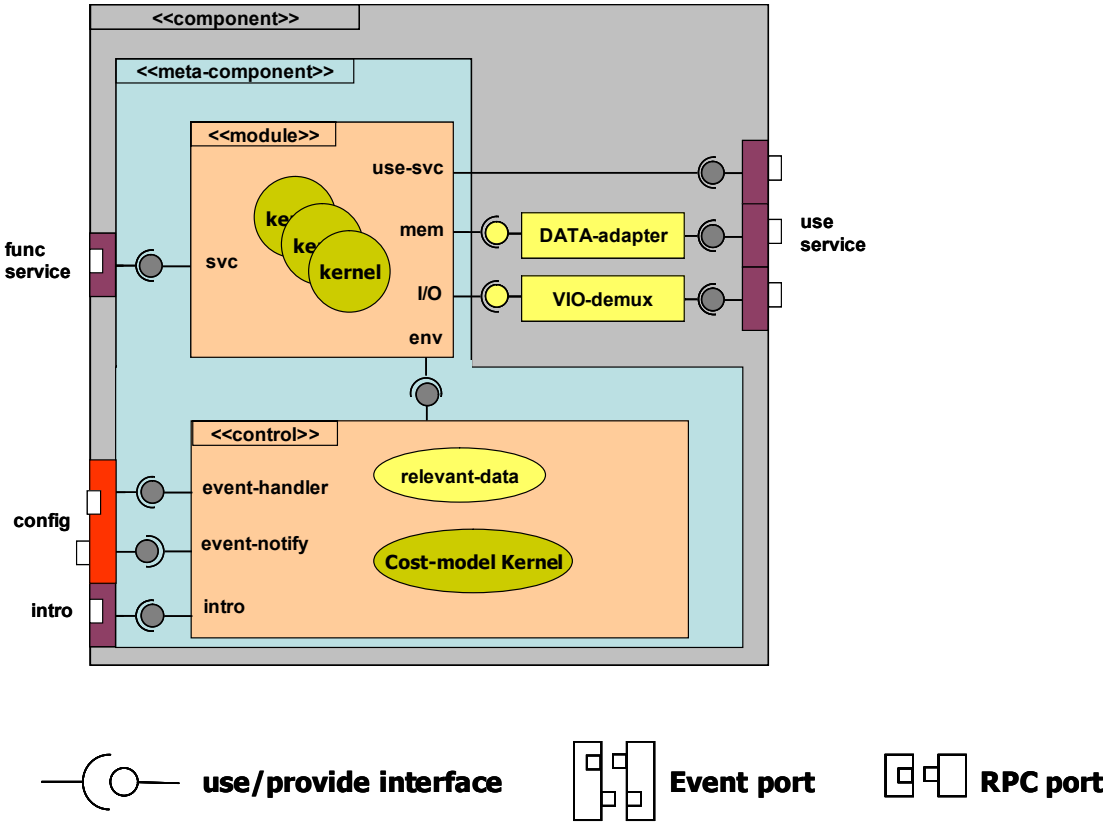


Fig. 1 Architettura di un meta-componente integrato da codice legacy.

3. Parallelizzazione di moduli legacy

In questa sezione è presentata una metodologia per una possibile parallelizzazione di moduli computazionali provenienti da codice legacy. Tale metodologia è limitata nell'utilizzo ad alcuni casi particolari, evidenziati nel contesto, ed è applicabile sia a

partire da codice sorgente che da codice oggetto. Alcuni tools di parallelizzazione semi-automatica a partire da codice oggetto sono in via di sviluppo.

Per modulo computazionale s'intende nel seguito un modulo *"stateless"* che fornisce metodi per elaborare dei dati applicativi, recuperando e mantenendo i risultati su risorse di memorizzazione esterne.

L'obiettivo è quello di ottenere un modulo parallelo che distribuisca parte dell'attività del modulo originario su più entità d'elaborazione utilizzando un semplice modello di parallelizzazione strutturata master-slave.

Il modulo parallelo sarà la parte funzionale di un meta-componente parallelo, ossia di un meta-componente caratterizzato da un grafo virtuale di processamento in cui un nodo Master delega alcune parti di elaborazione ai più nodi Slave in maniera concorrente.Tale modello deriva da un approccio di programmazione parallela strutturata di tipo stream-parallel e data-parallel [1,2] in cui i nodi Slave hanno mansioni di Worker, mentre il nodo Master ha mansioni di Producer/Consumer o Emitter/Collector inscindibili.

La metodologia indicata prevede due diverse strategie di parallelizzazione:

- distribuzione del carico computazionale, ossia l'utilizzo dei nodi Slave per l'elaborazione concorrente di alcune parti prettamente di calcolo del modulo legacy. Tale strategia è denominata "Parallelizzazione del Kernel"
- distribuzione del carico complessivo, ossia l'utilizzo dei nodi Slave per l'elaborazione concorrente di tutte la parti del modulo legacy. Tale strategia è denominata "Parallelizzazione del Modulo"

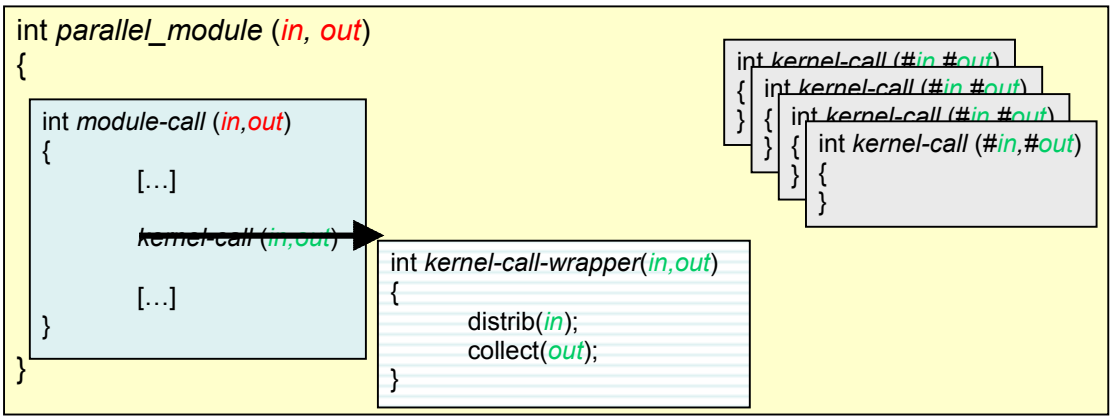
3.1 Parallelizzazione del kernel

Tale metodologia si applica ai moduli computazionali costituiti da uno o più kernel parallelizzabili nei dati d'ingresso.L'approccio è di tipo data-parallel. Per fissare le idee, nel seguito è presentato uno scenario di parallelizzazione di un modulo uni-kernel:

```
int module-call (in,out)
{
        […]

        kernel-call (in,out)

        […]
}
```

Il modulo presenta un metodo di chiamata *module-call* con un insieme di parametri d'ingresso *in* ed un insieme di parametri d'uscita *out*. Il modulo ha un'interfaccia use verso un kernel computazione *kernel-call*, con parametri d'ingresso *in* e parametri d'uscita *out*, dipendenti direttamente o indirettamenti dai parametri del modulo.

La parallelizzazione del kernel prevede il wrapping della chiamata al kernel stesso per l'esecuzione differita sui Worker con partizionamento dei dati. In altre parole s'introduce un kernel di wrapping nell'esecuzione del modulo sull'Emitter/Collector e si esegue il kernel legacy sui Worker, previo partizionamento dei dati.



Il modulo parallelo sarà costituito dal modulo legacy , dal kernel legacy e dal kernel di wrapping. Durante l'esecuzione sull'Emitter/Collector il modulo legacy

chiamerà il *kernel-call-wrapper* che si preoccuperà della distribuzione e successiva raccolta dei dati originariamente in ingresso al kernel legacy (*in,out*). Durante l'esecuzione sui Worker, ciascun kernel legacy elaborerà una porzione dei dati originari (*#in,#out*).

3.2 Parallelizzazione del modulo

Tale metodologia si applica ai moduli computazionali costituiti da un ciclo di iterazione su uno o più kernel non parallelizzabili nei dati d'ingresso. L'approccio è di tipo stream-parallel in cui lo stream coincide con il loop d'iterazione sui kernel. Per fissare le idee, nel seguito è presentato uno scenario di parallelizzazione di un modulo uni-kernel:

```
int module-call (in,out)
{
        […]
        while ( c )
        {
                […]
                kernel-call (in,out)
                […]
        }
        […]
}
```

Il modulo presenta un metodo di chiamata *module-call* con un insieme di parametri d'ingresso *in* ed un insieme di parametri d'uscita *out*. Il modulo ha internamente un loop iterativo non banale sulla chiamata al kernel, di condizione $c=c(i,j)$ direttamente o indirettamente dipendente dai parametri d'ingresso (*in*), dove $i=i(n)$ è la variabile di controllo del ciclo e $j=j(n)$ ne è la condizione di arresto. Inoltre

$$i(n) = f(i(n-1), in), \quad i_o = f_o(in)$$
$$j(n) = g(j(n-1), in), \quad j_o = g_o(in)$$

ossia, la variabile di controllo del ciclo e la relativa condizione d'arresto dipendono dai rispettivi valori al ciclo precedente, eccezion fatta per la prima iterazione.
Sotto tali ipotesi:

$$c(i,j) = true, \quad se\ i \neq j$$
$$c(i,j) = false, \quad se\ i = j$$

Infine, il modulo ha un'interfaccia use verso un kernel computazione *kernel-call*, con parametri d'ingresso *in* e parametri d'uscita *out*, dipendenti direttamente o indirettamenti dai parametri del modulo e dall'iterazione corrente.

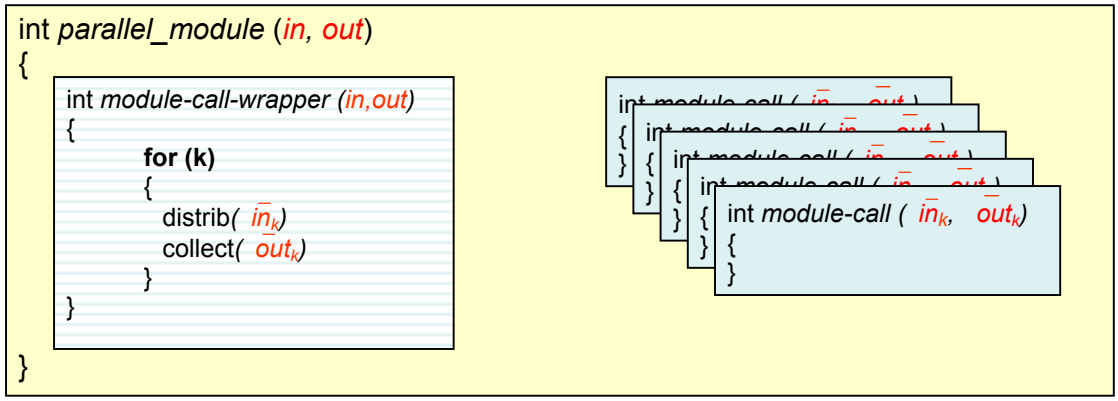
La parallelizzazione del modulo prevede la generazione di uno stream di dati mappabile con le iterazioni effettuate dal modulo sul kernel legacy e l'esecuzione sui Worker di più istanze dell'intero modulo su ciascun dato dello stream. In altre parole si tenta di eseguire sui Worker, in maniera concorrente, le diverse iterazioni che originariamente erano eseguite nel modulo legacy.

Ovviamente ciò è possibile solo se:

- è possibile intervenire sui parametri di chiamata per effettuare una sola iterazione alla volta sul kernel legacy

$$\exists\ \bar{in}:\ \bar{i}_o\ !=\ \bar{j}_o,\ e\ \ \bar{i}(n)\ =\ \bar{j}(n)\ \ per\ n > 0$$

- l'effetto complessivo delle singole chiamate $out_1(\ \bar{in}_1)\ + out_2(\ \bar{in}_2)\ + \ldots$ sia equivalente, dal punto di vista applicativo, all'unica chiamata originaria *out(in)*



Il modulo parallelo sarà costituito dal modulo legacy e da un modulo di wrapping. Durante l'esecuzione sul Producer/Consumer il modulo di wrapping si preoccuperà della generazione dello stream di dati corrispondenti al ciclo iterativo del modulo legacy, ed alla distribuzione e successiva raccolta di ciascun dato. Durante l'esecuzione sui Worker, ciascun modulo legacy elaborerà il dato dello stream corrispondente ad un'unica iterazione sulla chiamata al kernel legacy.

4. Template di Skeleton riconfigurabili Farm e Map

In questa sezione sono presentati in dettaglio i template per la parallelizzazione di codice legacy secondo le due strategie discusse in precedenza:

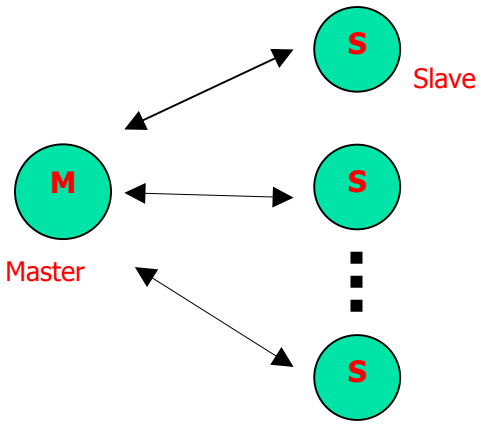
- parallelizzazione del kernel
- parallelizzazione del modulo

Entrambi i template sono finalizzati alla creazione di moduli paralleli basati su schemi notevoli di programmazione parallela strutturata, rispettivamente Map e Farm.

Il template per la parallelizzazione del kernel modella una versione ristretta del paradigma di parallelizzazione master-slave di dati compositi (data-parallel).

Il template per la parallelizzazione del modulo modella una versione ristretta del paradigma di parallelizzazione master-slave di un flusso di task (stream parallel).

Lo schema Master-Slave implementato dai template prevede un grafo virtuale composto da un processo Master e da più processi Slave, la cui cardinalità è modificabile all'attivazione.



I processi non godono di memoria condivisa. Lo scambio di dati tra processi avviene tramite un opportuno sistema di comunicazione orientato all'inter-scambio di messaggi (quale MPI) con un insieme ridotto di primitive. Il set di primitive permettono la attivazione dei processi, operazioni di trasmissione/ricezione punto punto (send-receive asincrona) e sincronizzazione collettiva (barrier).

Il mapping fra processi virtuali e nodi fisici avviene al momento di attivazione del componente parallelo che implementa il costrutto (configurazione) sulla base di considerazioni di ottimizzazione globale delle risorse computazionali disponibili al momento.

La potenza efficace disponibile su ogni nodo del grafo di mapping e la capacità dei canali di collegamento è conosciuta a tale momento e supposta costante per l'intera durata di esecuzione del task. Tale informazione è fornita al componente parallelo insieme agli altri parametri di attivazione.

4.1 Template Map

Il template Map è utilizzato per la creazione di un modulo parallelo secondo una strategia di parallelizzazione del kernel, a partire da un modulo legacy seriale. Tale template può essere utilizzato come guida alla parallelizzazione a partire da codice sorgente, inserendo il codice legacy nelle appropriate sezioni e specificando inoltre le necessarie modalità di serializzazione/deserializzazione dei dati, oppure in un ambiente semi-automatico a partire da codice oggetto, attraverso l'utilizzo di opportuni tools per la generazione del codice di wrapping del kernel.

Il template Map è implementato tramite un libreria (CCSkel-SDK) sviluppata con approccio object-oriented per l'istanziazione di un grafo di processamento Master-Slave basato su un paradigma di parallelizzazione data-parallel, in cui l'astrazione del dato da elaborare è l'unione dei parametri di chiamata del kernel legacy. In Appendice è riportato il progetto della libreria CCSkel con il relativo modello degli oggetti.

Il template è composto dai seguenti file, dove "*xxx*" sostituisce un nome identificativo per il modulo da parallelizzare:

- **map_*xxx*.h/cpp**
- **map_*xxx*_defs.h**
- **map_*xxx*_moc.hpp/cpp**
- **map_*xxx*_module.h/cpp**

La coppia di file **map_*xxx*.h/cpp** contiene la definizione di una classe (*Map_XXX*) che implementa tutte le funzionalità specifiche dei vari processi dello skeleton, tra cui:

- la procedura principale del processo Emitter (Master)
- la strategia di distribuzione e raccolta dei dati
- la procedura di processamento dei Worker (Slave)

Il file **map_*xxx*_defs.h** contiene la definizione dei dati scambiati tra processo Master e processo Slave.

Il file **map_*xxx*_moc.hpp** contiene la specifica delle operazioni di serializzazione/deserializzazione dei dati.

Il file **map_*xxx*_moc.cpp** contiene la specifica delle operazioni di creazione dello skeleton (Costruttori) per la corretta istanziazione delle classi che implementano i processi di Emitter e Worker.

La coppia di file **map_*xxx*_module.h/ccp** contiene la definizione del modulo parallelo, il quale crea l'istanza specializzata della classe che contiene le funzionalità dei vari elementi dello skeleton (*Map_XXX*), crea un'istanza di controllo appropriata per tale skeleton (*MapSkelControl*), configura la skeleton secondo il documento XML-ICC e ne avvia l'esecuzione.

4.1.1 Esempio di creazione di un modulo parallelo tramite parallelizzazione del kernel computazionale

In questa sezione è riportato un esempio di utilizzo del template Map per la parallelizzazione di un modulo seriale (CZSMotion) a partire da codice sorgente.

Il modulo di partenza ha la seguente interfaccia funzionale:

```
/**
 *  The CZS-Motion module namespace.
 */
namespace CZSMotion_Module
{
    using namespace std;

    /**
     *  The CZS-Motion Map estimation module call-method.
     *
     *  For each <frame>, from <begin> to <end>, inputs are:
     *
     *      <in_prefix>.<frame-1>.<in_suffix>, for prev frame
     *
     *      <in_prefix>.<frame>.<in_suffix>, for curr frame
     *
     *      <in_prefix>.<frame+1>.<in_suffix>, for next frame
     *
     *  Session output meta-data are:
     *
     *      <wdir>/DFL.<frame>..., for disparity from last
     *
     *      <wdir>/RFL.<frame>..., for reliability from last
     *
     *      <wdir>/DFN.<frame>..., for disparity from next
     *
     *      <wdir>/RFN.<frame>..., for reliability from next
     *
     *
     *  @param in_prefix    input file(s) prefix
     *  @param begin        start input frame number
     *  @param end          last input frame number
     *  @param in_suffix    input file(s) suffix
     *  @param wdir         session working dir
     *  @param disp_max     maximum displacement from (0,0)
     *  @param block_size   square block size.
     *
     *  @return 0 on success, -1 on error, -2 on params error
     */
    int call(const string &in_prefix, int begin, int end, const string &in_suffix,
             const string &wdir, int disp_max, int block_size);
}
```

Il modulo sfrutta internamente un kernel computazionale con la seguente interfaccia:

```
/**
 *  Perform Enhanced Circular Zonal Search based fast motion estimation.
 *
 *  @param prev previous frame
 *  @param curr current frame
 *  @param disp_max maximum displacement from (0,0)
 *  @param block_size square block size.
 *  @param rel reliability output map
 *  @param v_x horiz-component motion output map
 *  @param v_y vert-component motion output map
 *  @param nrow frame geometry, no. rows
 *  @param ncol frame geometry, no. columns
 *  @param nband frame geometry, no. bands
 *
 *  @return 0 on success, -1 on error, -2 on params error
 */
extern "C" int CZSMotion_Kernel(unsigned char *prev, unsigned char *curr,
                  int disp_max, int block_size, float *rel, short *v_x, short *v_y,
                  int nrow, int ncol, int nband);
```

La metodologia di parallelizzazione prevede l'esecuzione del modulo seriale sul processo Emitter con wrapping della chiamata al kernel per l'esecuzione concorrente del kernel legacy sui Worker, previo partizionamento dei dati.

I parametri di input del kernel rappresentano il dato in ingresso ai Worker, analogamente, i parametri di output rappresentano il dato in uscita dai Worker. Alcuni parametri sono contemporaneamente di input/output.

```
namespace CZSMotion_Module
{
    /**
     *  The internal kernel input of CZS-Motion module.
     *
     */
    struct czsmotion_kernel_input
    {
        unsigned char *prev;
        unsigned char *curr;
        int disp_max;
        int block_size;
        int nrow;
        int ncol;
        int nband;
    };

    /**
     *  The internal kernel output of CZS-Motion module.
     *
     */
    struct czsmotion_kernel_output
    {
        float *rel;
        short *v_x;
        short *v_y;
        int disp_max;
        int block_size;
        int nrow;
        int ncol;
        int nband;
        int retval;
    };
}
```

Le funzionalità del processo Emitter sono specificate attraverso la specializzazione della classe base IMapEmitter<Input,Output>, parametrizzata nei dati di ingresso e di uscita. I metodi da implementare sono:

- il metodo *parmain(...)*, attività principale dell'Emitter
- il metodo *distrib_strategy(...)*, per la definizione della strategia di distribuzione dei dati
- il metodo *collect_strategy(...)*, per la definizione della strategia di raccolta dei dati

All'atto della costruzione, l'Emitter riceve i parametri di chiamata del modulo (args).

```
/**
 *  The Emitter
 */
class Emitter : virtual public
IMapEmitter<czsmotion_kernel_input,czsmotion_kernel_output>
{
protected:
    /// Create
    Emitter(const class AL_DataSet &args);

public:
    /// Get instance
    static Emitter *instance();

private:
    /**
     *  The Emitter parallel main.
     */
    int parmain(const string &in_prefix, int begin, int end, const string &in_suffix,
                const string &wdir, int disp_max, int block_size);
    /**
     *  The Distribution strategy.
     *
     *  -------------------------------------------------
     *  @param in   the input item
     *  @param data the worker data-item to be distributed
     *  @param w    the worker
     */
    int distrib_strategy(czsmotion_kernel_input &in, czsmotion_kernel_input &data,
                         int w);
    /**
     *  The Collection strategy
     *
     *  -----------------------------
     *  @param out  the output item
     *  @param data the worker data-item collected
     *  @param w    the worker
     */
    int collect_strategy(czsmotion_kernel_output &out, czsmotion_kernel_output &data,
                         int w);
private:
    /// Arguments
    const class AL_DataSet &args;
};
```

Le funzionalità del processo Worker sono specificate attraverso la specializzazione della classe base IMapWorker<Input,Output>, parametrizzata nei dati di ingresso e di uscita. L'unico metodo da implementare è il metodo *process(...)*, attività principale del Worker.

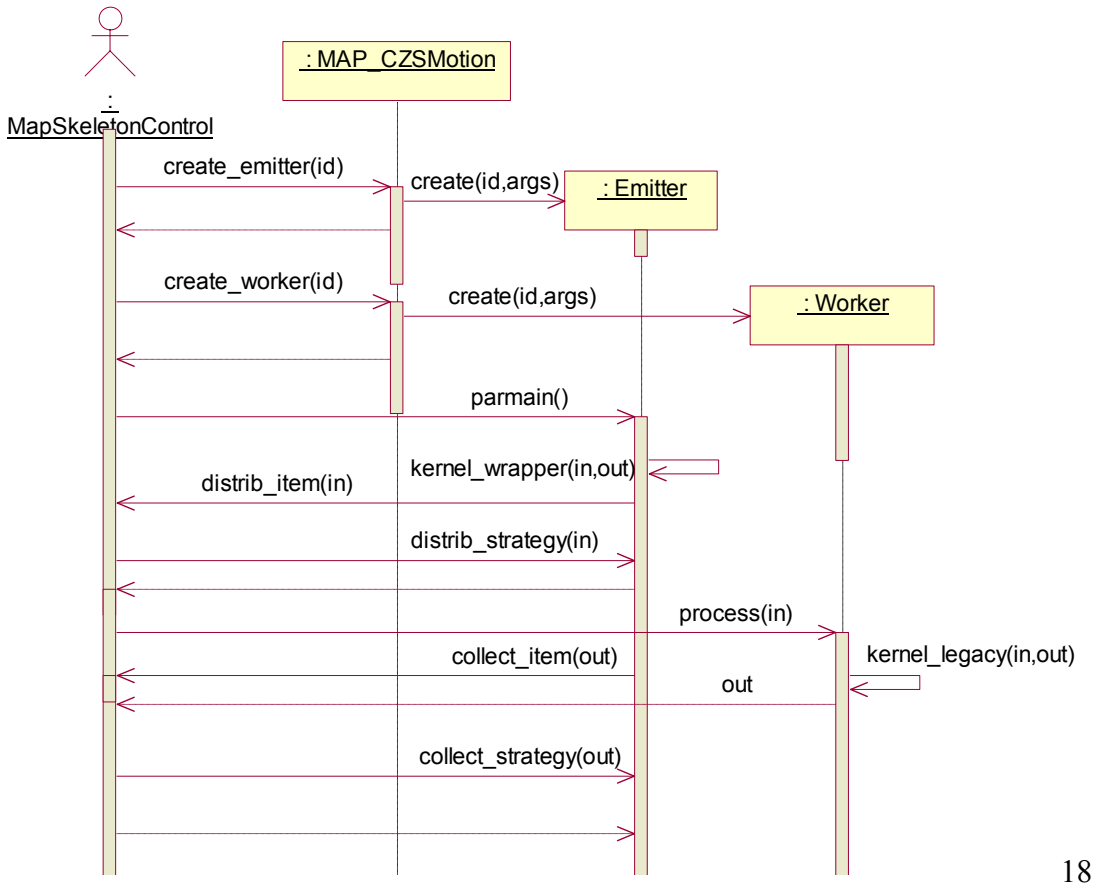
```
/**
 *  The Worker.
 */
class Worker : virtual public
IMapWorker<czsmotion_kernel_input,czsmotion_kernel_output>
{
protected:
    /// Create Worker.
    Worker(const class AL_DataSet &args);

    /**
     *  The Process phase.
     *
     *  --------------------------------------
     *  @param in  the worker  input data-item
     *  @param out the worker output data-Item
     *
     *  @retval 0=all_done, -1=on_error
     */
    int process(czsmotion_kernel_input &in, czsmotion_kernel_output &out);

private:
    /// Arguments
    const class AL_DataSet &args;
};
```

La definizione dello Skeleton Parallelo (Map_CZSMotion) avviene tramite specializzazione della classe base MapSkeleton, con implementazione dei metodi di istanziazione delle specifiche classi di Emitter e Worker.

Per evitare interferenze con altri moduli paralleli, la definizione dello Skeleton Parallelo contiene anche la definizione delle precedenti classi Emitter/Worker.

```
/**
 *  The CZS-Motion Map Skeleton.
 */
class Map_CZSMotion : public MapSkeleton
{
public:
    /// Create
    Map_CZSMotion(const class AL_DataSet &args);

    /// The Emitter
    class Emitter
    [...]

    /// The Worker
    class Worker
    [...]

private:
    /// Create implemented Emitter instance
    class IEmitter *create_emitter(int id) const;

    /// Create implemented Worker instance
    class IWorker *create_worker(int id) const;

private:
    /// Arguments
    const class AL_DataSet &args;
};
```

Di seguito è riportato il diagramma di sequenza che riassume l'attivita dello skeleton parallelo.



1. La classe di controllo dello skeleton map (*MapSkelControl*) invoca il metodo per la creazione di istanze di Emitter/Worker, attraverso la classe di skeleton parallelo (*Map_CZSMotion*) la quale fornisce le corrette implementazioni.
2. Successivamente, la classe di controllo invoca il metodo principale dell'Emitter che contiene l'attivazione del modulo seriale e la conseguente chiamata al kernel.
3. Nel processo Emitter, il kernel effettivamente chiamato è il kernel di wrapping che effettua due semplici operazioni:
    - distribuzione del dato di input, contenente tutti i parametri di input da passare al kernel legacy
    - raccolta del dato di output, contenente tutti i parametri di output restituiti dal kernel legacy
4. La distribuzione del dato di input avviene previo partizionamento dello stesso, tramite l'invocazione del metodo *distrib_strategy* sull'Emitter.
5. Ogni Worker processa una porzione del dato di input invocando il kernel legacy, e restituisce la propria porzione del dato di output.
6. Il riassemblamento del dato finale di output avviene tramite l'invocazione del metodo *collect_strategy* sull'Emitter.
7. Il kernel di wrapping termina restituendo i parametri di output contenuti nel dato finale. L'Emitter prosegue l'esecuzione.

L'implementazione del metodo *parmain* sull'Emitter avviene tramite una copia esatta del codice sorgente che implementa il modulo seriale, con l'unica eccezione della chiamata al kernel legacy la quale va sostituita con la chiamata al kernel di wrapping:

```
/**
 *  The Emitter parallel main.
 */
int Map_CZSMotion::Emitter::parmain(const string &in_prefix, int begin, int end,
          const string &in_suffix, const string &wdir, int disp_max, int block_size)
{
    [...]

        // frame cycle
    for (...)
        {
            [...]

                // CZS MMap kernel call
//          int retval = CZSMotion_Kernel(...);

            int retval = CZSMotion_Kernel_Wrapper(...);

            [...]
        }

    [...]

        // all done
    return (0);
}
```

Il kernel di wrapping avvia la distribuzione del dato di input incapsulando i parametri attuali di input e attende il dato di output per restituire i parametri di output:

```
/**
 *  Internal kernel wrapper.
 */
int CZSMotion_Kernel_Wrapper(unsigned char *prev, unsigned char *curr, int disp_max,
                                int block_size, float *rel, short *v_x, short *v_y,
                                int nrow, int ncol, int nband)
{
        // Item to be distributed
    czsmotion_kernel_input call_item =
       { prev,curr,disp_max,block_size,nrow,ncol,nband };

        // DISTRIBUTE
    Map_CZSMotion::Emitter::instance()->distrib_item(call_item);

        // Item returned
    czsmotion_kernel_output ret_item =
       { rel,v_x,v_y,disp_max,block_size,nrow,ncol,nband,0 };

        // COLLECT
    Map_CZSMotion::Emitter::instance()->collect_item(ret_item);

    return ret_item.retval;
}
```

La strategia di distribuzione prevede il broadcasting di alcuni parametri e lo scattering delle immagini. Il metodo da implementare riporta come parametri:

- un riferimento al dato da suddividere (*in*)
- un riferimento al dato da distribuire (*data*)
- il numero ordinale del worker a cui andrà la corrente porzione di dato

Il broadcasting dei parametri si implementa per semplice copia da *in* in *data*.

Lo scattering delle immagini si implementa tramite un opportuno metodo (*scatter*) ereditato dalla super classe.

Nell'esempio, lo scattering avviene per le due immagini *in.prev* e *in.curr*, ciascuna suddivisa per righe (*in.nrow*) con sovrapposizione di *disp_max*.

```
/**
 *  The Distribution strategy.
 *
 *  ------------------------------------------------
 *  @param in   the input item
 *  @param data the worker data-item to be distributed
 *  @param w    the worker
 */
int Map_CZSMotion::Emitter::distrib_strategy(czsmotion_kernel_input &in,
                                             czsmotion_kernel_input &data, int w)
{
    data.disp_max   = in.disp_max;
    data.block_size = in.block_size;
    data.ncol       = in.ncol;
    data.nband      = in.nband;

    scatter(data.prev, data.nrow, in.prev, in.nrow, in.ncol*in.nband, w, in.disp_max);
    scatter(data.curr, data.nrow, in.curr, in.nrow, in.ncol*in.nband, w, in.disp_max);

    return (0);
}
```

La fase di processamento riporta la chiamata al kernel legacy previa estrazione dei parametri di input dal dato di ingresso e con successivo incapsulamento dei parametri di output nel dato di uscita.

```
/**
 *  The Process phase.
 *
 *  ----------------------------------------
 *  @param in  the worker  input data-item
 *  @param out the worker output data-Item
 *
 *  @retval 0=all_done, -1=on_error
 */
int Map_CZSMotion::Worker::process(czsmotion_kernel_input &in,
                                   czsmotion_kernel_output &out)
{
    out.nrow       = in.nrow;
    out.ncol       = in.ncol;
    out.nband      = in.nband;
    out.disp_max   = in.disp_max;
    out.block_size = in.block_size;
    out.rel        = new float[out.nrow*out.ncol];
    out.v_x        = new short[out.nrow*out.ncol];
    out.v_y        = new short[out.nrow*out.ncol];

    OUT("\t>> Worker(%d): processing %4dx%d\n", id, in.ncol, in.nrow);

        // kernel-call
    out.retval = CZSMotion_Kernel(in.prev,in.curr, in.disp_max,in.block_size,
                                  out.rel out.v_x,out.v_y, in.nrow,in.ncol in.nband);

    return (0);
}
```

La strategia di raccolta prevede il gathering delle mappe di moto. Il metodo da implementare riporta come parametri:

- un riferimento al dato da riassemblare (*out*)
- un riferimento al dato raccolto (*data*)
- il numero ordinale del worker da cui proviene la corrente porzione di dato

Il gathering delle mappe di moto si implementa tramite un opportuno metodo (*gather*) ereditato dalla super classe.

Nell'esempio, il gathering avviene per le tre mappe *out.rel, out.v_x, out.v_y*, ciascuna riassemblata per righe con sovrapposizione di *disp_max*.

```
/**
 *  The Collection strategy
 *
 *  ----------------------------
 *  @param out  the output item
 *  @param data the worker data-item collected
 *  @param w    the worker
 */
int Map_CZSMotion::Emitter::collect_strategy(czsmotion_kernel_output &out,
                                             czsmotion_kernel_output &data, int w)
{
    if (out.retval != -1)
        out.retval = data.retval;

    gather(out.rel,data.rel,  out.nrow, out.ncol*sizeof(float), w, out.disp_max);
    gather(out.v_x,data.v_x,  out.nrow, out.ncol*sizeof(short), w, out.disp_max);
    gather(out.v_y,data.v_y,  out.nrow, out.ncol*sizeof(short), w, out.disp_max);

    return (0);
}
```

I metodi di creazione delle istanze di Emitter/Worker si implementano tramite l'ausilio delle classi utility (*Concrete_Emitter*, *Concrete_Worker*) parametrizzate rispettivamente nei tipi <Emitter,Input,Output> e <Worker,Input,Output>.

```
IEmitter *Map_CZSMotion::create_emitter(int id) const
{
        // new concrete emitter instance
    return new
    Concrete_Emitter<Emitter,czsmotion_kernel_input,czsmotion_kernel_output>(id,args);
}

IWorker *Map_CZSMotion::create_worker(int id) const
{
        // new concrete worker instance
    return new
    Concrete_Worker<Worker,czsmotion_kernel_input,czsmotion_kernel_output>(id,args);
}
```

Qualora necessario, è possibile specializzare gli operatori di serializzazione e deserializzazione delle strutture dati di input e/o output. Nell'esempio, il dato di input contiene due array allocati dinamicamente (*data.prev*, *data.curr*)

```
Push &operator <<(Push &pusher, czsmotion_kernel_input &data)
{
       // put data image description
    pusher << data.nrow << data.ncol << data.nband;

    return pusher   // put all
        << (Push::block) { data.nrow*data.ncol*data.nband, data.prev }
        << (Push::block) { data.nrow*data.ncol*data.nband, data.curr }
        << data.disp_max
        << data.block_size
        ;
}

Pop &operator >>(Pop &popper, czsmotion_kernel_input &data)
{
       // get data image description
    popper >> data.nrow >> data.ncol >> data.nband;

    data.prev = new unsigned char[data.nrow*data.ncol*data.nband];
    data.curr = new unsigned char[data.nrow*data.ncol*data.nband];

    return popper   // get all
        >> (Pop::block) { data.nrow*data.ncol*data.nband, data.prev }
        >> (Pop::block) { data.nrow*data.ncol*data.nband, data.curr }
        >> data.disp_max
        >> data.block_size
        ;
}

void Dealloc<czsmotion_kernel_input>::operator()(czsmotion_kernel_input &data) const
{
    delete (data.prev);
    delete (data.curr);
}
```

Analogamente, il dato di output contiene tre array allocati dinamicamente (*data.rel*, *data.v_x, data.v_y*)

```
Push &operator <<(Push &pusher, czsmotion_kernel_output &data)
{
       // put data image description
    pusher << data.nrow << data.ncol << data.nband;

    return pusher   // put all
        << (Push::block) { data.nrow*data.ncol*sizeof(float), data.rel }
        << (Push::block) { data.nrow*data.ncol*sizeof(short), data.v_x }
        << (Push::block) { data.nrow*data.ncol*sizeof(short), data.v_y }
        << data.disp_max
        << data.block_size
        << data.retval
        ;
}
```

```
Pop &operator >>(Pop &popper, czsmotion_kernel_output &data)
{
       // get data image description
    popper >> data.nrow >> data.ncol >> data.nband;

    data.rel = new float[data.nrow*data.ncol];
    data.v_x = new short[data.nrow*data.ncol];
    data.v_y = new short[data.nrow*data.ncol];

    return popper   // get all
        >> (Pop::block) { data.nrow*data.ncol*sizeof(float), data.rel }
        >> (Pop::block) { data.nrow*data.ncol*sizeof(short), data.v_x }
        >> (Pop::block) { data.nrow*data.ncol*sizeof(short), data.v_y }
        >> data.disp_max
        >> data.block_size
        >> data.retval
        ;
}

void Dealloc<czsmotion_kernel_output>::operator()(czsmotion_kernel_output &data) const
{
    delete (data.rel);
    delete (data.v_x);
    delete (data.v_y);
}
```

Infine, il modulo parallelo si presenta con la stessa interfaccia di chiamata del modulo legacy, con l'aggiunta dei parametri di configurazione per lo skeleton parallelo.

```
/**
 *  The CZSMotion map module namespace
 */
namespace Map_CZSMotion_Module
{
    using namespace std;

    /**
     *  The Map CZS-Motion parallel module call method.
     *
     *  Relevant-data are:
     *
     *      "MAP_CZSMOTION_MODULE_ICC", for XML-ICC configuration document
     *      "MAP_CZSMOTION_MODULE_LOG", for additional logs output
     *      "MAP_CZSMOTION_MODULE_MSG", for additional messages output
     *      "MAP_CZSMOTION_MODULE_DBG", for additional debug output
     *
     *  @see CZSMotion_Module::call
     *
     *  @param argc configuration args count
     *  @param argv configuration args values
     *
     *  @return 0 on success, -1 on error
     */
    int call(int argc, char *argv[], const string &in_prefix, int begin, int end,
          const string &in_suffix, const string &wdir, int disp_max, int block_size);
}
```

L'implementazione del metodo di chiamata del modulo parallelo:

- recupera il grafo di configurazione (*xmlicc*) dai relevant-data
- istanzia lo skeleton parallelo (*map_czsmotion*)
- istanzia una classe di controllo per uno skeleton di tipo map (*map_control*)
- configura lo skeleton tramite il grafo di configurazione (*xmlicc*)
- avvia l'esecuzione dello skeleton

```
int Map_CZSMotion_Module::call(int argc, char *argv[], const string &in_prefix,
                    int begin, int end, const string &in_suffix, const string &wdir,
                    int disp_max, int block_size)
{
```

```
        // The XML-ICC component configuration
    const char *xmlicc = getenv("MAP_CZSMOTION_MODULE_ICC");

    if (!xmlicc)
    {   ERR("! Map_CZSMotion_Module: can't find xml-icc configuration document.\n");
        return (-1);
    }

        // The Skeleton Execution arguments
    AL_DataSet args;

        // Serialize <string> "in_prefix"
    args.add( new AL_Data<string>("in_prefix", in_prefix) );

        // Serialize <int> "begin"
    args.add( new AL_Data<int>("begin", begin) );

        // Serialize <int> "end"
    args.add( new AL_Data<int>("end", end) );

        // Serialize <string> "in_suffix"
    args.add( new AL_Data<string>("in_suffix", in_suffix) );

        // Serialize <string> "wdir"
    args.add( new AL_Data<string>("wdir", wdir) );

        // Serialize <int> "disp_max"
    args.add( new AL_Data<int>("disp_max", disp_max) );

        // Serialize <int> "block_size"
    args.add( new AL_Data<int>("block_size", block_size) );


        // Create Map CZS-Motion Skeleton instance
    Map_CZSMotion map_czsmotion(args);


        // Create Map Skeleton control Instance
    MapSkelControl map_control(map_czsmotion);

        // Configure Skeleton
    if (map_control.conf(xmlicc, argc,argv) != 0)
        return (-1);

    return  // Start Skeleton Execution
        map_control.start();
}
```

4.2 Template Farm

Il template Farm è utilizzato per la creazione di un modulo parallelo a partire da un modulo legacy, secondo una strategia di parallelizzazione dell'intero modulo. Tale template può essere utilizzato come guida alla parallelizzazione a partire da codice sorgente, inserendo il codice legacy nelle appropriate sezioni e specificando inoltre le necessarie modalità di serializzazione/deserializzazione dei dati, oppure in un ambiente semi-automatico a partire da codice oggetto.

Il template Farm è implementato tramite un libreria (CCSkel-SDK) sviluppata con approccio object-oriented per l'istanziazione di un grafo di processamento Master-Slave basato su un paradigma di parallelizzazione stream-parallel, in cui l'astrazione del dato da elaborare è l'unione dei parametri di chiamata del modulo legacy. In Appendice è riportato il progetto della libreria CCSkel con il relativo modello degli oggetti.

Il template è composto dai seguenti file, dove "*xxx*" sostituisce un nome identificativo per il modulo da parallelizzare:

- **farm_*xxx*.h/cpp**
- **farm_*xxx*_defs.h**
- **farm_*xxx*_moc.hpp/cpp**
- **farm_*xxx*_module.h/cpp**

La coppia di file **farm_*xxx*.h/cpp** contiene la definizione di una classe (*Farm_XXX*) che implementa tutte le funzionalità specifiche dei vari processi dello skeleton, tra cui:

- la procedura principale del processo Emitter (Master)
- la procedura di processamento dei Worker (Slave)

Il file **farm_*xxx*_defs.h** contiene la definizione dei dati scambiati tra processo Master e processo Slave.

Il file **farm_*xxx*_moc.hpp** contiene la specifica delle operazioni di serializzazione/deserializzazione dei dati.

Il file **farm_*xxx*_moc.cpp** contiene la specifica delle operazioni di creazione dello skeleton (Construttori) per la corretta istanziazione delle classi che implementano i processi di Emitter e Worker.

La coppia di file **farm_*xxx*_module.h/ccp** contiene la definizione del modulo parallelo, il quale crea l'istanza specializzata della classe che contiene le funzionalità dei vari elementi dello skeleton (*Farm_XXX*), crea un'istanza di controllo appropriata per tale skeleton (*FarmSkelControl*), configura la skeleton secondo il documento XML-ICC e ne avvia l'esecuzione.

4.2.1 Esempio di creazione di un modulo parallelo tramite parallelizzazione dell'intero modulo seriale

In questa sezione è riportato un esempio di utilizzo del template Farm per la parallelizzazione di un modulo seriale (CZSMotion) a partire da codice sorgente.
Il modulo di partenza ha la seguente interfaccia funzionale:

```
/**
 *  The CZS-Motion module namespace.
 */
namespace CZSMotion_Module
{
    using namespace std;

    /**
     *  The CZS-Motion Map estimation module call-method.
     *
     *  For each <frame>, from <begin> to <end>, inputs are:
     *
     *      <in_prefix>.<frame-1>.<in_suffix>, for prev frame
     *
     *      <in_prefix>.<frame>.<in_suffix>, for curr frame
     *
     *      <in_prefix>.<frame+1>.<in_suffix>, for next frame
     *
     *  Session output meta-data are:
     *
     *      <wdir>/DFL.<frame>..., for disparity from last
     *
     *      <wdir>/RFL.<frame>..., for reliability from last
     *
     *      <wdir>/DFN.<frame>..., for disparity from next
     *
     *      <wdir>/RFN.<frame>..., for reliability from next
     *
     *
     *  @param in_prefix    input file(s) prefix
     *  @param begin        start input frame number
     *  @param end          last input frame number
     *  @param in_suffix    input file(s) suffix
     *  @param wdir         session working dir
     *  @param disp_max     maximum displacement from (0,0)
     *  @param block_size   square block size.
     *
     *  @return 0 on success, -1 on error, -2 on params error
     */
    int call(const string &in_prefix, int begin, int end, const string &in_suffix,
             const string &wdir, int disp_max, int block_size);
}
```

La metodologia di parallelizzazione prevede la scomposizione della chiamata originaria al modulo seriale in una sequenza di chiamate elementari, attivabili concorrentemente sui Worker, e successiva integrazione dei risultati parziali.
Nell'esempio, il modulo legacy itera internamente su di una sequenza definita dall'intervallo [*begin*,*end*]. La parallelizzazione riguarderà la scomposizione della chiamata relativa a tale intervallo in *end-begin*+1 chiamate elementari.

Il metodo principale dell'Emitter contiene sempre un'iterazione. La variabile di controllo di tale iterazione, insieme ad alcuni dei parametri di input che intervengono sulla condizione di partenza/arresto, rappresentano l'indice di riferimento dell'iterazione. I parametri di input del modulo che variano durante le varie iterazioni, rappresentano il dato in ingresso ai Worker. I parametri di output rappresentano il dato in uscita dai Worker.
I parametri di input che non variano durante le varie iterazioni, sono conosciuti ai Worker (args) e non rappresentano alcun dato da trasferire.

```
namespace CZSMotion_Module
{
    /**
     *  The CZS-Motion module control loop index
     *
     */
    struct czsmotion_module_index
    {
        int begin;
        int end;
        int frame;
    };

    /**
     *  The CZS-Motion module input
     *
     */
    struct czsmotion_module_input
    {
        int begin;
        int end;
    };

    /**
     *  The CZS-Motion module output
     *
     */
    struct czsmotion_module_output
    {
        int retval;
    };
}
```

Le funzionalità del processo Emitter sono specificate attraverso la specializzazione della classe base IFarmEmitter<Index,Input,Output>, parametrizzata nell'indice d'iterazione e nei dati di ingresso e di uscita. I metodi da implementare sono:

- il metodo *parmain(...)*, attività principale dell'Emitter
- il metodo *distrib_strategy(...)*, per la definizione della strategia di generazione delle chiamate elementari

- il metodo *collect_strategy(...)*, per la definizione della strategia di integrazione dei risultati parziali

```
/**
 *  The Emitter
 */
class Emitter : virtual public
IFarmEmitter<czsmotion_module_index,czsmotion_module_input,czsmotion_module_output>
{
protected:
    /// Create
    Emitter(const class AL_DataSet &args);

private:
    /**
     *  The Emitter parallel main.
     */
    int parmain(int begin, int end);

    /**
     *  The Distribution strategy.
     *
     *  ------------------------------------------------
     *  @param idx  the index item
     *  @param data the worker data-item to be distributed
     */
    int distrib_strategy(czsmotion_module_index &idx, czsmotion_module_input &data);

    /**
     *  The Collection strategy
     *
     *  -----------------------------
     *  @param idx  the index item
     *  @param in   the input item
     *  @param data the worker data-item collected
     */
    int collect_strategy(czsmotion_module_index &idx, czsmotion_module_input &in,
                         czsmotion_module_output &data);

private:
    /// Arguments
    const class AL_DataSet &args;
};
```

Le funzionalità del processo Worker sono specificate attraverso la specializzazione della classe base IFarmWorker<Input,Output>, parametrizzata nei dati di ingresso e di uscita. L'unico metodo da implementare è il metodo *process(...)*, attività principale del Worker.

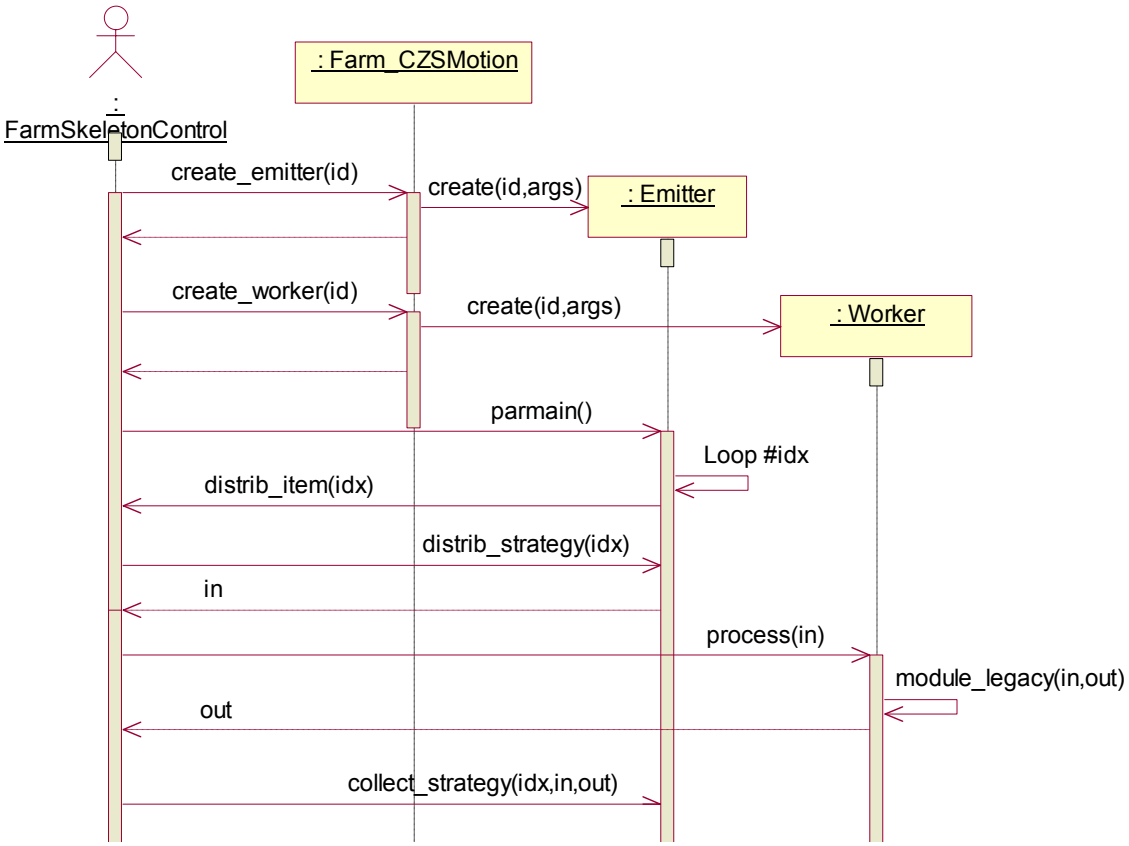
```
/**
 *  The Worker.
 */
class Worker : virtual public
IFarmWorker<czsmotion_module_input,czsmotion_module_output>
{
protected:
    /// Create Worker.
    Worker(const class AL_DataSet &args);

    /**
     *  The Process phase.
     *
     *  ---------------------------------------
     *  @param in  the worker  input data-item
     *  @param out the worker output data-Item
     *
     *  @retval 0=all_done, -1=on_error
     */
    int process(const string &in_prefix, const string &in_suffix, const string &wdir,
          int disp_max, int block_size, czsmotion_module_input &in,
                  czsmotion_module_output &out);
private:
    /// Arguments
    const class AL_DataSet &args;
};
```

La definizione dello Skeleton Parallelo (Farm_CZSMotion) avviene tramite specializzazione della classe base FarmSkeleton, con implementazione dei metodi di istanziazione delle specifiche classi di Emitter e Worker.

Per evitare interferenze con altri moduli paralleli, la definizione dello Skeleton Parallelo contiene anche la definizione delle precedenti classi Emitter/Worker.

```
/**
 *  The CZS-Motion Farm Skeleton.
 */
class Farm_CZSMotion : public FarmSkeleton
{
public:
    /// Create
    Farm_CZSMotion(const class AL_DataSet &args);

    /// The Emitter
    class Emitter
    [...]

    /// The Worker
    class Worker
    [...]

private:
    /// Create implemented Emitter instance
    class IEmitter *create_emitter(int id) const;

    /// Create implemented Worker instance
    class IWorker *create_worker(int id) const;

private:
    /// Arguments
    const class AL_DataSet &args;
};
```

Di seguito è riportato il diagramma di sequenza che riassume l'attivita dello skeleton:



1. La classe di controllo dello skeleton farm (*FarmSkelControl*) invoca il metodo per la creazione di istanze di Emitter/Worker, attraverso la classe di skeleton parallelo (*Farm_CZSMotion*) la quale fornisce le corrette implementazioni.
2. Successivamente, la classe di controllo invoca il metodo principale dell'Emitter che contiene il ciclo iterativo (Loop) sulla variabile *idx.* Ad ogni iterazione si richiede la distribuzione del dato relativo all'indice corrente.
3. Quando un Worker è disponibile per l'elaborazione, la distribuzione del dato (parziale) di input avviene in seguito alla generazione dello stesso, tramite l'invocazione del metodo *distrib_strategy* sull'Emitter.
4. Ogni Worker processa un dato (parziale) di input invocando il modulo legacy, e restituisce un dato (parziale) di output.
5. L'integrazione del dato parziale nel dato finale di output avviene tramite l'invocazione del metodo *collect_strategy* sull'Emitter.

L'implementazione del metodo *parmain* sull'Emitter avviene tramite la definizione del ciclo iterativo (Loop) su un insieme ridotto dei parametri d'ingresso. Ad ogni iterazione si richiede la distribuzione del dato relativo all'indice corrente.

Nell'esempio, tralasciando i parametri di input che non variano durante le varie iterazioni, si vuole scomporre la chiamata originale al modulo legacy, definita dalla coppia di parametri (*begin, end*), in *end*-*begin*+1 chiamate elementari, ciascuna definita dalla coppia di parametri (*frame*-1, *frame*+1), con *frame=begin…end*, essendo *frame* la variabile discriminante dell'iterazione.

```
/**
 *  The Emitter parallel main.
 */
int Farm_CZSMotion::Emitter::parmain(int begin, int end)
{
        // Emitter Loop
    for (int frame=begin; frame<=end; frame++)
        {
            // Loop index
        czsmotion_module_index index_item = { begin, end, frame };

            // DISTRIBUTE
        distrib_item(index_item);
        }

    return (0);
}
```

La strategia di distribuzione deve generare la chiamata elementare relativa all'iterazione definita dall'indice *idx,* sotto forma di dato di input (*data*).

Nell'esempio, per ogni iterazione definita sulla variabile *idx.frame*, la relativa chiamata sarà la coppia (*idx.frame*-1, *idx.frame*+1) con le dovute condizioni di confine.

```
/**
 *  The Distribution strategy.
 *
*  ------------------------------------------------
 *  @param idx  the index item
 *  @param data the worker data-item to be distributed
 */
int Farm_CZSMotion::Emitter::distrib_strategy(czsmotion_module_index &idx,
                                              czsmotion_module_input &data)
{
    data.begin = idx.frame==idx.begin? idx.begin: idx.frame-1;

    data.end = idx.frame==idx.end? idx.end: idx.frame+1;

    return (0);
}
```

La fase di processamento riporta la chiamata al modulo legacy previa estrazione dei parametri di input dal dato di ingresso e con successivo incapsulamento dei parametri di output nel dato di uscita.

```
/**
 *  The Process phase.
 *
*  ----------------------------------------
 *  @param in  the worker  input data-item
 *  @param out the worker output data-Item
 *
 *  @retval 0=all_done, -1=on_error
 */
int Farm_CZSMotion::Worker::process(const string &in_prefix, const string &in_suffix,
               const string &wdir, int disp_max, int block_size,
                       czsmotion_module_input &in, czsmotion_module_output &out)
{
                // CZS-Motion module call
    out.retval = CZSMotion_Module::call(in_prefix,in.begin,in.end,in_suffix,
                                                              wdir,disp_max,block_size);

    return (0);
}
```

La strategia di raccolta deve integrare il dato parziale restituito dal Worker (*data*), relativo all'iterazione *idx*, quando il dato di input era *in*.

Nell'esempio, nessuna particolare operazione è richiesta se non la stampa testuale di un messaggio di errore in caso di esecuzione non a buon fine.

```
/**
 *  The Collection strategy
 *
*  @param idx  the index item
 *  @param in   the input item
 *  @param data the worker data-item collected
 */
int Farm_CZSMotion::Emitter::collect_strategy(czsmotion_module_index &idx,
                           czsmotion_module_input &in, czsmotion_module_output &data)
{
    if (data.retval < 0)
        ERR("! Farm_CZSMotion: WARNING: in frame %d, return status = %d.\n",
                                                            idx.frame, data.retval);

    return (0);
}
```

Qualora necessario, è possibile specializzare gli operatori di serializzazione e deserializzazione delle strutture dati di indice, input e/o output. Nell'esempio, gli operatori di default sono sufficienti.

Infine, il modulo parallelo si presenta con la stessa interfaccia di chiamata del modulo legacy, con l'aggiunta dei parametri di configurazione per lo skeleton parallelo.

```
/**
 *  The CZSMotion farm module namespace
 */
namespace Farm_CZSMotion_Module
{
    using namespace std;

    /**
     *  The Farm CZS-Motion Map estimation parallel module call method.
     *
     *  Relevant-data are:
     *
     *      "FARM_CZSMOTION_MODULE_ICC", for XML-ICC configuration document
     *      "FARM_CZSMOTION_MODULE_LOG", for additional logs output
     *      "FARM_CZSMOTION_MODULE_MSG", for additional messages output
     *      "FARM_CZSMOTION_MODULE_DBG", for additional debug output
     *
     *  @see CZSMotion_Module::call
     *
     *  @param argc configuration args count
     *  @param argv configuration args values
     *
     *  @return 0 on success, -1 on error
     */
    int call(int argc, char *argv[], const string &in_prefix, int begin, int end,
          const string &in_suffix, const string &wdir, int disp_max, int block_size);
}
```

L'implementazione del metodo di chiamata del modulo parallelo:

- recupera il grafo di configurazione (*xmlicc*) dai relevant-data
- istanzia lo skeleton parallelo (*farm_czsmotion*)
- istanzia una classe di controllo per uno skeleton di tipo farm (*farm_control*)
- configura lo skeleton tramite il grafo di configurazione (*xmlicc*)
- avvia l'esecuzione dello skeleton

```
int Farm_CZSMotion_Module::call(int argc, char *argv[], const string &in_prefix,
                    int begin, int end, const string &in_suffix, const string &wdir,
                    int disp_max, int block_size)
{

        // The XML-ICC component configuration
    const char *xmlicc = getenv("MAP_CZSMOTION_MODULE_ICC");

    if (!xmlicc)
    {   ERR("! Map_CZSMotion_Module: can't find xml-icc configuration document.\n");
        return (-1);
    }

        // The Skeleton Execution arguments
    AL_DataSet args;

        // Serialize <string> "in_prefix"
    args.add( new AL_Data<string>("in_prefix", in_prefix) );

        // Serialize <int> "begin"
    args.add( new AL_Data<int>("begin", begin) );
```

```
        // Serialize <int> "end"
    args.add( new AL_Data<int>("end", end) );

        // Serialize <string> "in_suffix"
    args.add( new AL_Data<string>("in_suffix", in_suffix) );

        // Serialize <string> "wdir"
    args.add( new AL_Data<string>("wdir", wdir) );

        // Serialize <int> "disp_max"
    args.add( new AL_Data<int>("disp_max", disp_max) );

        // Serialize <int> "block_size"
    args.add( new AL_Data<int>("block_size", block_size) );


        // Create Farm CZS-Motion Skeleton instance
    Farm_CZSMotion farm_czsmotion(args);


        // Create Farm Skeleton control Instance
    FarmSkelControl farm_control(farm_czsmotion);

        // Configure Skeleton
    if (farm_control.conf(xmlicc, argc,argv) != 0)
        return (-1);

    return  // Start Skeleton Execution
        farm_control.start();
}
```

5. Test Case

In questa sezione sono riportati i test eseguiti sulla parallelizzazione di un modulo legacy proveniente da un'applicazione di restauro di filmati digitali [21].

Il modulo in questione, denominato Scratch-Detection, permette l'individuazione di graffi lineari, non regolari, su di una sequenza di immagini d'ingresso, attraverso l'utilizzo di un kernel computazionale (find_seeds) di Image Processing [20].

Il modulo parallelo è stato ottenuto attraverso una parallelizzazione del suddetto kernel, quindi è stato inserito in un componente parallelo con tecnologia MPI.

Il componente è stato testato in una griglia di 5 Workstation: Prometeo1, Prometeo2, RedVision, Ulisse, Vision300, secondo due profili di configurazione A e B. Per i dettagli architetturali delle Workstation si rimanda in Appendice.

I due profili di configurazione si differiscono per il grado di disomogeneità nella performance delle macchine partecipanti.

Ogni configurazione è stata testata utilizzando indici di performance omogenei (unbalanced) per verificare il grado di sbilanciamento iniziale, e 5 dei più comuni indici di performance utilizzati nelle metriche di BenchMarking.

La configurazione A, composta dai nodi Prometeo1, RedVision, Prometeo2, Ulisse, presenta uno sbilanciamento iniziale moderato.

La configurazione B, composta dai nodi Prometeo1, RedVision, Prometeo2, Vision300, presenta uno sbilanciamento iniziale elevato.

Gli indici di performance utilizzati sono:

- Bogomips, indice caratteristico della velocità assoluta della CPU, utilizzato nei SO Linux per la taratura dei tempi di attesa del Kernel.
- Mwips, Millions Whetstone Instructions Per Second, misura le prestazioni pure dell'unità in virgola mobile con un corto ciclo, adatto per mini-computers
- Dhryps, DHRYstone Per Second, misura le prestazioni dell'unità di calcolo con operazioni su interi
- Mflops, Millions of FLoating-point Operations Per Second, ottenuto tramite l'algortimo LinPack, misura le prestazioni in virgola mobile, adatto per Workstation

- Nbench, BYTEMark,v 2.2.2, misura le prestazioni di CPU, FPU e architettura di memoria di un sistema

Per il dettaglio di esecuzione dei vari algoritmi di misurazione delle performance si rimanda in appendice.

Il DataSet utilizzato nei test comprende tre filmati cinematografici digitalizzati ad alta risoluzione: Lohe1-1 (2880x2048), Lohe1-2 (1440x1024), Lohe4-1(720x512).

5.1 Risultati

Di seguito sono riassunti i risultati dei test effettuati. Per ogni tipo di indice di Performace utilizzato, sono evidenziati:

- Il peso attribuito a ciascun nodo (Emitter +Workers)
- La potenza totale impegnata, ossia il numero di nodi equivalenti calcolato in rapporto al nodo utilizzatto nell'esecuzione in seriale
- Lo SpeedUp misurato (sui tempi di calcolo)
- Lo SpeedUp teorico ottenibile in base alla potenza impegnata
- Un indice di bilanciamento, valutato come σ_t / $\bar{t}$, ossia come la media delle variazioni massime dei tempi di calcolo dei Workers attorno il valore centrale, in proporzione relativa. Valori alti indicano un elevato grado di sbilanciamento, valori bassi indicano un miglior bilanciamento.

Lohe1-1 (2880x2048), Configurazione A								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	1,43	1,53	0,35
bogomips	0,30	0,59	0,30	0,32	4,99	1,57	1,63	0,03
mwips	0,38	0,44	0,38	0,39	4,17	1,49	1,55	0,19
dhryps	0,36	0,52	0,36	0,37	4,43	1,52	1,58	0,18
mflops	0,37	0,56	0,37	0,38	4,51	1,42	1,59	0,36
nbench	0,35	0,53	0,35	0,35	4,50	1,53	1,59	0,13

Lohe1-2(1440x1024), Configurazione A								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	1,44	1,58	0,43
bogomips	0,30	0,59	0,30	0,32	4,99	1,59	1,71	0,11
mwips	0,38	0,44	0,38	0,39	4,17	1,49	1,61	0,29
dhryps	0,36	0,52	0,36	0,37	4,43	1,53	1,65	0,26
mflops	0,37	0,56	0,37	0,38	4,51	1,42	1,65	0,46
nbench	0,35	0,53	0,35	0,35	4,50	1,54	1,65	0,25

Lohe4-1(720x512), Configurazione A								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	1,47	1,57	0,37
bogomips	0,30	0,59	0,30	0,32	4,99	1,62	1,69	0,03
mwips	0,38	0,44	0,38	0,39	4,17	1,53	1,60	0,20
dhryps	0,36	0,52	0,36	0,37	4,43	1,57	1,63	0,22
mflops	0,37	0,56	0,37	0,38	4,51	1,46	1,64	0,38
nbench	0,35	0,53	0,35	0,35	4,50	1,58	1,64	0,16

Lohe1-1 (2880x2048), Configurazione B								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	0,88	1,53	0,08
bogomips	0,30	0,59	0,30	0,11	4,29	1,44	1,57	0,97
mwips	0,38	0,44	0,38	0,18	3,61	1,21	1,47	0,59
dhryps	0,36	0,52	0,36	0,12	3,76	1,39	1,49	0,83
mflops	0,37	0,56	0,37	0,07	3,69	1,37	1,48	0,71
nbench	0,35	0,53	0,35	0,11	3,82	1,40	1,50	0,88

Lohe1-2(1440x1024), Configurazione B								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	0,82	1,58	1,00
bogomips	0,30	0,59	0,30	0,11	4,29	1,44	1,63	0,08
mwips	0,38	0,44	0,38	0,18	3,61	1,17	1,52	0,36
dhryps	0,36	0,52	0,36	0,12	3,76	1,37	1,54	0,21
mflops	0,37	0,56	0,37	0,07	3,69	1,38	1,53	0,28
nbench	0,35	0,53	0,35	0,11	3,82	1,41	1,56	0,20

Lohe4-1(720x512), Configurazione B								
Indici di Performance	**Em**	**W_1**	**W_2**	**W_3**	**Potenza Impegnata**	**Speed Up**	**Max Speed-Up Ottenibile**	**Indice Bilanciamento**
unbalanced	0,33	0,33	0,33	0,33	4,00	0,87	1,57	0,92
bogomips	0,30	0,59	0,30	0,11	4,29	1,46	1,61	0,05
mwips	0,38	0,44	0,38	0,18	3,61	1,21	1,51	0,41
dhryps	0,36	0,52	0,36	0,12	3,76	1,41	1,53	0,14
mflops	0,37	0,56	0,37	0,07	3,69	1,41	1,53	0,26
nbench	0,35	0,53	0,35	0,11	3,82	1,44	1,55	0,09

APPENDICE A – Host Performance

In questa sezione sono riportati i dettagli architetturali delle workstation partecipanti alla griglia computazionale, nonché il risultato degli algoritmi di misurazione degli indici di performance utilizzati nei test.

Prometeo1, Prometeo2

Model	Generic i686
CPUs	Dual GenuineIntel® Pentium® III (Coppermine™) 866MHz
BogoMips	1730.15
Cache	256 KB
OS	RedHat Linux release 7.3 (Valhalla) kernel 2.4.18-3smp

```
################################################
Whetstone Double  Precision Benchmark in C/C++

Date          Mon May  2 10:55:15 CEST 2005

Model         prometeo2.pa.icar.cnr.it, generic i686

CPU           Pentium III (Coppermine)

Clock MHz     866.311

Cache         256 KB

OS            Red Hat Linux release 7.3 (Valhalla) kernel 2.4.18-3smp

Compiler      gcc version 2.96

Options       -O4

Run by        Fabio Collura

From          ICAR - CNR, Palermo

Email         fabio.collura@pa.icar.cnr.ut


Loop content                  Result              MFLOPS      MOPS   Seconds

N1 floating point      -1.12398255667393521       272.395              0.235
N2 floating point      -1.12187079889296859       217.519              2.060
N3 if then else         1.00000000000000000                 331.797     1.040
N4 fixed point         12.00000000000000000                 468.844     2.240
N5 sin,cos etc.         0.49902937281515342                  16.893    16.420
N6 floating point       0.99999987890803044        34.630             51.930
N7 assignments          3.00000000000000000                  84.400     7.300
```

```
N8 exp,sqrt etc.        0.75100163018458566                  6.562     18.900

MWIPS                                                      332.984            100.125
```

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Execution starts, 10000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    10000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
```

```
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:    1.0
Dhrystones per Second:                      1011804.4
```

```
Rolled Double Precision Linpack

    times are reported for matrices of order   100
      dgefa      dgesl      total       kflops     unit      ratio
 times for array with leading dimension of  201
      -0.00      -0.00      -0.00-164770917254598046515 2      -0.00      -0.00
      -0.00      -0.00      -0.00-164770917254598046515 2      -0.00      -0.00
       0.01      -0.00       0.01      68667       0.03       0.18
       0.00      -0.00       0.00     228889       0.01       0.05
 times for array with leading dimension of 200
      -0.00      -0.00      -0.00-123729077059313549312      -0.00      -0.00
      -0.00      -0.00      -0.00-123729077059313549312      -0.00      -0.00
       0.01       0.00       0.01      68667       0.03       0.18
       0.00       0.00       0.00     228889       0.01       0.05
Rolled Double  Precision 223.524 Mflops ; 10 Reps
Rolled Double Precision Linpack

     norm. resid      resid           machep         x[0]-1        x[n-1]-1
       1.9        8.39915160e-14  2.22044605e-16 -6.22835117e-14 -4.16333634e-14
```

```
BYTEmark* Native Mode Benchmark ver. 2 (10/95)
Index-split by Andrew D. Balsa (11/97)
Linux/Unix* port by Uwe F. Mayer (12/96,11/97)

TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          386.08  :       9.90  :       3.25
STRING SORT         :          35.301  :      15.77  :       2.44
BITFIELD            :      1.2679e+08  :      21.75  :       4.54
FP EMULATION        :          22.972  :      11.02  :       2.54
FOURIER             :            8189  :       9.31  :       5.23
ASSIGNMENT          :          6.0594  :      23.06  :       5.98
IDEA                :          974.16  :      14.90  :       4.42
HUFFMAN             :          385.21  :      10.68  :       3.41
NEURAL NET          :          8.0772  :      12.98  :       5.46
LU DECOMPOSITION    :          418.12  :      21.66  :      15.64
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 14.540
FLOATING-POINT INDEX: 13.781
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW================================
CPU                 : Dual GenuineIntel Pentium III (Coppermine) 866MHz
L2 Cache            : 256 KB
OS                  : Linux 2.4.18-3smp
C compiler          : gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110)
libc                : ld-2.2.5.so
MEMORY INDEX        : 4.048
INTEGER INDEX       : 3.342
FLOATING-POINT INDEX: 7.643
```

```
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
* Trademarks are property of their respective holder.
```

RedVision

Model	Generic i686
CPUs	Dual GenuineIntel® Intel® Xeon™ 1685MHz
BogoMips	3368.55
Cache	256 KB
OS	RedHat Linux release 7.3 (Valhalla) kernel 2.4.18-3smp

```
################################################
Whetstone Double  Precision Benchmark in C/C++

Date         Mon May  2 14:55:22 CEST 2005

Model        redvision.pa.icar.cnr.it, generic i686

CPU          Intel(R) Xeon(TM) CPU 1700MHz

Clock MHz    1685.167

Cache        256 KB

OS           Red Hat Linux release 7.3 (Valhalla) kernel 2.4.18-3smp

Compiler     gcc version 2.96

Options      -O4

Run by       Fabio Collura

From         ICAR - CNR, Palermo

Email        fabio.collura@pa.icar.cnr.it


Loop content                  Result              MFLOPS      MOPS   Seconds

N1 floating point     -1.12398255667393521       369.623              0.207
N2 floating point     -1.12187079889296859       202.107              2.650
N3 if then else        1.00000000000000000                 340.866     1.210
N4 fixed point        12.00000000000000000                 240.474     5.220
N5 sin,cos etc.        0.49902937281515342                  17.487    18.960
N6 floating point      0.99999987890803044        47.767             45.000
N7 assignments         3.00000000000000000                 168.519     4.370
N8 exp,sqrt etc.       0.75100163018458566                   5.596    26.490

MWIPS                                             382.779            104.107
```

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Execution starts, 10000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    10000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:    0.7
Dhrystones per Second:                      1442307.8
```

```
Rolled Double Precision Linpack

    times are reported for matrices of order   100
      dgefa      dgesl      total       kflops     unit      ratio
 times for array with leading dimension of  201
       0.00       0.00       0.00         inf       0.00       0.00
       0.01      -0.00       0.01       68667       0.03       0.18
      -0.00      -0.00      -0.00-164770917254598046515 2      -0.00      -0.00
       0.00      -0.00       0.00      686667       0.00       0.02
 times for array with leading dimension of 200
       0.00       0.00       0.00303941714232542842060 8       0.00       0.00
       0.00       0.00       0.00303941714232542842060 8       0.00       0.00
       0.00       0.00       0.00303941714232542842060 8       0.00       0.00
       0.00      -0.00       0.00      343333       0.01       0.04
Rolled Double  Precision 335.286 Mflops ; 10 Reps
Rolled Double Precision Linpack

     norm. resid      resid           machep         x[0]-1        x[n-1]-1
       1.9        8.39915160e-14  2.22044605e-16 -6.22835117e-14 -4.16333634e-14
```

```
BYTEmark* Native Mode Benchmark ver. 2 (10/95)
Index-split by Andrew D. Balsa (11/97)
Linux/Unix* port by Uwe F. Mayer (12/96,11/97)

TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          587.52  :      15.07  :       4.95
STRING SORT         :          52.438  :      23.43  :       3.63
BITFIELD            :      1.9245e+08  :      33.01  :       6.90
FP EMULATION        :          47.883  :      22.98  :       5.30
FOURIER             :          9436.8  :      10.73  :       6.03
ASSIGNMENT          :          12.779  :      48.62  :      12.61
IDEA                :          980.46  :      15.00  :       4.45
HUFFMAN             :          720.45  :      19.98  :       6.38
NEURAL NET          :          11.499  :      18.47  :       7.77
LU DECOMPOSITION    :          682.92  :      35.38  :      25.55
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 23.451
FLOATING-POINT INDEX: 19.141
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW================================
CPU                 : Dual GenuineIntel Intel(R) Xeon(TM) CPU 1685MHz
L2 Cache            : 256 KB
OS                  : Linux 2.4.18-3smp
C compiler          : gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110)
libc                : ld-2.2.5.so
MEMORY INDEX        : 6.807
INTEGER INDEX       : 5.225
FLOATING-POINT INDEX: 10.616
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
* Trademarks are property of their respective holder.
```

Ulisse

Model	Generic i686
CPUs	Intel® Pentium® III Mobile 1193MHz
BogoMips	2378.95
Cache	512 KB
OS	RedHat Linux release 7.3 (Valhalla) kernel 2.4.18-3

```
################################################
Whetstone Double  Precision Benchmark in C/C++

Date         Mon May  2 10:50:21 CEST 2005

Model        ulisse.pa.icar.cnr.it, generic i686

CPU          Intel(R) Pentium(R) III Mobile CPU      1200MHz

Clock MHz    1193.129

Cache        512 KB

OS           Red Hat Linux release 7.3 (Valhalla) kernel 2.4.18-3

Compiler     gcc version 2.96

Options      -O4

Run by       Fabio Collura

From         ICAR - CNR, Palermo

Email        fabio.collura@pa.icar.cnr.it


Loop content                  Result              MFLOPS      MOPS   Seconds

N1 floating point      -1.12398255667393521       405.314              0.218
N2 floating point      -1.12187079889296859       298.797              2.070
N3 if then else         1.00000000000000000                 453.626     1.050
N4 fixed point         12.00000000000000000                 611.658     2.370
N5 sin,cos etc.         0.49902937281515342                  23.390    16.370
N6 floating point       0.99999987890803044        47.618             52.130
N7 assignments          3.00000000000000000                 118.944     7.150
N8 exp,sqrt etc.        0.75100163018458566                   9.130    18.750

MWIPS                                              459.704            100.108
```

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Execution starts, 10000000 runs through Dhrystone
Execution ends
```

```
Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    10000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          134532808
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:    0.7
Dhrystones per Second:                      1415094.4
```

```
Rolled Double Precision Linpack

    times are reported for matrices of order   100
```

```
      dgefa      dgesl      total       kflops     unit      ratio
 times for array with leading dimension of  201
       0.00       0.00       0.00         inf       0.00       0.00
       0.01      -0.00       0.01       68667       0.03       0.18
      -0.00      -0.00      -0.00-164770917254598046515 2      -0.00      -0.00
       0.00       0.00       0.00      343333       0.01       0.04
 times for array with leading dimension of 200
       0.00       0.00       0.00308941714232542842060 8       0.00       0.00
       0.00       0.00       0.00308941714232542842060 8       0.00       0.00
      -0.00      -0.00      -0.00-411927293136495116288       -0.00      -0.00
       0.00       0.00       0.00      343333       0.01       0.04
Rolled Double  Precision 335.286 Mflops ; 10 Reps
Rolled Double Precision Linpack

     norm. resid      resid           machep         x[0]-1        x[n-1]-1
       1.9        8.39915160e-14  2.22044605e-16 -6.22835117e-14 -4.16333634e-14
```

```
BYTEmark* Native Mode Benchmark ver. 2 (10/95)
Index-split by Andrew D. Balsa (11/97)
Linux/Unix* port by Uwe F. Mayer (12/96,11/97)

TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          530.04  :      13.59  :       4.46
STRING SORT         :           49.26  :      22.01  :       3.41
BITFIELD            :      1.7359e+08  :      29.78  :       6.22
FP EMULATION        :          31.575  :      15.15  :       3.50
FOURIER             :           11263  :      12.81  :       7.19
ASSIGNMENT          :          8.3267  :      31.68  :       8.22
IDEA                :          1337.9  :      20.46  :       6.08
HUFFMAN             :          530.24  :      14.70  :       4.70
NEURAL NET          :          11.568  :      18.58  :       7.82
LU DECOMPOSITION    :          574.44  :      29.76  :      21.49
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 20.015
FLOATING-POINT INDEX: 19.205
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW================================
CPU                 : GenuineIntel Intel(R) Pentium(R) III Mobile CPU 1193MHz
L2 Cache            : 512 KB
OS                  : Linux 2.4.18-3
C compiler          : gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110)
libc                : ld-2.2.5.so
MEMORY INDEX        : 5.584
INTEGER INDEX       : 4.593
FLOATING-POINT INDEX: 10.652
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
* Trademarks are property of their respective holder.
```

Vision300

Model	Generic i686
CPUs	Intel® Pentium® II (Klamath™) 300MHz

BogoMips	599.65
Cache	512 KB
OS	RedHat Linux release 7.3 (Valhalla) kernel 2.4.18-3

```
################################################
Whetstone Double  Precision Benchmark in C/C++

Date         Tue May  3 18:36:22 CEST 2005

Model        vision300.pa.icar.cnr.it, generic i686

CPU          Pentium II (Klamath)

Clock MHz    300.689

Cache        512 KB

OS           Red Hat Linux release 7.3 (Valhalla) kernel 2.4.18-3

Compiler     gcc version 2.96

Options      -O4

Run by       Fabio Collura

From         ICAR - CNR, Palermo

Email        fabio.collura@pa.icar.cnr.it


Loop content                  Result              MFLOPS      MOPS   Seconds

N1 floating point      -1.12398255667393521        91.680              0.320
N2 floating point      -1.12187079889296859        71.805              2.860
N3 if then else         1.00000000000000000                109.068     1.450
N4 fixed point         12.00000000000000000                156.782     3.070
N5 sin,cos etc.         0.49902937281515342                  5.665    22.440
N6 floating point       0.99999987890803044        23.862             34.540
N7 assignments          3.00000000000000000                 28.322     9.970
N8 exp,sqrt etc.        0.75100163018458566                  2.195    25.900

MWIPS                                             151.964            100.550
```

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Execution starts, 10000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
```

```
Bool_Glob:            1
        should be:    1
Ch_1_Glob:            A
        should be:    A
Ch_2_Glob:            B
        should be:    B
Arr_1_Glob[8]:        7
        should be:    7
Arr_2_Glob[8][7]:     10000010
        should be:    Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:           134532808
        should be:    (implementation-dependent)
  Discr:              0
        should be:    0
  Enum_Comp:          2
        should be:    2
  Int_Comp:           17
        should be:    17
  Str_Comp:           DHRYSTONE PROGRAM, SOME STRING
        should be:    DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:           134532808
        should be:    (implementation-dependent), same as above
  Discr:              0
        should be:    0
  Enum_Comp:          1
        should be:    1
  Int_Comp:           18
        should be:    18
  Str_Comp:           DHRYSTONE PROGRAM, SOME STRING
        should be:    DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:            5
        should be:    5
Int_2_Loc:            13
        should be:    13
Int_3_Loc:            7
        should be:    7
Enum_Loc:             1
        should be:    1
Str_1_Loc:            DHRYSTONE PROGRAM, 1'ST STRING
        should be:    DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:            DHRYSTONE PROGRAM, 2'ND STRING
        should be:    DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:    3.0
Dhrystones per Second:                      336134.5
```

```
Rolled Double Precision Linpack

    times are reported for matrices of order   100
      dgefa      dgesl      total       kflops     unit      ratio
 times for array with leading dimension of  201
       0.01       0.00       0.01      68667       0.03       0.18
       0.02      -0.00       0.02      34333       0.06       0.36
       0.01       0.00       0.01      68667       0.03       0.18
```

```
       0.01       -0.00       0.01       52821       0.04       0.23
 times for array with leading dimension of 200
       0.02        0.00       0.02       34333       0.06       0.36
       0.01        0.00       0.01       68667       0.03       0.18
       0.02       -0.00       0.02       34333       0.06       0.36
       0.02        0.00       0.02       42917       0.05       0.29
Rolled Double  Precision 41.911 Mflops ; 10 Reps
Rolled Double Precision Linpack

     norm. resid      resid           machep         x[0]-1        x[n-1]-1
       1.9        8.39915160e-14  2.22044605e-16 -6.22835117e-14 -4.16333634e-14
```

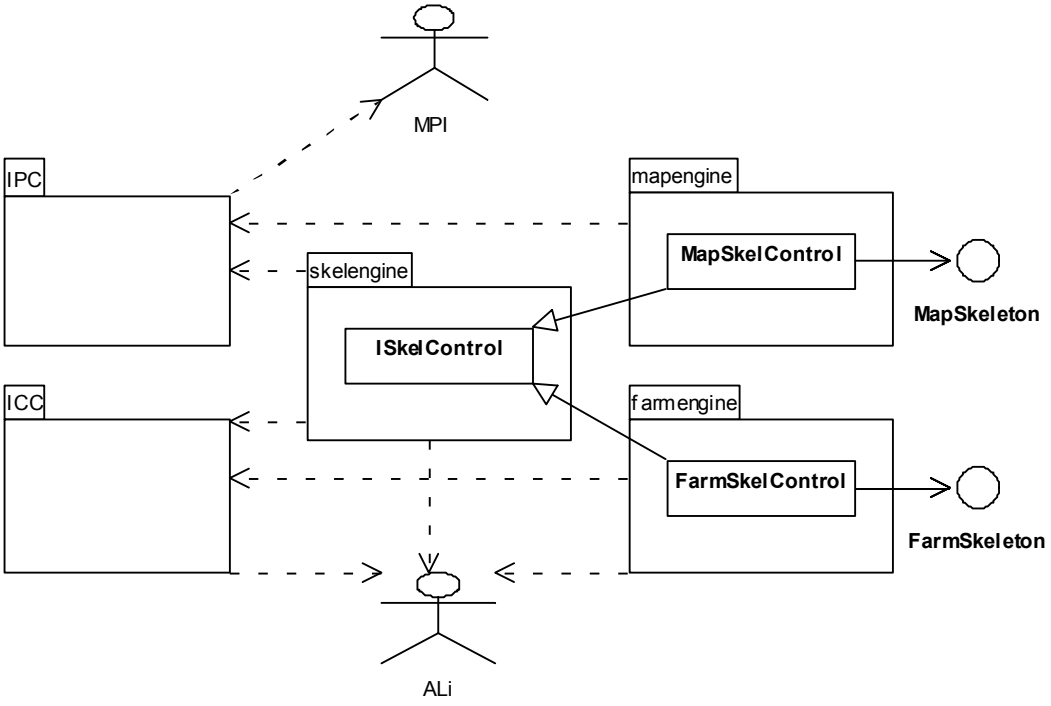
```
BYTEmark* Native Mode Benchmark ver. 2 (10/95)
Index-split by Andrew D. Balsa (11/97)
Linux/Unix* port by Uwe F. Mayer (12/96,11/97)

TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          121.98  :       3.13  :       1.03
STRING SORT         :           11.69  :       5.22  :       0.81
BITFIELD            :      4.2257e+07  :       7.25  :       1.51
FP EMULATION        :          7.7106  :       3.70  :       0.85
FOURIER             :          2754.1  :       3.13  :       1.76
ASSIGNMENT          :          1.7999  :       6.85  :       1.78
IDEA                :           327.3  :       5.01  :       1.49
HUFFMAN             :          129.06  :       3.58  :       1.14
NEURAL NET          :          2.7132  :       4.36  :       1.83
LU DECOMPOSITION    :          103.48  :       5.36  :       3.87
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 4.740
FLOATING-POINT INDEX: 4.183
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW================================
CPU                 : GenuineIntel Pentium II (Klamath) 301MHz
L2 Cache            : 512 KB
OS                  : Linux 2.4.18-3
C compiler          : gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110)
libc                : ld-2.2.5.so
MEMORY INDEX        : 1.296
INTEGER INDEX       : 1.105
FLOATING-POINT INDEX: 2.320
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
* Trademarks are property of their respective holder.
```

APPENDICE B – Progetto della libreria CCSkel

In questa sezione è riportata la documentazione del progetto della libreria CCSkel (v3.4), per la creazione di Componenti Configurabili basati su Skeleton Paralleli Map e Farm con bilanciamento del carico computazionale.

B.1 – Architettura della libreria

La libreria CCSkel è organizzata nei seguenti package:



Il package ***mapengine*** contiene le classi per la realizzazione di uno skeleton parallelo di tipo Map

Il package ***farmengine*** contiene le classi per la realizzazione di uno skeleton parallelo di tipo Farm

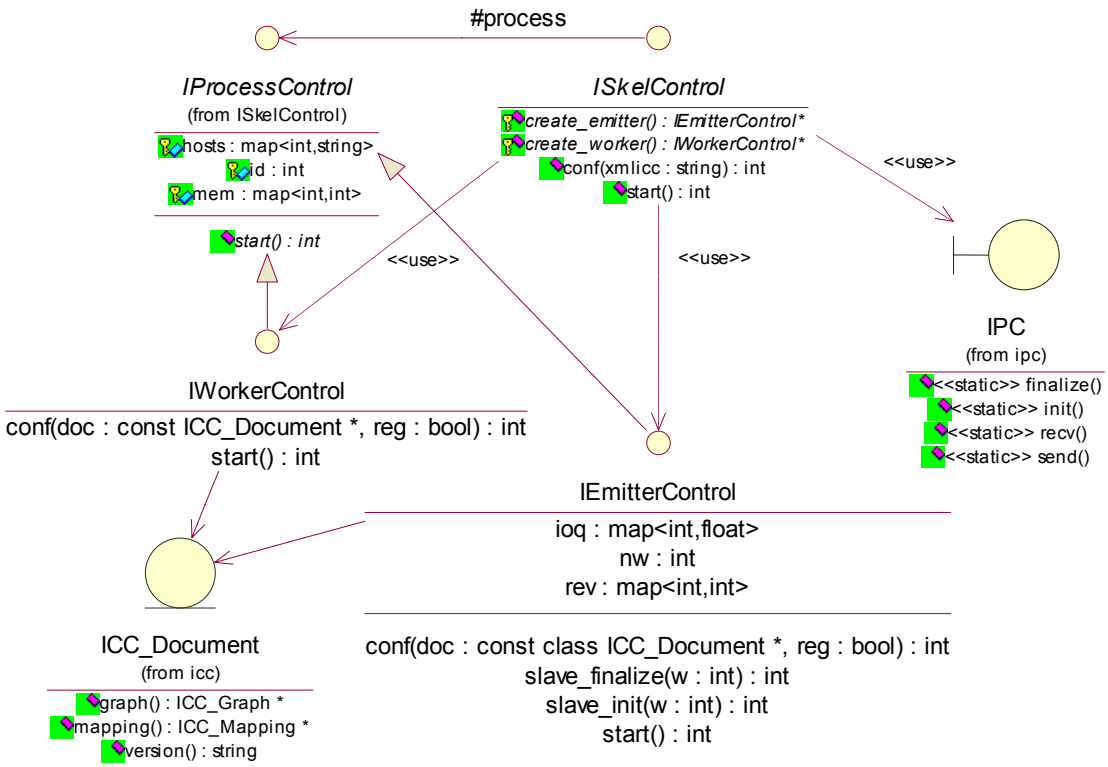
Il package ***skelengine*** contiene le classi di base dello skeleton, comuni ad entrambe le versioni Map/Farm

Il package ***ICC*** fornisce il supporto per l'interpretazione dell'XML-ICC contenente la configurazione del componente che ospita lo skeleton.

La libreria ***IPC*** fornisce il supporto per l'attivazione del componente e lo scambio di messaggi a run-time. Essa è implementata tramite il sistema MPI

B.2 – Modello degli Oggetti

Di seguito è riportato il diagramma delle classi relativo al package ***skelengine***.



La classe ***IPC*** fornisce il set di primitive di comunicazione tra processi:

- **init**, inizializzazione
- **send**, invio di un messaggio
- **recv**, ricezione di un messaggio
- **fini**, finalizzazione

La classe ***ISkeletonControl*** definisce l'interfaccia di controllo per un generico skeleton parallelo. Le funzionalità fornite sono:

- **conf**, per la configurazione del grafo virtuale/fisico dei processi
- **start**, per l'avvio dello skeleton parallelo (esecuzione)

La classe ***IWorkerControl*** definisce l'interfaccia di controllo per un generico processo Worker dello skeleton parallelo.

La classe ***IEmitterControl*** definisce l'interfaccia di controllo per un generico processo Emitter dello skeleton parallelo.

La classe ***IProcessControl*** definisce la generica interfaccia di processo.

Nella pagina successiva è riportato il diagramma delle classi relativo al package ***mapengine***.

La classe ***MapSkelControl*** implementa l'interfaccia *ISkelControl* secondo uno skeleton parallelo di tipo map.

La classe ***MapWorkerControl*** implementa l'interfaccia *IWorkerControl* per il controllo di un processo Worker relativo ad uno skeleton parallelo di tipo map.

La classe ***MapEmitterControl*** implementa l'interfaccia *IEmitterControl* per il controllo di un processo Emitter relativo ad uno skeleton parallelo di tipo map. Le funzionalità di base fornite sono:

- **distrib_policy**, applica la politica di distribuzione dei dati
- **distrib**, distribuisce le porzioni di dato ai Worker
- **collect**, recupera le porzioni di dato dai Worker

La classe ***MapSkeleton*** definisce la classe base per l'implementazione delle funzionalità di uno skeleton parallelo di tipo map.

La classe ***IMapEmitter*** definisce l'interfaccia funzionale del processo Emitter e le funzionalità di controllo parametrizzate nei due tipi **Input,Output** che rappresentano il tipo dei dati in ingresso ed in uscita dall'Emitter. Le funzionalità previste per il processo Emitter sono:

- **parmain**, funzione principale dell'Emitter
- **distrib_strategy<Input>**, applica la strategia di suddivisione del dato di tipo Input
- **collect_strategy<Output>**, applica la strategia di raccolta del dato di tipo Output

Le funzionalità di controllo sono:

- **distrib_item<Input>**, avvia la distribuzione di un dato di tipo Input
- **collect_item<Output>**, avvia la raccolta di un dato di tipo Output
- **scatter**, suddivide un dato secondo la politica distribuzione
- **gather**, riassembla un dato

La classe ***MapEmitter*** implementa le funzionalità di controllo tipizzate definite nell'interfaccia *IMapEmitter<Input,Outout>* attraverso le funzionalità di base fornite dalla classe di controllo per un processo Emitter (*MapEmitterControl*).

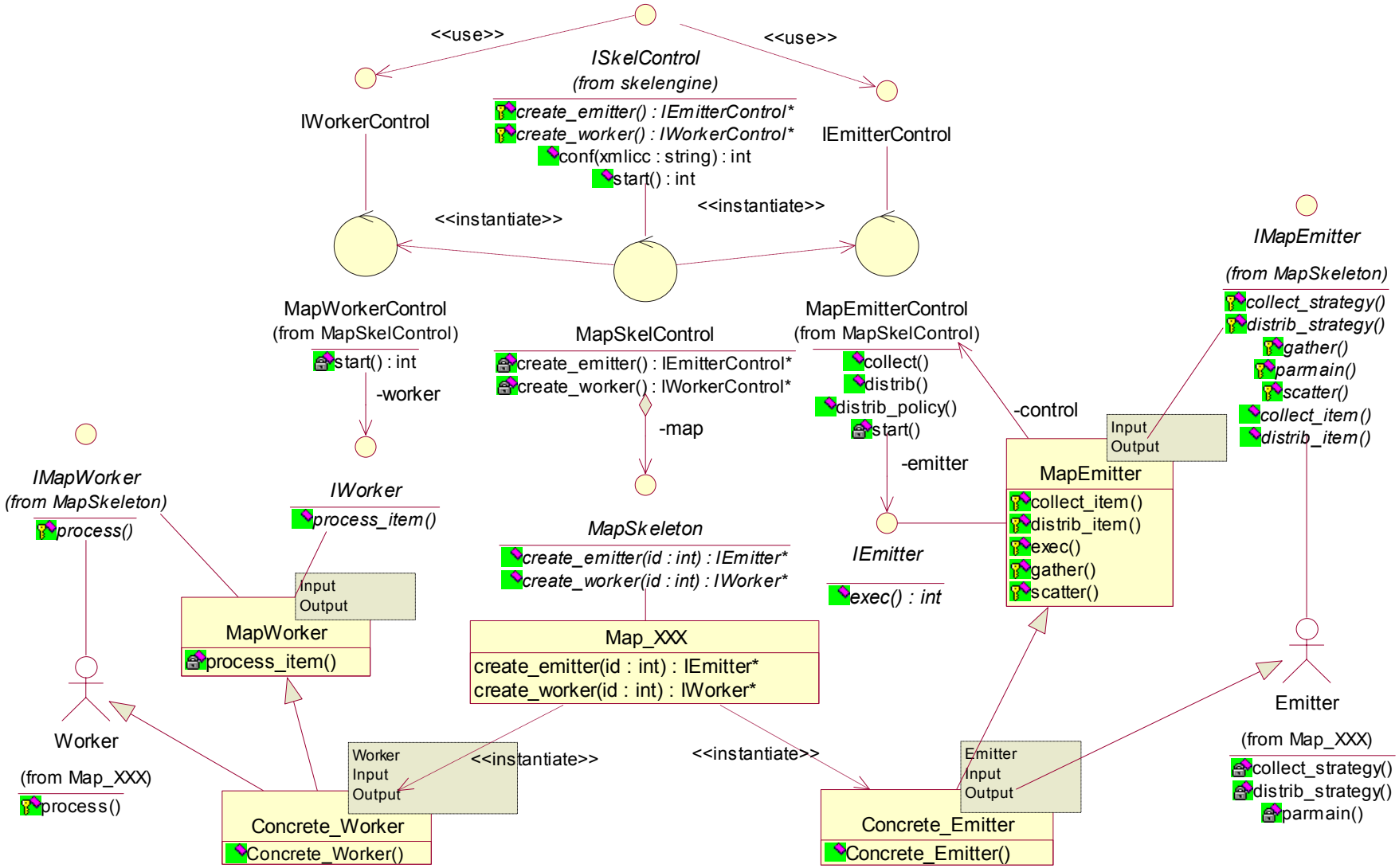
La classe ***IWorker*** definisce la generica interfaccia funzionale del processo Worker. Le funzionalità previste sono:

- **process_item**, per il processamento di un generico dato distribuito dall'Emitter

La classe ***IMapWorker*** definisce la specifica interfaccia funzionale del processo Worker parametrizzata nei due tipi **Input,Output** che rappresentano il tipo dei dati in ingresso ed in uscita dal Worker. Le funzionalità previste sono:

- **process<Input,Output>**, per il processamento di un dato di tipo Input con un risultato di tipo Output

La class ***MapWorker*** implementa la generica interfaccia funzionale del processo Worker (*IWorker*) attraverso la specifica interfaccia funzionale tipizzata (*IMapWorker<Input,Output>*).


<<use>>
<<use>>
ISkelControl
(from skelengine)
create_emitter() : IEmitterControl*
create_worker() : IWorkerControl*
conf(xmlicc : string) : int
start() : int
IWorkerControl
IEmitterControl
<<instantiate>>
<<instantiate>>
MapWorkerControl
(from MapSkelControl)
start() : int
-worker
MapSkelControl
create_emitter() : IEmitterControl*
create_worker() : IWorkerControl*
-map
MapEmitterControl
(from MapSkelControl)
collect()
distrib()
distrib_policy()
start()
-emitter
-control
IMapEmitter
(from MapSkeleton)
collect_strategy()
distrib_strategy()
gather()
parmain()
scatter()
collect_item()
distrib_item()
Input
Output
MapEmitter
collect_item()
distrib_item()
exec()
gather()
scatter()
IMapWorker
(from MapSkeleton)
process()
IWorker
process_item()
MapSkeleton
create_emitter(id : int) : IEmitter*
create_worker(id : int) : IWorker*
IEmitter
exec() : int
Input
Output
MapWorker
process_item()
Map_XXX
create_emitter(id : int) : IEmitter*
create_worker(id : int) : IWorker*
Worker
(from Map_XXX)
process()
Worker
Input
Output
<<instantiate>>
<<instantiate>>
Emitter
Input
Output
Concrete_Worker
Concrete_Worker()
Concrete_Emitter
Concrete_Emitter()
Emitter
(from Map_XXX)
collect_strategy()
distrib_strategy()
parmain()

Nella pagina successiva è riportato il diagramma delle classi relativo al package ***farmengine***.

La classe ***FarmSkelControl*** implementa l'interfaccia *ISkelControl* secondo uno skeleton parallelo di tipo farm.

La classe ***FarmWorkerControl*** implementa l'interfaccia *IWorkerControl* per il controllo di un processo Worker relativo ad uno skeleton parallelo di tipo farm.

La classe ***FarmEmitterControl*** implementa l'interfaccia *IEmitterControl* per il controllo di un processo Emitter relativo ad uno skeleton parallelo di tipo farm. Le funzionalità di base fornite sono:

- **distrib_policy**, applica la politica di distribuzione dei dati
- **distrib**, distribuisce un dato al primo Worker libero
- **collect**, recupera un dato elaborato dal Worker

La classe ***FarmSkeleton*** definisce la classe base per l'implementazione delle funzionalità di uno skeleton parallelo di tipo farm.

La classe ***IFarmEmitter*** definisce l'interfaccia funzionale del processo Emitter e le funzionalità di controllo parametrizzate nei tretipi **Index**,**Input,Output** che rappresentano il tipo dei dati in ingresso ed in uscita dall'Emitter. Le funzionalità previste per il processo Emitter sono:

- **parmain**, funzione principale dell'Emitter
- **distrib_strategy<Index,Input>**, applica la strategia di distribuzione del dato di tipo Input a partire dalla relativa variabile di controllo Index
- **collect_strategy<Index,Input,Output>**, applica la strategia di raccolta del dato di tipo Output a partire dalla relativa variabile di controllo Index e dall'originario dato di tipo Input

Le funzionalità di controllo sono:

- **distrib_item<Index,Input>**, avvia la distribuzione di un dato di tipo Input relativamente alla variabile di controllo Index

La classe ***FarmEmitter*** implementa le funzionalità di controllo tipizzate definite nell'interfaccia *IFarmEmitter<Index,Input,Outout>* attraverso le funzionalità di base fornite dalla classe di controllo per un processo Emitter (*FarmEmitterControl*).

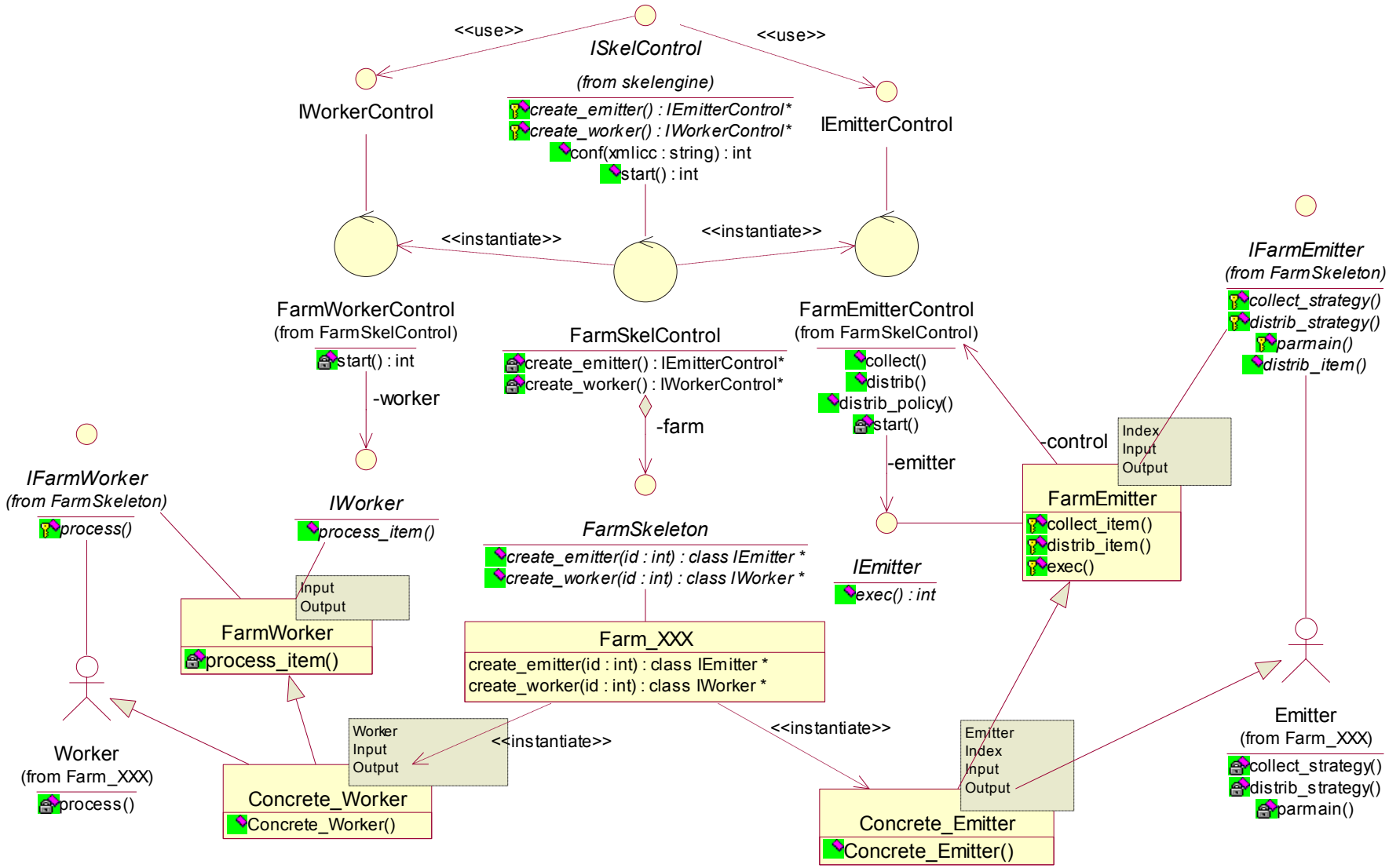
La classe ***IWorker*** definisce la generica interfaccia funzionale del processo Worker. Le funzionalità previste sono:

- **process_item**, per il processamento di un generico dato distribuito dall'Emitter

La classe ***IFarmWorker*** definisce la specifica interfaccia funzionale del processo Worker parametrizzata nei due tipi **Input,Output** che rappresentano il tipo dei dati in ingresso ed in uscita dal Worker. Le funzionalità previste sono:

- **process<Input,Output>**, per il processamento di un dato di tipo Input con un risultato di tipo Output

La class ***FarmWorker*** implementa la generica interfaccia funzionale del processo Worker (*IWorker*) attraverso la specifica interfaccia funzionale tipizzata (*IFarmWorker<Input,Output>*).


ISkelControl
(from skelengine)
create_emitter() : IEmitterControl*
create_worker() : IWorkerControl*
conf(xmlicc : string) : int
start() : int
<<use>>
<<use>>
IWorkerControl
IEmitterControl
<<instantiate>>
<<instantiate>>
FarmWorkerControl
(from FarmSkelControl)
start() : int
-worker
FarmSkelControl
create_emitter() : IEmitterControl*
create_worker() : IWorkerControl*
-farm
FarmEmitterControl
(from FarmSkelControl)
collect()
distrib()
distrib_policy()
start()
-emitter
-control
IFarmEmitter
(from FarmSkeleton)
collect_strategy()
distrib_strategy()
parmain()
distrib_item()
Index
Input
Output
FarmEmitter
collect_item()
distrib_item()
exec()
IFarmWorker
(from FarmSkeleton)
process()
IWorker
process_item()
Input
Output
FarmWorker
process_item()
FarmSkeleton
create_emitter(id : int) : class IEmitter *
create_worker(id : int) : class IWorker *
IEmitter
exec() : int
Farm_XXX
create_emitter(id : int) : class IEmitter *
create_worker(id : int) : class IWorker *
Worker
(from Farm_XXX)
process()
Worker
Input
Output
<<instantiate>>
<<instantiate>>
Concrete_Worker
Concrete_Worker()
Emitter
Index
Input
Output
Concrete_Emitter
Concrete_Emitter()
Emitter
(from Farm_XXX)
collect_strategy()
distrib_strategy()
parmain()

REFERENCES

[1] M. Cole. Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Parallel and Distributed Computing. Pitman, 1989

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, *P^3L: A structured high level programming language and its structured support*, Concurrency: Practice and Experience, 7(3):225-255, May 1995

[3] A. Benoit, M. Cole, S. Gilmore and J. Hillstonm. "Evaluating the Performance of Skeleton-Based High Level Parallel Programs", ICCS 2004, Kraków, Poland, Springer-Verlag LNCS Vol. 3038, pp. 289-296, June 6-9, 2004

[4] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo "Components for high performance Grid programming in the Grid.it Project". Intl. Workshop on Component Models and Systems for Grid Applications.

[5] M. Danelutto, "QoS in parallel programming through application managers" Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing, Lugano, Feb. 2005

[6] S. Lombardo, A. Machì. "A model for a component based grid-aware scientific library service". Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, pp. 423-428

[7] F.Berman, G.C. Fox, A.J.G.Hey: Grid Computing. Making the Global Infrastructure a Reality. Wiley 2003

[8] Web Service Level Agreements (WSLA) Project - SLA Compliance Monitoring for e-Business on demand http://www.research.ibm.com/wsla/

[9] The Workflow Management Coalition, www.wfmc.org

[10] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. 7th International Conference on Cooperative Information Systems (CoopIS 2000), LNCS volume 1901, pages 18-29. Springer-Verlag, Berlin, 2000.

[11] Business Process Execution Language for Web Services, the specification. http://www-106.ibm.com/developerworks/library/ws-bpel/

[12] Workflow Process Definition Interface -- XML Process Definition Language (XPDL) Document Number WFMC-TC-1025

[13] The Workflow Reference Model (WFMC-TC-1003, 19-Jan-95, 1.1)

[14] Unified Modelling Language, OMG Specification, www.omg.org

[15]Clemens Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.

[16] R. Armstrom, D. Gannon, K. Keahey, S.Kohn, L.McInnes, S. Parker, B. Smolinsk. "Toward a common component architecture for high-performance scientific computing". In Conference on High Performance Distributed Computing, 1999

[17] A. Machì., F.Collura, S. Lombardo," Dependable Execution of Workflow Activities on a Virtual Private Grid Middleware", submitted to "2nd International Workshop on Active and Programmable Grids Architectures and Components APGAC'05-ICCS-2005 - Atlanta, USA; 22-25 May 2005

[18] Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects" Wiley & Sons in 2000, ISBN 0-471-60695-2.

[19] F.Collura,S.Lombardo,A.Machì,"Legacy Software Integration Environment: Metodologie, patterns e tools per l'integrazione nell'ambiente di programmazione Grid.it", Rapport Tecnico ICAR, n. RT-ICAR-PA-04-15

[20] A.Machi, F.Collura, F.Nicotra, "Detection of Irregular Linear Scratches in Aged Motion Picture Frames and Restoration using Adaptive Masks",IASTED SIP02, Kawaii USA 2002

[21] A.Machì, F.Collura, F.Nicotra, "An Interactive Distributed Environment for Digital Film Restoration" A. Laganà et als. (Eds.) Computational Science and its Applications ICCSA 2004 Perugia, Italy