Politecnico di Milano

Dipartimento di Matematica

Dispensa di

Elementi di Programmazione Avanzata in R

a cura di

Laura Azzimonti
Stefano Baraldo

 Autori: Laura Azzimonti e Stefano Baraldo, Dipartimento di Matematica "F. Brioschi", Politecnico di Milano, piazza Leonardo da Vinci 32, 20133 Milano; email: laura.azzimonti@gmail.com, stefano.baraldo@gmail.com.

Indice

Introduzione

`R` [1] è un linguaggio di programmazione, nonché un programma per interpretare tale linguaggio, orientato ad applicazioni in ambito statistico, ideato nel 1993 da Ross Ihaka e Robert Gentleman. Nacque come interprete per un linguaggio di programmazione funzionale simile a Scheme, ma in breve tempo gli ideatori cominciarono ad adottare le strutture sintattiche del software commerciale S, ideato da Rick Becker, John Chambers e Allan Wilks a metà anni '70. Ad oggi, pur essendo un software open source autonomo e ampiamente supportato da una nutritissima comunità, `R` mantiene una compatibilità quasi piena con S e S-PLUS, differenziandosene, a livello utente, solo per alcuni dettagli come le regole di *scoping*.

Il codice sorgente di `R` è scritto in linguaggio C, ed è compilabile per tutte le principali piattaforme. Supporta in modo naturale l'implementazione di pacchetti aggiuntivi ed estensioni scritte in C, C++ e Fortran (nonché naturalmente nel linguaggio R stesso), ma varie estensioni sono disponibili per l'interazione con programmi scritti in altri linguaggi, tra cui ad esempio Python, Perl e SQL.

La comunità di `R` è ad oggi molto estesa, e continua a migliorare ed estendere il programma rendendo disponibili vari pacchetti sul Comprehensive R Archive Network (*CRAN*). Tali estensioni rispondono da un lato alla necessità rendere disponibili a tutti metodi statistici (e non) anche molto avanzati, dall'altro a quella di sopperire alle limitazioni del programma base in fatto di efficienza. In questo lavoro daremo alcuni cenni riguardanti questo secondo aspetto: innanzitutto vengono descritti alcuni aspetti chiave del funzionamento del programma, dopodiché vengono esposti alcuni elementi di tecniche avanzate come la programmazione a oggetti, la chiamata di funzioni scritte in altri linguaggi di programmazione e il calcolo parallelo, quest'ultimo argomento corredato anche di alcuni casi test.

All'interno dell'elaborato non verranno esposti i fondamenti della sintassi del linguaggio `R` , poiché sarebbe fuori dallo scopo del lavoro. In ogni caso la trattazione è pensata per essere accessibile, con poco sforzo, anche a chi ancora non conosce questo linguaggio. Chi fosse interessato ad approfondirne

le basi è invitato a consultare i manuali [2] e [3].

Questo elaborato è stato redatto facendo riferimento a R versione 2.11.0.

Capitolo 1

Come funziona R

L'obiettivo di questo capitolo non è quello di fornire una descrizione completa ed esaustiva del funzionamento di `R` e della struttura del suo codice sorgente, ma di dare un'idea generale delle principali meccaniche sottostanti al linguaggio `R` e all'interfaccia di uso abituale.

1.1 La struttura dati `SEXPREC`

Lavorando normalmente in `R` è possibile utilizzare oggetti di natura molto varia, tra cui naturalmente variabili numeriche, stringhe, vettori e matrici, oppure strutture più complesse come liste, dataframes (tabelle di dati compositi), factors (vettori di etichette con un numero prestabilito di livelli) e altri. Tutti questi oggetti sono in realtà memorizzati da `R` come puntatori a oggetti di un unico tipo, una `struct` denominata nel sorgente C `SEXPREC`, la cui definizione è la seguente:

```
1  typedef struct SEXPREC {
2     SEXPREC_HEADER;
3     union {
4        struct primsxp_struct primsxp;
5        struct symsxp_struct symsxp;
6        struct listsxp_struct listsxp;
7        struct envsxp_struct envsxp;
8        struct closxp_struct closxp;
9        struct promsxp_struct promsxp;
10       } u;
11 } SEXPREC, *SEXP;
```

La macro alla riga 2 fa riferimento a `struct` che includono informazioni riguardanti il tipo dell'oggetto memorizzato (nonché dei puntatori usati dal

garbage collector), mentre la `union u` contiene varie triplette di puntatori a `SEXPREC` (tranne `primsxp`, che è un intero per motivi che saranno chiariti nel paragrafo 1.4), utilizzate in modo diverso a seconda del tipo di dato per il quale è istanziato il singolo oggetto `SEXPREC`. Il dato effettivo, ad esempio l'`int` che contiene il valore numerico di un oggetto `R` di tipo `integer`, viene allocato nello spazio in memoria immediatamente successivo al relativo `SEXPREC`, in una posizione e per una lunghezza che sono quindi univocamente determinate dalla posizione del relativo oggetto `SEXPREC` (di dimensione nota a priori) e dal tipo di dato.

All'interno dei sorgenti, o all'interno di codici aggiuntivi che intendono farne uso, i `SEXPREC` sono gestiti attraverso puntatori, denominati `SEXP`, e gestiti nel codice attraverso una vasta gamma di macro (ne vedremo un esempio di utilizzo nel paragrafo 3.1. `R` crea uno di questi oggetti per *ogni* struttura da esso utilizzata, incluse non solo le variabili classiche, ma anche le funzioni e gli environments che contengono le variabili. Questa uniformità nella struttura di dati e operatori, se da un lato diminuisce molto le prestazioni computazionali, rende il linguaggio estremamente accessibile.

1.2 La gestione della memoria

A differenza di quanto avviene nel C e C++, in R l'utente non deve preoccuparsi dell'allocazione (dinamica) o cancellazione della memoria esplicitamente.

Per quando riguarda l'allocazione, l'allocator di `R` la effettua automaticamente all'assegnazione di una variabile o alla chiamata di una funzione. Per evitare copie inutili, `SEXPREC_HEADER` contiene al suo interno la variabile `unsigned int named`, che assume valore 0, 1 o 2 a seconda che l'oggetto in questione sia referenziato da nessuna, una o più variabili in `R` . In situazioni come un assegnamento `b <- a`, finché `a` o `b` non vengono modificati non è necessario allocare un'ulteriore copia del dato a cui le due variabili fanno riferimento; questa nuova allocazione in effetti non viene effettuata, e un'eventuale funzione (primitiva, si veda il paragrafo 1.4) che debba agire su `a` o `b` eseguirà prima il controllo

```
1 if (NAMED(x) == 2)
2    x = duplicate(x);
```

duplicando il dato solo al momento opportuno.

Analogamente ad altri linguaggi interpretati, il compito di liberare la memoria inutilizzata è affidato a un garbage collector. Il garbage collector di `R` è di tipo generazionale, ovvero classifica gli oggetti in memoria in 3

"generazioni" ed effettua continuamente cicli di "raccolta della spazzatura", all'interno dei quali cerca quali oggetti non sono più referenziati da `R` e rende la memoria da essi occupata disponibile per un nuovo utilizzo. Ogni 20 cicli di raccolta tra gli oggetti più giovani viene effettuato un ciclo sugli oggetti delle due generazioni più giovani, e ogni 5 cicli di quest'ultimo tipo ne viene effettuato uno su tutti gli oggetti.

Avere presente questo meccanismo si rivelerà importante al paragrafo 3.1, quando vedremo come maneggiare oggetti `SEXP` all'interno di codici C creati dall'utente.

1.3 Il loop *read-parse-evaluate*

Come in ogni linguaggio interpretato, ogni operazione richiesta a `R` dall'utente deve passare attraverso 2 fasi fondamentali: il *parsing* delle istruzioni, ovvero l'individuazione, nel testo delle istruzioni, delle strutture sintattiche proprie del linguaggio e dei simboli definiti, e l'*evaluation*, ovvero la valutazione dei valori effettivi dei simboli individuati e lo svolgimento delle operazioni su tali valori.

Il parser di `R` è stato creato attraverso GNU Bison 2.3, un generatore di parser in base a una grammatica fornita. Nel caso di `R` la grammatica è sostanzialmente quella di S, con differenze minime. Le strutture sintattiche riconosciute sono riassunte dalla Tabella 1.1 (tratta da [4]), dove *expr* rappresenta a sua volta un'espressione, *op* e *unaryOp* sono rispettivamente un operatore binario e unario, *nameString* e *constNameString* rappresentano stringhe/nomi sintattici e costanti numeriche/logiche, *Name* è il nome di una variabile mentre *sublist* e *formlist* sono liste di espressioni o combinazioni *Name*=*expr* separati da virgole.

Il parser individua le espressioni valide, ma non è detto che tali espressioni abbiano senso al momento dell'evaluation. Ad esempio, utilizzando la funzione di `R` `parse`, con la quale possiamo ottenere la *unevaluated expression* da una stringa di codice, si può lanciare

```
> a <- "ciao"
> uneval <- parse(text="a+1")
> uneval
expression(a+1)
attr(,"srcfile")
<text>
```

senza ottenere errori. Tuttavia, essendo il + un operatore non definito per stringhe, l'evaluator si accorgerà dell'inconsistenza.

Tipo espressione	**Regola**
Call	*expr* (*sublist*)
Binary	*expr op expr* *expr* $ *nameString* *expr* @ *nameString* *nameString* :: *nameString* *nameString* ::: *nameString*
Unary	*unaryOp expr*
Subset	*expr* [*sublist*] *expr* [[*sublist*]]
Conditional	if(*expr*) if(*expr*) else *expr*

Iteration	for(*Name* in *expr*) *expr*
	while(*expr*) *expr*
	repeat *expr*

Grouping	(*expr*) { *exprlist* }
Function	function(*formlist*) *expr*
Flow	break next
Token	*constNameString*

Tabella 1.1: Lista delle espressioni valide riconosciute dal parser.

```
> eval(uneval)
Errore in a + 1 : argomento non numerico trasformato in
    operatore binario
```

Quando un comando viene lanciato, il parser estrae le espressioni valide e le inserisce in un vettore opportunamente ordinato, che viene poi passato all'evaluator. *Tutte* le operazioni in `R` vengono risolte dall'evaluator attraverso chiamate a funzioni, compresi l'assegnamento o l'effettuazione di cicli. Ad esempio le istruzioni per una condizione logica di tipo `if`

```
if(a>b){
    a-b
} else {
    b-a
}
```

vengono eseguite come se fossero la chiamata seguente, che volendo può essere effettuata in qualsiasi codice:

```
1 `if`(`>`(a,b),`-`(a,b),`-`(b,a))
```

Le istruzioni legate a parole chiave od operatori, come `if`, `while`, `+`, `[<-` e altri, sono effettivamente codificate in `R` come funzioni, utilizzabili nella forma consueta se richiamate tra backticks. In questo modo, essendo noto l'ordine con cui effettuare le varie chiamate grazie al parser, qualsiasi codice può venire eseguito come una complessa chiamata di funzioni in sequenza o annidate.

Fino ad ora siamo stati vaghi sul concetto di funzione, facendo riferimento in particolare a ciò che l'utente utilizza come tali all'interno del linguaggio `R` . In realtà si possono individuare due tipi principali di funzioni: quelle ordinarie, definite tramite linguaggio `R` , e quelle primitive, il cui corpo è in realtà scritto nel sorgente del programma. Queste ultime hanno un'importanza fondamentale, in quanto *la valutazione di un'espressione atomica viene effettuata solamente se richiesta da una funzione primitiva.*

1.4 Funzioni ordinarie e primitive

Una *funzione ordinaria*, o meglio un oggetto di tipo *closure*, è costituito da un `SEXPREC` che usa i puntatori definiti in `struct closxp_struct closxp` (si veda 1.1), ovvero `*formals`, `*body` e `*env`. Questi tre campi sono i componenti fondamentali dell'oggetto closure, e possono essere visualizzati attraverso apposite funzioni:

```
1  > sprod <- function(x1=1, x2=1){
2  +  return(x1%*%x2)
3  + }
4  > formals(sprod)
5  $x1
6  $x2
7  > body(sprod)
8  {
9     return(x1 %*% x2)
10 }
11 > environment(fun)
12 <environment: R_GlobalEnv>
```

Il campo *formals* contiene gli argomenti formali, ovvero i nomi degli argomenti da passare contenuti nella definizione della funzione; il campo *body* contiene il vero e proprio codice da eseguire quando la funzione viene lanciata; infine, *environment* fa riferimento all'ambiente in cui vanno cercate le variabili non locali alla funzione.

Nel caso dell'esempio precedente, l'*enclosing environment* della funzione `sprod`, al momento della definizione, viene impostato per default, ovvero quello globale della sessione di `R` . Una funzione ha tipicamente un environment differente da `R_GlobalEnv` solamente se esse proviene da un pacchetto in cui è stato specificato un *namespace*, oppure quando la funzione è stata chiamata a sua volta da un'altra funzione; in tal caso l'enclosing environment è costituito dalle variabili locali della funzione chiamante. La ricerca delle variabili viene effettuata secondo un criterio gerarchico: se un nome non viene trovato nello scope locale, viene cercato nell'enclosing environment, poi nell'enclosing environment di quest'ultimo, e così via, fino a quello globale.

L'evaluation di una funzione ordinaria procede in 3 passi:

- *Matching* degli argomenti passati con gli argomenti formali. Poiché le funzioni utilizzate in `R` hanno spesso molti argomenti, per comodità d'uso non è obbligatorio che essi vengano passati seguendo l'ordine specificato dalla definizione, ma questo rende necessaria l'identificazione degli argomenti. Esistono più criteri di matching, che vengono applicati secondo una priorità prestabilita. Innanzitutto viene effettuato un matching esatto, ovvero una variabile passata con nome identico a quello di uno degli argomenti formali viene associata a tale argomento. Dopodiché viene effettuato un matching parziale, in base alle porzioni iniziali dei nomi. Infine, le variabili rimanenti vengono associate in base all'ordine con cui si presentano. Se nella definizione della funzione è presente tra gli argomenti formali ..., gli argomenti passati non associabili a nessun argomento formale vengono trasferiti al posto dei ... presenti nella chiamata a una funzione inclusa nel `body`.

- *Creazione di un environment*, quello locale, che contiene gli argomenti passati ed è incluso nell'*enclosing environment* della funzione.

- *Evaluation* del corpo della funzione e return. In `R` esiste un meccanismo detto *lazy evaluation*, che ha lo scopo di evitare valutazioni non necessarie di variabili. Per ogni argomento formale viene creato un oggetto di tipo *promise*, che contiene un'espressione, il riferimento a un environment e una flag `seen`; l'espressione contenuta nella promise viene valutata solo una volta (viene controllata preventivamente la flag seen per sapere se è necessaria la valutazione) nell'environment specificato, e solo in 2 casi: se la variabile diventa argomento di una funzione primitiva o se l'evaluation è necessaria per la selezione di un metodo (si veda il capitolo 2).

Le funzioni di base di R sono implementate invece come *primitive*. Chiamare dalla console di R il nome di una di tali funzioni fornisce per tutte un risultato simile al seguente:

```
> `if`
.Primitive("if")
```

Il corpo della funzione non è accessibile a livello utente, perché esso è codificato direttamente nel sorgente C del programma. Ogni volta che l'evaluator incontra una funzione etichettata come primitiva ne cerca corrispondenza in una tabella, che si può trovare nel file sorgente `names.c`:

```
attribute_hidden FUNTAB R_FunTab[] ={
...
/* Primitives */
{"if",   do_if,   0,200,-1,{PP_IF,   PREC_FN,1}},
{"while",do_while,0,100,-1,{PP_WHILE,PREC_FN,0}},
{"for",  do_for,  0,100,-1,{PP_FOR,  PREC_FN,0}},
...}
```

La prima colonna contiene il nome della funzione R (eventualmente privata dei backticks) da associare a una determinata funzione C, indicata nella seconda colonna, da chiamare con alcuni parametri, definiti nelle colonne successive. Alcune delle funzioni C primitive sono associate in realtà a più primitive R , svolgendo operazioni che vengono distinte, a seconda dei casi, attraverso i parametri aggiuntivi contenuti nella tabella:

```
...
{"+",do_arith,PLUSOP, 1,2,{PP_BINARY,PREC_SUM, 0}},
{"-",do_arith,MINUSOP,1,2,{PP_BINARY,PREC_SUM, 0}},
{"*",do_arith,TIMESOP,1,2,{PP_BINARY,PREC_PROD,0}},
...
```

Per questioni di efficienza, le funzioni primitive si differenziano dalle ordinarie anche per il fatto di eseguire direttamente il matching ordinale degli argomenti, che è naturalmente più rapido. Inoltre tali funzioni si dividono in due categorie: le *builtin* ricevono come argomenti una lista di oggetti, mentre le *special* ricevono in ingresso delle unevaluated expressions. Questa distinzione permette, di nuovo, di non effettuare valutazioni potenzialmente inutili; ad esempio, la funzione `` `+` `` dovrà sicuramente valutare entrambi i suoi argomenti, ed è quindi una funzione builtin, mentre la funzione `` `if` `` riceve in ingresso 3 argomenti, ma tra il secondo e il terzo solo e soltanto uno viene poi effettivamente utilizzato (in base alla condizione logica passata come primo argomento).

Capitolo 2

Programmazione ad oggetti

La programmazione ad oggetti in `R` è resa possibile in modo molto diverso da come lo si intende tipicamente nei linguaggi di programmazione object oriented; le classi sono ad esempio più simili a delle semplici `struct` C che non alle classi C++. Le *classi S3*, che fungono tutt'ora da struttura per buona parte degli output delle funzioni più usate di `R` , consistono semplicemente in liste di oggetti diversi alle quali viene associata un'etichetta; le *classi S4* invece, che mimano le classi introdotte nelle ultime versioni di S, dispongono di una struttura ben definita. Quelli che vengono chiamati *metodi* in `R` sono invece a tutti gli effetti degli overloading di funzioni, che permettono a una stessa funzione di avere comportamenti diversi a seconda della classe dei suoi argomenti.

2.1 Classi S3

Una classe in stile S3 si definisce molto semplicemente come una lista, dopodiché le si applica l'etichetta corrispondente alla classe a cui la si vuole associare con il comando `class`.

```
1 > P1=NULL
2 > P1$x=1
3 > P1$y=1
4 > P2=NULL
5 > P2$r=sqrt(2)
6 > P2$theta=3*pi/4
7 > class(P1)="xypoint"
8 > class(P2)="rthetapoint"
```

Possiamo notare che non è stata dichiarata da nessuna parte la struttura delle due classi create a partire dalle liste `P1` e `P2`: il controllo della presenza

di determinati campi e della validità del loro contenuto è lasciato completamente al programmatore e all'utente. Nell'esempio in alto abbiamo creato due punti in coordinate cartesiane e polari, e vogliamo scrivere una funzione, denominata `xpos`, che restituisca la coordinate cartesiana x del punto a prescindere dal sistema di riferimento con cui l'abbiamo salvato. Innanzitutto creiamo la funzione che effettuerà la scelta del metodo S3 adatto, chiamando al suo interno `UseMethod`:

```
xpos=function(x, ...)
UseMethod("xpos")
```

A questo punto i metodi possono essere definiti semplicemente scrivendo nuove funzioni nella forma `funzione.classe`, specificando un corpo diverso a seconda dei casi:

```
> xpos.xypoint=function(x) x$x
> xpos.rthetapoint=function(x) x$r*cos(x$theta)
> xpos(P1)
[1] 1
> xpos(P2)
[1] -1
```

Possiamo notare che `xpos` esamina la classe dell'argomento passato di volta in volta, e su questa base ha scelto quale metodo utilizzare.

Come semplice meccanismo di polimorfismo è inoltre possibile specificare per un oggetto S3 più classi di appartenenza simultanee:

```
s3class <- NULL
s3class$name <- "nome"
s3class$number <- 0
class(s3class) <- c("Nome","Matricola","Nome_Matricola")
```

2.2 Classi S4

Una classe S4 è una struttura dati contenente diversi attributi (denominati *slot*) di tipo predeterminato. Per definire una classe di questo tipo si ricorre al comando

```
setClass("class_name", representation(x="type1", y="type2
   "), prototype=list(x=valx,y=valy)
```

dove `class_name` è una stringa che definisce il nome della classe, mentre `representation` è una funzione che definisce gli attributi e ne associa un tipo (nel caso dell'esempio sopra, gli attributi sono `x` e `y`); l'argomento opzionale

`prototype` permette invece di stabilire dei valori di default per gli attributi, inserendoli all'interno di una lista.

Per creare un'istanza di una classe definita in precedenza si utilizza la funzione `new`, eventualmente specificando tra i suoi argomenti i valori di inizializzazione per gli attributi. L'accesso agli attributi (tutti pubblici) viene effettuato attraverso l'operatore `@` o con la funzione `slot`:

```
> setClass("punto2D",representation(x="numeric",y="
   numeric"),prototype=list(x=0,y=0))
[1] "punto2D"
> P <- new("punto2D",x=4)
> P@x
[1] 4
> slot(P,"x")
[1] 4
> P@y
[1] 0
> P@y <- "ciao"
Errore in checkSlotAssignment(object, name, value) : ...
```

A differenza delle classi S3, le classi S4 permettono l'assegnamento di valori agli attributi solo per il tipo indicato nella definizione, il quale può consistere in un tipo standard di `R` o in una classe già definita. Nonostante questo controllo, è possibile che siano richieste ulteriori proprietà agli attributi non verificabili col solo controllo del tipo (per esempio, che un vettore di tipo `numeric` abbia una lunghezza predeterminata). Esistono 2 meccanismi per controllare la coerenza degli oggetti creati: l'overloading del costruttore e il controllo di validità.

Anticipando per un attimo l'argomento della Sezione 2.3, vediamo la definizione del metodo `initialize`, ovvero un overloading della funzione omonima, per la classe `punto2D`. Nel corpo del metodo possiamo notare che sono indicate le istruzioni da eseguire prima di effettuare l'inizializzazione standard: in questo caso, se l'utente crea un oggetto `punto2D` senza specificare i valori iniziali di `x` e `y`, (la funzione `missing(par)` restituisce `TRUE` se la funzione in cui è contenuta ha ricevuto come parametro `par`, `FALSE` altrimenti), viene chiamata una versione meno specifica di `initialize`, ovvero quella standard, con parametri inizializzati a 0.

```
setMethod("initialize","punto2D",function(.Object,x,y
   ,...){
   if(missing(x) & missing(y)){
      callNextMethod(.Object,x=0,y=0,...)
   }
}
```

Inoltre può essere definita una funzione di controllo della validità di un oggetto, che oltre ad essere richiamata esplicitamente può essere impostata per controllare la validità prima delle inizializzazioni e delle copie. Ad esempio, volendo creare una classe `curva` contenente due vettori `x` e `y` che definiscano le coordinate della discretizzazione di una curva in $\mathbb{R}^2$, saremo interessati ad assicurarci che i due vettori abbiamo la stessa lunghezza. La funzione `validObject` si può quindi definire come:

```
validObject <- function(object) {
   if(length(object@x) == length(object@y)) TRUE
   else paste("Unequal x,y lengths?)
}
```

Tale funzione può essere impostata come funzione di validazione alla definizione della classe, passando il parametro `validity=validObject` nella chiamata a `setClass`, oppure può essere eseguita in un momento successivo per verificare la validità dei parametri, con la chiamata `setValidity("curva", validObject)`.

2.2.1 Ereditarietà

La programmazione ad oggetti in `R` prevede anche un meccanismo di ereditarietà da altre classi S4 o da tipi standard, mentre non è possibile ereditare da classi S3, le quali non hanno una struttura ben definita e conforme a quella del più recente paradigma di programmazione ad oggetti di S ed `R` .

Una classe può essere definita come "figlia" con due meccanismi. Il più immediato è l'uso dell'argomento opzionale `contains` di `setClass`:

```
setClass("punto3D",representation(z="numeric"),contains="
   punto2D")
```

Un oggetto `punto3D` avrà ora 3 attributi, di cui due, `x` e `y`, ereditati dalla classe `punto2D`; inoltre a un attributo definito come di classe `punto2D` potrà essere assegnato anche un oggetto di classe `punto3D`. L'argomento `contains` può anche essere una lista, in modo tale che la nuova classe erediti da più "genitori". Il secondo meccanismo è definire una *class union*:

```
setClassUnion("unione",c("classe1","classe2"))
```

In questo modo viene creata la classe `unione`, la quale non ha attributi ma risulta essere madre di `classe1` e `classe2`. La classe `unione` potrà quindi essere specificata come tipo di un attributo, in modo da permettere che siano conformi a quell'attributo oggetti di una qualsiasi sua classe figlia; notiamo che il meccanismo di ereditarietà in questo caso è invertito: si specifica quali

classi possono ereditare dalla class union, anziché dire di ogni classe figlia che la class union è loro madre.

Si possono infine definire classi *virtuali*, che esattamente come in C++ non possono venire istanziate pur essendo possibile ereditare da esse; il loro scopo è quello di fornire uno scheletro comune valido per molte classi che però non hanno alcuna funzione se non vengono specializzate.

```
setClass("virtuale",representation(x="numeric"),
    contains="VIRTUAL")
```

Come nel caso della class union, una classe virtuale può essere specificata come tipo di un attributo.

2.3 Metodi S4

Un metodo in stile S4 viene creato attraverso la seguente definizione:

```
setMethod("function_name",signature(x="type1",y="type2"),
    definition)
```

dove `function_name` è il nome della funzione che si vuole creare o di cui si vuole effettuare l'overloading, `signature` definisce le classi degli argomenti che provocano la chiamata di questa versione della funzione, mentre al posto di `definition` va inserito il corpo della funzione da eseguire se gli argomenti passati corrispondono alle classi indicate nella signature. Quando viene chiamata, `setMethod` inserisce il metodo in una tabella di funzioni, pronto per essere selezionato nel caso in cui i tipi degli argomenti passati coincidano con i tipi previsti dalla definizione del metodo.

Vediamo un esempio in cui si vuole creare nuovi metodi `plot` e `[[` per la classe `cerchio`, costituita da una struttura di due attributi numerici rappresentanti centro e raggio.

```
setClass("cerchio",representation(R="numeric",C="numeric"
    ))
setMethod("plot",signature(x="cerchio", y='missing'),
    function(x,y,...){
       r <-slot(x,"R")
       cent <- slot(x,"C")
       t <- seq(0,2*pi,by=0.01)
       plot(cent[1]+r*cos(t),cent[2]+r*sin(t))
    }
)

setMethod("[[",signature(x = "cerchio",i="character",j="
    missing"),
```

```
12     function (x, i, j, ...){
13        return(slot(x,i))
14     }
15 )
16
17 oggetto_cerchio <- new("cerchio")
```

Il metodo `plot` per `cerchio` permette di disegnare il grafico della curva in modo molto semplice e intuitivo, richiamando `plot(oggetto_cerchio)`; siccome la versione standard della funzione `plot` prevede due argomenti obbligatori, il secondo non può essere ignorato, e deve essere quindi impostato come `'missing'`. Il metodo per l'operatore `[[` si può scrivere in modo analogo, dato che, come abbiamo visto nella sezione 1.4, anche gli operatori sono in realtà funzioni (primitive); di nuovo, la funzione standard richiede tre argomenti obbligatori, ma l'uso che vogliamo fare dell'operatore è del tipo `oggetto_cerchio["C"]` per richiamare l'attributo `C`, quindi `j` viene viene impostato come mancante.

Rispetto all'ereditarietà, il comportamento dei metodi è quello naturale: un metodo che si applica ad argomenti di una certa classe si applica anche agli oggetti di classi che ereditano da essa.

2.3.1 Funzioni generiche

Il *method dispatch*, ovvero la selezione del metodo opportuno per gli argomenti passati, viene effettuato dalla versione *generica* della funzione per cui è stato scritto il metodo. Le funzioni generiche sono visualizzabili invocando la funzione `getGenerics`, e alcune di esse (ad esempio gli operatori logici e di confronto e molte funzioni matematiche di base) sono state create come tali nativamente; quando si scrive un metodo per una funzione che non è generica, `R` crea una generica implicita, che effettua solamente la selezione del metodo, ma è possibile definire una generica personalizzata utilizzando `setGeneric`. Ad esempio:

```
1 setGeneric("plot", useAsDefault=function(x,y,...)
     graphics::plot(x, y, ...))
```

Il codice riportato sopra in realtà corrisponde esattamente a ciò che esegue `R` quando crea la generica implicita.

Le funzioni generiche possono essere richiamate esplicitamente nel corpo dei metodi scritti per esse, chiamando `callGeneric("nome_funzione")`, oppure si può forzare la scelta di un altro metodo rispetto a quello corrente, utilizzando `callNextMethod`. Nel primo caso, l'uso tipico consiste nel processare gli argomenti all'inizio del corpo del metodo, e successivamente effettuare

un method dispatch con nuovi argomenti; nel secondo caso accade la stessa cosa, ma il metodo attuale viene rimosso dalla lista di metodi selezionabili dalla generica per il method dispatch.

Un'altra tecnica utile è la definizione di gruppi di funzioni generiche; la funzione `Arith`, ad esempio, è una *group generic*, a cui fanno capo gli operatori matematici binari:

```
> getGroupMembers(Arith)
[1] "+"   "-"   "*"   "^"   "%%"  "%/%" "/"
```

La funzione `Arith` non viene mai chiamata direttamente, ma estenderla implica l'estensione di tutte le funzioni appartenenti al gruppo. Ad esempio un'estensione di `Arith` per la classe `cerchio` potrebbe essere la seguente:

```
> setMethod("Arith",c("cerchio","cerchio"),
+   function(e1, e2) {
+     d=new("cerchio")
+     d@R=callGeneric(e1@R,e2@R)
+     d@C=callGeneric(e1@C,e2@C)
+     return(d)
+  }
+ )
[1] "Arith"
> C1=new("cerchio")
> C1@R=3
> C1@C=c(1,2)
> C2=new("cerchio")
> C2@R=2
> C2@C=c(1,0)
> C1+C2
An object of class "cerchio"
Slot "R":
[1] 5

Slot "C":
[1] 2 2

> C1*C2
An object of class "cerchio"
Slot "R":
[1] 6

Slot "C":
[1] 1 0
```

Poiché la funzione `Arith` prevede i due argomenti `e1` ed `e2`, che rappresentano gli argomenti di un qualsiasi operatore binario del suo gruppo, l'overloading di questa funzione accoppiata a un uso ragionevole di `callGeneric` (in questo caso sui singoli attributi dei due oggetti `cerchio` ricevuti in ingresso) permette di generalizzare in una sola volta tutte le operazioni binarie alla classe `cerchio`.

I gruppi possono anche essere creati annidati fra loro, ma occorre fare attenzione alla gerarchia di selezione. Il method dispatch effettuato dalla funzione generica sceglie, con il seguente ordine di priorità:

1. un metodo con firma compatibile
2. un metodo compatibile per la group generic, se presente
3. un metodo compatibile per la group generic della group generic e così via, se presenti
4. una metodo compatibile ereditato da una superclasse degli argomenti. La distanza da tali metodi e' data dalla somma su tutti gli argomenti della lontananza della superclasse (distanza 1 per la madre, 2 per la "nonna", ecc.)

2.4 Un trucco: programmazione ad oggetti con le closures

Come abbiamo visto, la programmazione ad oggetti in `R` comporta notevoli differenze rispetto a quella che si può effettuare ad esempio nel C++; in particolare, non esistono attributi privati, cosa che rende le classi poco più che semplici `struct`, inoltre i metodi sono sostanzialmente degli overloading, quindi anch'essi possono essere solo pubblici. In [5] è stata proposta una tecnica che è in grado di ovviare in parte a questi problemi e che si può applicare senza l'uso di pacchetti aggiuntivi; essa consiste in un uso intelligente delle proprietà delle *closures*, ovvero semplicemente le funzioni di `R` , e dell'operatore di assegnamento non locale `<<-`. Tale operatore cerca la variabile da assegnare nell'environment in cui viene valutato l'assegnamento, e solo successivamente negli environment esterni; se non trova la variabile, la crea nell'environment globale. Vediamo un esempio in cui viene creata la "classe" `vector`, la quale contiene gli attributi `V`, un vettore, ed `M`, la sua media campionaria.

```
1 vector<-function(v){
2    mean_vector<- function(){
```

```
3         M <<-mean(V)
4     }
5     V<-v
6     M<-NULL
7     mean_vector()
8     printM<- function(){
9         return(M)
10    }
11    printV<- function(){
12        return(V)
13    }
14    modifica<-function(v){
15        V<<-v
16        mean_vector()
17    }
18    element<-function(i){
19        printV()[i]
20    }
21    return(list(modifica=modifica,element=element,printM=
          printM,esterna=esterna))
22 }
```

La funzione `vector` rappresenta la classe che vogliamo definire, e tutto ciò che vogliamo sia pubblico viene restituito dalla funzione; questo meccanismo funziona poiché, nel caso particolare in cui una funzione restituisca almeno una sua funzione interna, l'environment interno non viene eliminato come di solito accade, quindi sia le variabili che le funzioni definite vengono conservate. Nell'esempio abbiamo reso le variabili interne `V` e `M`, che svolgono il ruolo di attributi, non accessibili dall'esterno, ma solo attraverso appositi funzioni interne, ovvero i metodi della nostra classe. Tutti i metodi sono stati restituiti dalla funzione/classe `vector` tranne `printV`, che non è utilizzabile dall'esterno ma viene sfruttato dal metodo pubblico `element`. A questo punto si potrà accedere ad attributi e metodi con l'operatore di lista `$`, utilizzando in modo del tutto analogo all'operatore `.` in C++:

```
1 > ogg <- vector(c(1,2,3))
2 > ogg$printV()
3 Errore: tentativo di applicare una non-funzione
4 > ogg$element(3)
5 [1] 3
6 > ogg$modifica(c(4,5,6))
7 > ogg$printM()
8 [1] 5
```

In particolare si può osservare che l'utilizzo delle funzioni interne permette di modificare gli attributi della classe, cosa che non può avvenire utilizzando i metodi classici in quanto è contro la filosofia di `R` permettere che una funzione possa modificare al suo interno un oggetto definito all'esterno di essa.

Capitolo 3

Interazioni con altri linguaggi

Con l'eventuale ausilio di alcune sue estensioni, `R` offre la possibilità di interagire con vari altri programmi e linguaggi di programmazione. Un esempio molto semplice, per interagire con la shell UNIX, è la funzione `system`:

```
1 > system("a='uso la shell!'; echo $a")
2 uso la shell!
```

Al di là delle possibilità offerte dai pacchetti aggiuntivi, `R` offre in modo naturale delle interfacce a funzioni scritte in C o Fortran, attraverso le funzioni `.C`, `.Fortran`, `.Call` e `.External`. L'uso di tali funzioni è esposto nel paragrafo 3.1, mentre il paragrafo 3.2 fornisce alcuni cenni all'uso di R all'interno di programmi C.

3.1 Richiamare funzioni in C/Fortran

Le 4 funzioni primitive che permettono di interagire con codici C e Fortran (e C++) offrono un'interfaccia superficialmente piuttosto simile, ma si distinguono per alcuni aspetti fondamentali. In generale, ricevono come primo argomento una stringa che denota il nome di una funzione C o Fortran, e come argomenti opzionali (denotati da ...) tutti gli argomenti che si vogliono passare a tale funzione. La funzione individuata dal primo argomento deve essere stata preventivamente caricata da uno shared object, che si può creare in modo molto semplice da terminale/prompt dei comandi con il comando

```
$> R CMD SHLIB mylib.c
```

che crea la libreria dinamica mylib.so. Se tale libreria fa parte di un pacchetto, essa viene caricata automaticamente al caricamento dello stesso, mentre nel caso in cui la si voglia caricare manualmente si ricorre alla funzione `dyn.load`.

```
> dyn.load("mylib.so")
> myfun(arg1,arg2,...){
    ...
    .C("myfunc",arg1,arg2,...)
    ...
}
> dyn.unload("mylib.so")
```

3.1.1 .C e .Fortran

Le funzioni primitive `.C` e `.Fortran` forniscono interfacce a codici C/Fortran in modo tale che all'interno di tali codici non sia necessario manipolare oggetti `SEXPREC`. Questo viene fatto convertendo gli argomenti di `.C` o `.Fortran` dal secondo in avanti in variabili di tipi C/Fortran, secondo le corrispondenze indicate in Tabella 3.1.

Tipo R	**Tipo C**	**Tipo Fortran**
`logical`	`int *`	`INTEGER`
`integer`	`int *`	`INTEGER`
`double`	`double *`	`DOUBLE PRECISION`
`complex`	`Rcomplex *`	`DOUBLE COMPLEX`
`character`	`char **`	`CHARACTER*255`
`raw`	`char *`	nessuno

Tabella 3.1: Conversioni effettuate da `.C` e `.Fortran`

Gli argomenti che si possono passare a `.C` e `.Fortran` sono quindi limitati ai soli tipi previsti nella prima colonna della Tabella 3.1.

Vediamo più nel dettaglio il funzionamento di queste funzioni attraverso un esempio, che presenta tre implementazioni per `R` del crivello di Eratostene: la prima scritta interamente in codice `R`, la seconda implementata attraverso una funzione C incapsulata da un wrapper `R` e la terza implementata in modo analogo ma con una funzione Fortran. La versione `R` dell'algoritmo cui faremo riferimento è la seguente.

```
#Crivello di Eratostene. Riceve un intero n come
   argomento e trova tutti i numeri primi fino a n.
sieve <- function(n){
   s <- seq(2,n)
   if(n<4){
   } else {
```

```
6         for(i in 2:(floor(sqrt(n)))){
7            if(s[i-1]!=0){
8               times <- 2
9               while(i*times <= n){
10                 s[(i*times)-1] <- 0
11                 times <- times+1
12              }
13           }
14        }
15    }
16    return(s[which(s!=0)])
17 }
```

L'algoritmo è computazionalmente molto oneroso, quindi può risultare conveniente farne eseguire la parte più pesante da un codice scritto in un linguaggio più efficiente. A seconda che si voglia utilizzare una funzione C o Fortran, si può scrivere uno dei due wrapper seguenti:

```
1 dyn.load("sieve_c.so")
2 sieve_c <- function(n){
3    res=seq(2,n)
4    res=.C("sieve_c",as.integer(n),as.integer(res))[[2]]
5    return(res[which(res!=0)])
6 }
7 dyn.unload("sieve_c.so")
```

```
1 dyn.load("sieve_f.so")
2 sieve_f <- function(n){
3    res=seq(2,n)
4    res=.Fortran("sieve_c",as.integer(n),as.integer(res))
        [[2]]
5    return(res[which(res!=0)])
6 }
7 dyn.unload("sieve_f.so")
```

Notiamo che per sicurezza gli argomenti sono stati passati forzandoli a un tipo convertibile dalle due funzioni di interfaccia. `.C` e `.Fortran` restituiscono in una lista i valori aggiornati degli argomenti (dal secondo in avanti) che sono stati loro passati, dei quali in questo caso ci interessava il secondo, estratto dalla lista attraverso l'operatore `[[`. La variabile `res`, in origine il vettore degli interi da 2 a `n`, è stata modificata in modo da avere degli zeri in corrispondenza dei numeri non primi; il return del wrapper contiene infine una selezione dei soli elementi di `res` diversi da 0.

Le funzioni `sieve_c` e `sieve_f` sono presentate di seguito.

```c
#include<Rmath.h>
//s è la lista dei numeri da 2 a n
void sieve_c(int* n, int* s){
   int times;
   if(*n<4){} else {
      for(int i=2; i<=floor(sqrt(*n)); ++i){
         if(s[i-2]!=0){
            times=2;
            while(i*times <= *n){
               s[(i*times)-2]=0;
               times++;
            }
         }
      }
   }
}
```

```
C S è la lista dei numeri da 2 a N
      SUBROUTINE SIEVE_F(N, S)
      INTEGER N, S(N), TIMES, I
      IF (N.LT.4) THEN
                        GOTO 20
      ELSE
           DO 20, I=2,FLOOR(SQRT(FLOAT(N)))
                 IF (S(I-1).NE.0) THEN
                          TIMES=2
10                        IF (I*TIMES.LE.N) THEN
                              S(I*TIMES-1)=0
                              TIMES=TIMES+1
                              GOTO 10
                          ELSE
                ENDIF
                 ELSE
                 ENDIF
20         CONTINUE
      ENDIF
      END
```

Come possiamo notare, le due funzioni sono state scritte senza tener conto del fatto che sarebbero state successivamente utilizzate da `R`. Nella versione C è stato importato l'header file `Rmath.h` per sfruttare le funzioni matematiche definite nel sorgente di `R` (nello specifico la radice quadrata), ma ciò non è obbligatorio dato che tutte le variabili in gioco sono di tipi standard C.

I vantaggi nell'uso di tali funzioni stanno nella semplicità di interazione con R e nella possibilità di utilizzare codici già scritti per altri scopi, adottando accorgimenti minimi. D'altra parte in R è possibile maneggiare oggetti molto complessi e scomporre tali oggetti in componenti più semplici, come stringhe, tipi numerici e variabili logiche, può risultare molto scomodo: basti pensare a un `dataframe` che può contenere in ogni colonna variabili di tipo factor, character, integer, double, Date o di altri tipi non standard, che sarebbe molto difficile ridurre a vettori di stringhe e numeri.

3.1.2 .Call e .External

Le funzioni `.Call` e `.External` sono in grado di ricevere in ingresso oggetti R di qualsiasi tipo, grazie all'omogeneità di base degli oggetti `SEXPREC`; tuttavia questo avviene al prezzo di una complessità molto maggiore del codice C che deve maneggiare tali oggetti. All'interno del sorgente di R sono infatti definite varie macro che permettono di operare direttamente sugli oggetti `SEXPREC`, ma poiché tale struttura è stata progettata in modo da risultare "opaca" all'utente, maneggiarla implica alcune difficoltà, legate sia all'estrazione delle informazioni sia alla gestione della memoria.

Vediamo un esempio di uso di `.Call` per implementare, ancora una volta, il crivello di Eratostene. Il wrapper R in questo caso diventa

```
dyn.load("sieve_call.so")
sieve_call <- function(n){
   result <- .Call("sieve_call",n)
   return(result[which(result!=0)])
}
dyn.unload("sieve_call.so")
```

mentre la funzione `sieve_call` ha la seguente definizione:

```
#include "Rdefines.h"
SEXP sieve_call(SEXP n){
   int times;
   int nn=INTEGER_VALUE(n);
   result=NULL;
   PROTECT(result=allocVector(INTSXP,nn-1));
   for(int i=2; i<nn; ++i){
      INTEGER(result)[i-2]=i;
   }
   if(nn<4){} else {
      for(int i=2; i<=floor(sqrt(nn)); ++i){
         if(INTEGER(result)[i-2]!=0){
            times=2;
```

```
14              while(i*times <= nn){
15                  INTEGER(result)[(i*times)-2]=0;
16                  times++;
17              }
18          }
19      }
20  }
21  UNPROTECT(1);
22  return result;
23 }
```

Il file `Rdefines.h` è stato incluso per poter avere accesso alla definizione di `SEXP` e alle macro che devono essere utilizzate all'interno del codice, in questo caso `INTEGER_VALUE`, `PROTECT`, `INTSXP` e `INTEGER`. La funzione C `sieve_call` riceve in ingresso dei `SEXP` e restituisce un `SEXP`, che eventualmente è una lista contenente altri oggetti analoghi. Per poter estrarre l'intero contenuto nel `SEXP n` in forma di `int` è necessario ricorrere alla macro `INTEGER_VALUE`. Non scendiamo nel dettaglio riguardo all'uso delle altre macro utilizzate per lavorare su un `SEXP` che fa riferimento a numeri interi, ma particolare attenzione va rivolta a `PROTECT`. Alla riga 6 vogliamo allocare un `SEXPREC` denominato `result`, che non è però referenziato da nessuna variabile della sessione attuale di `R`: questo significa che di norma il garbage collector potrebbe eliminare `result` da un momento all'altro. La macro `PROTECT` permette di indicare al garbage collector che la variabile `return` non va cancellata, mentre con `UNPROTECT(k)` si tolgono dalla pila delle variabili protette le `k` che stanno in cima.

Come abbiamo visto, affrontando alcune complicazioni a livello di programmazione è possibile lavorare sugli oggetti `SEXP` (o meglio sui `SEXPREC` da essi puntati) direttamente nel codice C. La funzione `.External` funziona in modo analogo, ma mentre `.Call` può ricevere in ingresso solo un massimo di 65 argomenti, `.External` può ricevere in ingresso un numero arbitrario di argomenti, che vengono poi passati alla funzione C chiamata all'interno di un `SEXPREC` di tipo lista.

3.2 Eseguire R da un programma C

Data la vasta disponibilità di algoritmi, in particolare per metodi statistici, implementati per `R` , in casi particolari può essere utile eseguire direttamente funzioni o parti di codice scritte in questo linguaggio, richiamando temporaneamente `R` da un programma C. Vediamo come ciò possa essere fatto osservando un esempio in cui si vuole lanciare lo script `R` esempi_locpoly.r

dal `main` del programma C *myCprogram*, e inserire successivamente il risultato nel `double ser`.

```
//file "main.c"

#include <Rinternals.h>
#include <R_ext/Parse.h>
#include <Rdefines.h>
#include <stdio.h>
```

A parte `stdio.h`, che serve a gestire input e output, gli header files caricati definiscono strutture e funzioni interne di R. `Rinternals.h` contiene le definizioni delle strutture, come ad esempio `SEXP`; `R_ext/Parse.h` contiene la dichiarazione della funzione `R_ParseVector`, che serve ad effettuare il parsing della stringa di codice R importata; `Rdefines.h` definisce invece varie macro che permettono di operare su `SEXP`.

```
void init_R() {
   extern Rf_initEmbeddedR(int argc, char **argv);
   int argc = 2;
   char *argv[] = {"myCprogram","--silent"};

   Rf_initEmbeddedR(argc, argv);
}
```

`init_R` è un semplice wrapper alla funzione `Rf_initEmbeddedR`. Permette di passare come argomenti il nome del programma che lancia R, in modo da distinguerne istanze diverse, ed eventuali parametri scelti tra quelli disponibili lanciando `R` da riga di comando.

```
void readR(char filename[20],char * cmd){
   char temp[100];
   FILE* input;
   input=fopen(filename,"r");
   while(!feof(input)){
      fscanf(input,"%s",temp);
      if(temp[0]!='#'){
         strcat(cmd,temp);
         strcat(cmd,"; ");
      }
   }
   fclose(input);
}
```

La funzione `readR` legge semplicemente il file di testo contenente il codice `R` da eseguire all'interno del programma.

```
SEXP RinC(char filename[20]){
   init_R();

   char cmd[10000]="";
   readR(filename,cmd);

   SEXP cmdSexp, cmdexpr, ans = R_NilValue;

   PROTECT(cmdSexp = allocVector(STRSXP, 1));

   ParseStatus status;
   SET_STRING_ELT(cmdSexp, 0, mkChar(cmd));
   cmdexpr = PROTECT(R_ParseVector(cmdSexp, -1, &status,
      R_NilValue));
   if (status != PARSE_OK) {
      UNPROTECT(2);
      error("invalid call %s", cmd);
   }
   int i;
   for(i = 0; i < length(cmdexpr); i++){
      ans = eval(VECTOR_ELT(cmdexpr, i), R_GlobalEnv);
   }
   UNPROTECT(2);
   Rf_endEmbeddedR(0);

   return ans;
}
```

All'interno di `RinC` viene innanzitutto inizializzato `R`, dopodiché viene letto il file `filename` contenente il codice da processare e il suo contenuto è inserito nel `SEXP` di tipo stringa `cmdSexp`; ciò avviene preallocando `cmdSexp` attraverso la funzione `allocVector` e inserendovi il vettore di caratteri `cmd` per mezzo della macro `SET_STRING_ELT`. Ricordiamo che, essendo `cmdSexp` una variabile `SEXP`, deve essere protetta dal garbage collector attraverso l'apposita macro `PROTECT`. Il parsing viene effettuato attraverso la funzione `R_ParseVector`, e l'unevaluated expression risultante `cmdexpr`, se sintatticamente corretta, viene valutata in ogni sua parte all'interno di un ciclo, per mezzo di `eval`. Infine deve essere chiusa la versione *embedded* di `R`.

```
int main(){
   SEXP res;
   res=RinC("esempi_locpoly.r");
   double ser=*REAL(res);
   printf("%f",ser);
```

```
6     printf("%s","\n");
7     return 0;
8  }
```

All'interno del `main` troviamo il `SEXP res` che ospita il risultato ottenuto dall'esecuzione dello script `esempi_locpoly.r`. Le righe che seguono l'assegnamento di `res` sono semplicemente strumentali alla visualizzazione del risultato, che richiede l'estrazione del valore numerico ottenuto attraverso la macro `REAL`.

Per compilare ed eseguire questo programma è opportuno definire preliminarmente, da prompt dei comandi o shell (come nell'esempio), la variabile d'ambiente

```
$> R_HOME_DIR="/Library/Frameworks/R.framework/Resources"
$> export R_HOME_DIR
```

Il file main.c va naturalmente compilato linkando le opportune librerie di R, nel modo seguente:

```
$> cc -I$R_HOME_DIR/include/ -L$R_HOME_DIR/lib/ -lR main.
   c -o myCprogram.out
```

Al momento dell'esecuzione la variabile `R_HOME_DIR` diventa essenziale, in quanto l'attuale processo deve conoscere la home directory di R per poterlo lanciare quando richiesto.

```
$> ./myCprogram.out
KernSmooth 2.23 loaded
Copyright M. P. Wand 1997-2009
-5.312348
```

Capitolo 4

Calcolo parallelo in R

L'utilizzo di tecniche di calcolo parallelo può essere molto utile anche in ambito statistico. Per questo motivo sono stati creati alcuni pacchetti per il software `R` che permettono lo scambio di informazioni tra sottoprocessi. Tra i pacchetti più importanti ricordiamo `Rmpi` e `snow`. L'obiettivo di questo capitolo è quello di presentare i comandi principali di questi pacchetti e mostrare l'implementazione in `R` di un algoritmo in parallelo. Sono stati inoltre effettuati dei casi test per verificare l'efficienza di diversi metodi.

4.1 `Rmpi`

Il principale pacchetto `R` che permette di sfruttare le potenzialità del calcolo parallelo è `Rmpi` [7]. Tale pacchetto è basato su MPI (Message Passing Interface) e permette comunicazioni di tipo *master-slave* tra diversi sottoprocessi all'interno di un sistema a memoria distribuita. Il pacchetto `Rmpi` è in realtà un vero e proprio *wrapper* che permette di richiamare in `R` funzioni C che implementano il protocollo MPI.

4.1.1 Principali funzioni di `Rmpi`

La prima operazione dopo aver caricato la libreria `Rmpi` tramite il comando `library(Rmpi)`, è la creazione di un determinato numero di sottoprocessi. La creazione di N sottoprocessi avviene per mezzo del comando

```
1 mpi.spawn.Rslaves(nslaves=N).
```

Il comando `mpi.spawn.Rslaves` ha in realtà molti più argomenti:

```
1 mpi.spawn.Rslaves(Rscript=system.file("slavedaemon.R",
    package="Rmpi"), nslaves=mpi.universe.size(), root=0,
    intercomm=2, comm=1)
```

L'argomento `Rscript` permette di lanciare uno script `R` in batch; l'argomento di default permette di lanciare lo script `slavedaemon.R` che crea in modo interattivo gli *environment* `R` per i sottoprocessi. L'argomento di default per `nslaves` è la funzione `mpi.universe.size()` che ritorna in numero di cpu all'interno del cluster. Gli altri argomenti permettono rispettivamente di specificare quale sottoprocesso sarà il *master* e di definire un *intercommunicator number*, che permette le comunicazioni tra *master* e *slave*, e un *communicator number*.
Una volta terminata l'esecuzione del codice è necessario chiudere i sottoprocessi aperti tramite il comando

```
1 mpi.close.Rslaves(comm=1)
```

L'argomento `comm` permette di specificare quali sottoprocessi chiudere; in particolare verranno chiusi tutti i sottoprocessi appartenenti al *communicator number* `comm`.

4.1.2 Invio dati

L'invio dei dati ai sottoprocessi avviene per mezzo delle funzioni `mpi.send` e `mpi.bcast`. La funzione `mpi.send` permette l'invio di dati *point-to-point*, ovvero ad un solo sottoprocesso. Tale funzione è un comando *blocking*, ovvero non permette di eseguire le righe successive di codice finché non è stato completato; di tale funzione esiste anche una versione *non-blocking* che permette di continuare l'esecuzione del codice mentre vengono inviati i dati. Solitamente nel caso di invio di dati è però preferibile usare la versione *blocking*, tranne in casi specifici.
La funzione `mpi.send` permette di inviare dati semplici di tipo intero, double o carattere. Gli argomenti della funzione sono:

```
1 mpi.send(x, type, dest, tag, comm = 1)
```

in cui `x` rappresenta il dato inviato, `type` il tipo di dato inviato, `dest` il sottoprocesso destinatario, `tag` un etichetta collegata al dato e `comm` il *communicator number*.
L'invio di un singolo dato semplice può essere molto limitante nella pratica; per ovviare questo problema è stata definita la funzione `mpi.send.Robj` che è un'estensione della funzione `mpi.send`. La funzione `mpi.send.Robj`, di cui esiste anche una versione *non-blocking* `mpi.isend.Robj`, permette di inviare un qualsiasi oggetto creato in `R` ad un sottoprocesso, per esempio anche una funzione. Come `mpi.send` permette l'invio di dati *point-to-point*. Gli argomenti di `mpi.send.Robj`

```
1 mpi.send.Robj(obj, dest, tag, comm = 1)
```

sono analoghi a quelli della funzione `mpi.send`.

La funzione `mpi.bcast` permette di inviare un dato, al contrario di `mpi.send`, a tutti i sottoprocessi appartenenti ad un *communicator* (comunicazione *one-to-all*). Anche questa funzione è *blocking* e permette di inviare solo dati in formato intero, double o carattere. Gli argomenti della funzione sono

```
1 mpi.bcast(x, type, rank = 0, comm = 1)
```

in cui `x` rappresenta il dato inviato, `type` il tipo, `rank` il numero del sottoprocesso che invia i dati, `comm` il *communicator* a cui inviare il dato.
Per inviare un qualsiasi oggetto creato in `R` si può usare l'estensione `mpi.bcast.Robj`. Tale funzione ha argomenti analoghi a quelli definiti per `mpi.bcast`:

```
1 mpi.bcast.Robj(obj, rank = 0, comm = 1)
```

Un'altra funzione che implementa una comunicazioni *one-to-all* è `mpi.scatter`. `mpi.scatter` divide i dati in parti uguali tra i sottoprocessi appartenenti allo stesso *communicator* e ne invia una parte ad ogni sottoprocesso in ordine di *rank*. Gli argomenti, analoghi a quelli delle funzioni descritte precedentemente, sono:

```
1 mpi.scatter(x, type, rdata, root = 0, comm = 1)
```

Come per le funzioni descritte in precedenza esiste un'estensione della funzione `mpi.scatter`, chiamata `mpi.scatter.Robj`, che permette di dividere in parti uguali tra i diversi sottoprocessi strutture contenenti dati più complessi creati in `R` . Nel caso in cui si voglia inviare i dati dal *master* a tutti gli *slaves* è possibile usare la funzione `mpi.scatter.Robj2slave` in cui non deve essere specificato da quale processo inviare i dati.
Per dividere in parti non uguali i dati tra i sottoprocessi si può invece usare la funzione `mpi.scatterv` che permette di specificare oltre ai parametri di `mpi.scatter`, `scounts` che rappresenta un vettore di interi contenente la dimensione dei blocchi da inviare ad ogni sottoprocesso.

4.1.3 Esecuzione funzioni

Per far eseguire una funzione a tutti i sottoprocessi si possono usare tre funzioni: `mpi.bcast.cmd`, `mpi.remote.exec` e `mpi.apply`.
La funzione `mpi.bcast.cmd` è un'estensione di `mpi.bcast` che permette di far eseguire a tutti i sottoprocessi appartenenti allo stesso *communicator* un comando. Per esempio

```
1 mpi.bcast.cmd(rank <-mpi.comm.rank(), comm = 1)
```

permette di eseguire all'interno di ogni sottoprocesso l'istruzione

```
rank <-mpi.comm.rank()
```

che salva nella variabile locale `rank` il *rank* del sottoprocesso che sta eseguendo il comando. In alternativa `mpi.bcast.cmd` permette di far eseguire a tutti i sottoprocessi una funzione già inviata ad ogni sottoprocesso come oggetto `R` :

```
fun_slave<-function(){
    ...
}
mpi.bcast.Robj2slave(fun_slave)
mpi.bcast.cmd(fun_slave())
```

In questo esempio la funzione `fun_slave` viene inviata a tutti i sottoprocessi per mezzo di `mpi.bcast.Robj2slave` e viene poi eseguita in tutti i sottoprocessi per mezzo di `mpi.bcast.cmd`. La funzione `mpi.bcast.cmd` permette però solo l'esecuzione della funzione e non permette di ricevere l'output della funzione eseguita dai singoli sottoprocessi.

La funzione `mpi.remote.exec` permette anch'essa di far eseguire una funzione da ogni sottoprocesso con gli stessi argomenti. Questa funzione, contrariamente a `mpi.bcast.cmd` permette di passare degli argomenti alla funzione da eseguire in ogni sottoprocesso e permette di salvare in una lista i risultati ottenuti dall'esecuzione della funzione nei diversi sottoprocessi. Gli argomenti di `mpi.remote.exec` sono:

```
mpi.remote.exec(cmd, ..., comm = 1, ret = TRUE)
```

`cmd` è la funzione che verrà eseguita dai singoli sottoprocessi, `comm` il *communicator* a cui viene inviata la funzione e `ret` specifica se la funzione ha valori di ritorno. Gli eventuali argomenti della funzione eseguita dai singoli sottoprocessi vengono passati al posto di `...`:

```
fun_slave<-function(a,b){
    ...
}
mpi.bcast.Robj2slave(fun_slave)
mpi.remote.exec(fun_slave(),a=a0,b=b0)
```

In questo esempio al posto di `...` vengono specificati gli argomenti `a` e `b` della funzione `fun_slave` che era stata precedentemente inviata a tutti i sottoprocessi per mezzo di `mpi.bcast.Robj2slave`.

L'ultima funzione che permette di eseguire una funzione all'interno dei sottoprocessi appartenenti ad un *communicator* è `mpi.apply`. Contrariamente alle due funzioni appena presentate `mpi.apply` permette di eseguire la funzione specificata con argomenti diversi per ogni sottoprocesso:

```
mpi.apply(x, fun, ..., comm=1)
```

Gli argomenti della funzione `fun` sono specificati all'interno dell'*array* `x`, in modo che `x[[i]]` contenga gli argomenti da inviare all'i-esimo sottoprocesso. Eventuali argomenti opzionali della funzione `fun` posso essere specificati al posto di `...` in modo analogo a quanto descritto per la funzione `mpi.remote.exec`. La funzione `mpi.apply` ha anche una versione *non-blocking* `mpi.iapply`. Nel pacchetto `Rmpi` è presente un'ulteriore estensione alla funzione `mpi.apply` chiamata `mpi.applyLB` al cui interno è implementata una versione *load balancing* di `mpi.apply` in grado di bilanciare il carico sui diversi sottoprocessi.

4.1.4 Ricezione dati

La ricezione di messaggi può essere di due tipi: *one-to-one* o *all-to-one*. Mentre la prima consiste nella ricezione da parte di un sottoprocesso dei dati relativi ad un solo sottoprocesso, la seconda permette ad un sottoprocesso di ricevere i dati inviati da tutti gli altri sottoprocessi.
La funzione `mpi.recv` permette la ricezione di dati semplici come interi, double o caratteri con un meccanismo *point-to-point*. Gli argomenti della funzione `mpi.recv` sono:

```
mpi.recv(x, type, source, tag, comm = 1)
```

in cui `x` è il dato di tipo `type` inviato dal sottoprocesso `source` con etichetta `tag`. La funzione `mpi.recv` è una funzione *blocking*; la funzione equivalente a `mpi.recv` *non-blocking* è `mpi.irecv`. Tale funzione ha argomenti analoghi a quelli della funzione `mpi.recv`.
Come nel caso di `mpi.send` esiste un'estensione della funzione `mpi.recv`, chiamata `mpi.recv.Robj`, che permette di ricevere oggetti più complessi di semplici interi, double o caratteri, definiti in `R` . Gli argomenti di tale funzione sono:

```
mpi.recv.Robj(source, tag, comm = 1, status = 0)
```

La versione *non-blocking* di `mpi.recv.Robj` è `mpi.irecv.Robj`.

Per effettuare una comunicazione *all-to-one* si può utilizzare la funzione `mpi.gather`. Gli argomenti della funzione sono:

```
mpi.gather(x, type, root = 0, comm = 1)
```

in cui `x` è il dato di tipo `type` inviato da tutti i sottoprocessi al sottoprocesso `root`. I dati vengono poi raccolti dal processo `root` in ordine di *rank*. Una variante di `mpi.gather` è

```
mpi.gatherv(x, type, rdata, rcounts, root = 0, comm = 1)
```

che permette di specificare in `rcounts` la lunghezza del messaggio ricevuto da ogni sottoprocesso. La funzione `mpi.gather` permette, come `mpi.scatter`, l'invio di dati solo di tipo semplice; per poter inviare dati di tipo diverso si può usare la funzione `mpi.gather.Robj` che ha i seguenti argomenti:

```
1 mpi.gather.Robj(obj, root = 0, comm = 1)
```

Le funzioni `mpi.gather` e `mpi.gather.Robj` hanno inoltre un'estensione che permette di inviare i dati non solo al sottoprocesso `root`, ma a tutti gli altri sottoprocessi, implementando in questo modo una comunicazione *all-to-all*.

```
1 mpi.allgather(x, type, comm = 1)
```

In questo caso non è più richiesto il parametro `root` in ingresso.

Una funzione alternativa per effettuare una comunicazione *all-to-one* è `mpi.reduce`. Questa funzione permette di effettuare una operazione semplice su i dati ricevuti da tutti i sottoprocessi. Gli argomenti della funzione sono:

```
1 mpi.reduce(x, type=2, op=c("sum","prod","max","min","
    maxloc","minloc"), dest = 0, comm = 1)
```

in cui `x` è il dato di tipo `type` su cui verrà effettuata l'operazione `op`. Le operazioni che possono essere effettuate sono somma, prodotto, ricerca del massimo/minimo e del punto di massimo/minimo. Il risultato dell'operazione viene salvato nella variabile `x` del sottoprocesso `dest`. Anche di questa funzione esiste una variante che permette lo scambio di dati tra tutti i sottoprocessi, chiamata `mpi.allreduce`.

4.1.5 Struttura del codice

Per capire le funzioni descritte fino ad ora abbiamo analizzato i codici sorgenti del pacchetto `R` . Le definizioni delle funzioni `R` sono contenute nel file `Rcall.R`. La maggior parte delle funzioni di `Rmpi` sono semplicemente dei *wrapper* per le analoghe funzioni della libreria C. Un esempio di funzione contenuta nel file `Rcall.R` è `mpi.send`:

```
1 mpi.send <-function(x, type, dest, tag, comm=1){
2    .Call("mpi_send", .force.type(x,type), as.integer(type
       ), as.integer(dest), as.integer(tag), as.integer(
       comm), PACKAGE="Rmpi"')
3 }
```

All'interno della funzione `mpi.send` viene richiamata la funzione C `mpi_send`. Tale funzione è definita all'interno del file `Rmpi.c`. All'interno dell'*header file* `Rmpi.h` viene inclusa oltre alla libreria `R.h`, anche la libreria `mpi.h`:

```
1 #include <mpi.h>
2 #include <R.h>
```

La funzione `mpi_send` contenuta nel file `Rmpi.c` è la seguente:

```
1  SEXP mpi_send(SEXP sexp_data, SEXP sexp_type, SEXP
       sexp_dest, SEXP sexp_tag, SEXP sexp_comm){
2      int slen, len=LENGTH(sexp_data), type=INTEGER(
           sexp_type)[0], dest=INTEGER(sexp_dest)[0];
3      int commn=INTEGER(sexp_comm)[0], tag=INTEGER(sexp_tag)
           [0];
4      switch (type){
5          case 1: mpi_errhandler(MPI_Send(INTEGER(sexp_data),
                len, MPI_INT, dest, tag, comm[commn]));
6          break;
7          case 2: mpi_errhandler(MPI_Send(REAL(sexp_data),
               len, MPI_DOUBLE, dest, tag, comm[commn]));
8          break;
9          case 3: slen=LENGTH(STRING_ELT(sexp_data,0));
10             MPI_Send(CHAR2(STRING_ELT(sexp_data, 0)), slen,
                   MPI_CHAR, dest, tag, comm[commn]);
11         break;
12         case 4: MPI_Send(RAW(sexp_data), len, MPI_BYTE,
               dest, tag, comm[commn]);
13         break;
14         default:PROTECT(sexp_data=AS_NUMERIC(sexp_data));
15             mpi_errhandler(MPI_Send(REAL(sexp_data), 1,
                   datatype[0], dest, tag, comm[commn]));
16             UNPROTECT(1);
17         break;
18     }
19 return R_NilValue;
20 }
```

All'interno di questa funzione viene effettuato uno *switch* sui diversi tipi di dati che possono essere passati come argomento alla funzione. Nel caso per esempio in cui il dato inviato sia un intero, alla riga 5, viene effettuato un *cast* a intero del dato di tipo `SEXP`; il dato così convertito viene passato come argomento delle funzione C `MPI_Send`, contenuta nella libreria `mpi`. I dati di tipi diversi vengono trattati in modo analogo. Anche la funzione `mpi.send.Robj` richiama la funzione C `mpi.send`:

```
1 mpi.send.Robj <-function(obj,dest,tag,comm=1){mpi.send(x
      =.mpi.serialize(obj), type=4, dest=dest, tag=tag, comm
      =comm)}
```

modificando semplicemente l'oggetto `R` che si vuole inviare.

Tra le funzioni del pacchetto `Rmpi` sono presenti anche delle funzioni più complesse che non sono semplicemente *wrapper* per le funzioni della libreria C `mpi`. Un esempio è la funzione `mpi.apply`:

```
1  mpi.apply<-function (x,fun,...,comm=1) {
2     n <-length(x)
3     nslaves <-mpi.comm.size(comm) -1
4     if (nslaves < n)
5        stop("data length must be at most total slave size"
            )
6     if (!is.function(fun))
7        stop("fun is not a function")
8     length(list(...))
9     tag <-floor(runif(1, 1, 1000))
10    mpi.bcast.cmd(.mpi.slave.apply(), comm = comm)
11    mpi.bcast(as.integer(c(tag, n)), type = 1, comm = comm
         )
12    mpi.bcast.Robj(list(fun = fun, dot.arg = list(...)),
         rank = 0, comm = comm)
13    if (n < nslaves)
14       x = c(x, as.list(integer(nslaves -n)))
15    mpi.scatter.Robj(c(list("master"), as.list(x)), root =
          0, comm = comm)
16    out <-as.list(integer(n))
17    for (i in 1:n) {
18       tmp <-mpi.recv.Robj(mpi.any.source(), tag, comm)
19       src <-mpi.get.sourcetag()[1]
20       out[[src]] <-tmp
21    }
22    out
23 }
```

La funzione `mpi.apply` richiama infatti al suo interno altre funzioni del pacchetto `Rmpi` tra cui `mpi.bcast.cmd`, riga 10, `mpi.bcast`, riga 11, `mpi.bcast.Robj`, riga 12, `mpi.scatter.Robj`, riga 15 al fine di applicare ai sottoprocessi *slave* la funzione `fun` con un argomento diverso per ogni sottoprocesso.

4.1.6 Esempi codice

Abbiamo implementato tre versioni dell'algoritmo delle k-medie che sfruttano il calcolo parallelo. L'algoritmo delle k-medie è un algoritmo di clustering secondo cui, fissato il numero di cluster che si è interessati a trovare e fissato un rappresentante (media o un medoide) per ogni cluster, si associa

ogni dato ad un cluster in base alla distanza tra il dato e i diversi medoidi. Una volta associati i dati ai cluster, si possono ricalcolare i rappresentanti dei singoli cluster ed iterare la procedura descritta. Questo algoritmo si presta ad una sua facile parallelizzazione, nel caso in cui si usi come rappresentante la media. I passi dell'algoritmo possono infatti essere effettuati su sottoinsiemi di dati e i rappresentati dei cluster calcolati dai singoli sottoprocessi possono essere utilizzati per calcolare i rappresentanti dei cluster globali effettuando semplicemente una media.

Brute-Force

Questo metodo è il più semplice che può essere utilizzato per parallelizzare il codice delle k-medie. Il metodo *Brute-Force* consiste nel dividere il problema in N sottoproblemi, con N pari al numero di sottoprocessi *slave*. Nel caso delle k-medie la suddivisione del problema consiste nella suddivisione del dataset in N gruppi e nell'invio dell'i-esimo sottoinsieme di dati all'i-esimo sottoprocesso. Ogni sottoprocesso applica poi l'algoritmo delle k-medie al sottoinsieme dei dati inviatogli e invia al sottoprocesso *master* i medoidi calcolati a partire dal sottoinsieme di dati presenti nella sua memoria. Il sottoprocesso *master* calcola i medoidi globali effettuando la media dei medoidi calcolati dai singoli sottoprocessi.
La funzione inviata ai sottoprocessi, in cui si ipotizza di conoscere i medoidi (`Mk`) e i dati (`dati`), è:

```
1  kmean_slave<- function(){
2     ni=dim(dati)[1]
3     p=dim(Mk)[2]
4     K=dim(Mk)[1]
5     cluster=double(ni)
6     distanza=double(K)
7     for(i in 1:ni){
8        for(k in 1:K){
9           distanza[k]=distanzaE(dati[i,], Mk[k,])
10       }
11       cluster[i]=which.min(distanza)
12    }
13    for(k in 1:K){
14       Mk[k,]=mean(dati[cluster==k,])
15    }
16    result=list(M=Mk,C=cluster)
17    result
18 }
```

All'interno della funzione, alla riga 9, viene calcolata la distanza tra i dati inviati al sottoprocesso e i medoidi globali calcolati al passo precedente; successivamente, alla riga 11, viene assegnato ogni dato al cluster il cui medoide è più vicino. Infine la funzione `kmeans_slave` calcola, alla riga 14, i nuovi medoidi relativi ai dati inviati al sottoprocesso. I medoidi e gli assegnamenti ai cluster vengono poi restituiti dalla funzione alla riga 16.

Il sottoprocesso *master*, dopo aver diviso i dati in N gruppi e averli raccolti all'interno di una lista di N elementi, invia per mezzo di `mpi.scatter` gli elementi della lista ai singoli sottoprocessi:

```
1 mpi.scatter.Robj2slave(dati)
```

Invia inoltre per mezzo di `mpi.bcast` le funzioni a tutti i sottoprocessi:

```
1 mpi.bcast.Robj2slave(kmean_slave)
2 mpi.bcast.Robj2slave(distanzaE)
```

Dopo aver inviato a tutti i sottoprocessi i dati e le funzioni da eseguire, viene effettuato un numero fissato di iterazioni, sufficienti per arrivare a convergenza, all'interno delle quali il sottoprocesso *master* invia a tutti i sottoprocessi i medoidi calcolati all'iterazione precedente e fa eseguire a tutti i sottoprocessi la funzione `kmeans_slave`. Infine vengono calcolate le medie globali a partire dai risultati locali dei singoli sottoprocessi salvati in `rssresult`.

```
1 mpi.bcast.Robj2slave(Mk)
2 rssresult <- mpi.remote.exec(kmean_slave())
```

Task-Push

Il secondo metodo descritto risulta essere un po' più complesso del metodo *Brute-Force*, ed è utile nel caso in cui non si possa dividere il problema in un numero di sottoproblemi pari al numero di sottoprocessi. Per superare questo problema vengono assegnati a priori più sottoproblemi (*task*) allo stesso sottoprocesso; in particolare appena un sottoprocesso termina un *task* assegnatogli, gli vengono inviati i dati relativi al successivo *task* della coda. Tale processo prosegue finché non sono stati svolti tutti i *task* assegnati ad un sottoprocesso.

La funzione inviata a tutti i processi, in cui si ipotizza siano noti i dati, i medoidi (`Mk`) e un vettore (`fold`) contenente gli indici dei *task* a cui appartengono i singoli dati analizzati dal sottoprocesso, è la seguente:

```
1 kmean_slave <- function() {
2    task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())
3    task_info <- mpi.get.sourcetag()
```

```
4     tag <- task_info[2]
5     while (tag != 2) {
6        foldNumber <- task$foldNumber
7        ni=sum(fold==foldNumber)
8        p=dim(Mk)[2]
9        K=dim(Mk)[1]
10       cluster=double(ni)
11       distanza=double(K)
12       for(i in 1:ni){
13          for(k in 1:K){
14             distanza[k]=distanzaE(dati[fold==foldNumber
                  ,][i,], Mk[k,])
15          }
16          cluster[i]=which.min(distanza)
17       }
18       for(k in 1:K){
19          Mk[k,]=mean(dati[fold==foldNumber,][cluster==k
                  ,])
20       }
21       result <- list(result=Mk,C=cluster,foldNumber=
             foldNumber)
22       mpi.send.Robj(result,0,1)
23       task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
             ())
24       task_info <- mpi.get.sourcetag()
25       tag <- task_info[2]
26    }
27    junk <- 0
28    mpi.send.Robj(junk,0,2)
29 }
```

All'interno della funzione viene per prima cosa ricevuto un nuovo messaggio da una qualsiasi fonte con un qualsiasi *tag*, alla riga 2. Da tale messaggio viene estratta per prima cosa la *tag*. Se il messaggio ricevuto è etichettato con `tag=1` significa che è presente nella coda relativa al sottoprocesso almeno un altro *task* da effettuare e il messaggio contiene informazioni relative al primo di tali *task*; se invece il messaggio è etichettato con `tag=2` significa che sono terminati i *task* da effettuare. Finché il sottoprocesso non riceve un messaggio con etichetta pari a 2 viene effettuato il *task*. In particolare per prima cosa viene estratto il valore `foldNumber`, riga 6, che rappresenta il numero del *task* che si sta eseguendo, poi viene effettuato l'algoritmo delle k-medie classico calcolando la distanza tra i dati e i medoidi, alla riga 14, assegnando i dati al cluster il cui medoide è più vicino, alla riga 16, e calcolando i nuovi medoidi,

alla riga 19. I nuovi medoidi calcolati e i cluster a cui sono stati abbinati i dati vengono quindi inviati al sottoprocesso *master*, alla riga 21, e viene ricevuto un nuovo messaggio, alla riga 23. Una volta terminati i *task* assegnati al sottoprocesso, il sottoprocesso invia un messaggio al *master* con `tag=2`.

Come nel metodo descritto nella sezione precedente, il sottoprocesso *master*, divide i dati in NT gruppi, con NT pari al numero di sottoproblemi o *task*. Dato che l'assegnamento dei *task* ai sottoprocessi veiene effettuato a priori, può essere creata una lista di N elementi che contengono i dati che verranno utilizzati dai singoli sottoprocessi. Viene inoltre creata una seconda lista (`fold`) i cui elementi contengono informazioni relative al *task* in cui dovranno essere analizzati i dati. Queste due liste vengono inviate ai diversi sottoprocessi per mezzo di `mpi.scatter` che invia ogni elemento della lista al relativo sottoprocesso.

```
1 mpi.scatter.Robj2slave(dati)
2 mpi.scatter.Robj2slave(fold)
```

Oltre ai due vettori vengono inviate le funzioni `distanzaE` e `kmeans_slave` a tutti i sottoprocessi per mezzo di `mpi.bcast`:

```
1 mpi.bcast.Robj2slave(distanzaE)
2 mpi.bcast.Robj2slave(kmean_slave)
```

Anche il codice eseguito dal sottoprocesso *master* risulta essere più complesso rispetto a quello descritto nel metodo precedente. Dopo aver inviato a tutti i sottoprocessi i medoidi calcolati al passo precedente, il sottoprocesso *master* richiama la funzione `kmeans_slave` per mezzo di `mpi.bcast.cmd`, in modo che tutti i sottoprocessi *slave* siano in attesa di un messaggio da parte del sottoprocesso *master*.

```
1 mpi.bcast.Robj2slave(Mk)
2 mpi.bcast.cmd(kmean_slave())
```

Le informazioni riguardanti i *task* vengono inviate dal sottoprocesso *master* a tutti i sottoprocessi *slave* per mezzo di `mpi.send`. L'oggetto che viene inviato è il numero del *task* che verrà eseguito; tale messaggio viene inviato con etichetta pari ad 1, che indica che il messaggio contiene un nuovo *task*.

```
1 for (i in 1:NTASKS) {
2    slave_id <- slave_ids[i]
3    mpi.send.Robj(Tasks[[i]],slave_id,1)
4 }
```

Una volta inviati i messaggi contenenti i *task*, il sottoprocesso *master* ricevere i risultati dai diversi sottoprocessi per mezzo di `mpi.recv`. Tali dati vengono utilizzati per calcolare i medoidi globali e assegnare ogni dato ad un cluster.

```
for (i in 1:length(tasks)) {
   message <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
      ())
   ...
}
```

Dopo aver ricevuto i risultati relativi a tutti i *task*, viene chiuso l'*handshake* inviando a tutti i sottoprocessi un messaggio vuoto con `tag=2` per indicare che sono stati inviati tutti i *task* e si attende di ricevere un messaggio da tutti i sottoprocessi *slave* anch'esso con `tag=2` per indicare che l'esecuzione della funzione è terminata.

```
for (i in 1:n_slaves) {
   junk <- 0
   mpi.send.Robj(junk,i,2)
}
for (i in 1:n_slaves) {
   mpi.recv.Robj(mpi.any.source(),2)
}
```

Task-pull

L'ultimo metodo che presentiamo è il più complesso e il più completo tra i tre. Questo modello permette, come il precedente, di definire un numero di *task* superiore al numero di sottoprocessi; i *task* non vengono però assegnati a priori ai diversi sottoprocessi ma si crea una coda che permette di assegnare un *task* al primo sottoprocesso "libero", fino a che non sono stati eseguiti tutti i *task*.

La funzione che viene eseguita da tutti i sottoprocessi *slave*, che considera noti solo i medoidi (`Mk`) e un vettore (`fold`) che rappresenta l'indice del *task* che si sta eseguendo, è la seguente:

```
kmean_slave <- function() {
   junk <- 0
   done <- 0
   while (done != 1) {
      mpi.send.Robj(junk,0,1)
      task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
         ())
      task_info <- mpi.get.sourcetag()
      tag <- task_info[2]
      if (tag == 1) {
         foldNumber=task$foldNumber
         dati <- task$dati
```

```
12          ni=dim(dati)[1]
13          p=dim(Mk)[2]
14          K=dim(Mk)[1]
15          cluster=double(ni)
16          distanza=double(K)
17          for(i in 1:ni){
18             for(k in 1:K){
19                distanza[k]=distanzaE(dati[i,], Mk[k,])
20             }
21             cluster[i]=which.min(distanza)
22          }
23          for(k in 1:K){
24             Mk[k,]=mean(dati[cluster==k,])
25          }
26          results <- list(result=Mk,C=cluster,foldNumber=
               foldNumber)
27          mpi.send.Robj(results,0,2)
28       }
29       else if (tag == 2) {
30          done <- 1
31       }
32    }
33    mpi.send.Robj(junk,0,3)
34 }
```

Nella funzione `kmeans_slave` sono presenti numerosi scambi di messaggi per verificare se il sottoprocesso è pronto a ricevere un nuovo *task* o se tutti i *task* sono stati eseguiti. Queste informazioni come nel metodo precedente vengono scambiate per mezzo dell'etichetta legata al messaggio inviato/ricevuto. In particolare nei messaggi ricevuti dal sottoprocesso *master* `tag=1` significa che il messaggio contiene informazioni relative ad un nuovo *task* da eseguire, `tag=2` significa che tutti i *task* sono stati eseguiti. Nei messaggi che i sottoprocessi *slave* inviano al sottoprocesso *master* `tag=1` significa che il sottoprocesso è pronto a ricevere un nuovo *task*, `tag=2` significa che il *task* è stato eseguito e il messaggio contiene i risultati, `tag=3` serve invece a chiudere l'*handshake*. Appena il sottoprocesso è pronto ad eseguire un nuovo *task* invia al sottoprocesso *master* un messaggio vuoto con `tag=1`, alla riga 5, e si mette in attesa di ricevere un messaggio da parte del sottoprocesso *master*, alla riga 6. Se tale messaggio ha `tag=1` si estraggono le informazioni riguardanti il *task* e si esegue l'algoritmo delle k-medie, se invece ha `tag=2` (ovvero i *task* sono conclusi), il sottoprocesso non aspetta più di ricevere nuovi messaggi e invia un messaggio vuoto con `tag=3` al *master*, alla riga 33. Nel caso in cui il messaggio ricevuto abbia `tag=1`, vengono estratti dal

messaggio i dati su cui effettuare le k-medie e il numero del *task* (contenuto nella variabile `foldNumber`). Dopo aver estratto i dati, viene effettuato l'algoritmo delle k-medie calcolando le distanze, alla riga 19, tra i dati e i medoidi, assegnando i dati ai cluster, alla riga 21, e calcolando i nuovi medoidi, alla riga 24. I dati così calcolati vengono poi inviati al sottoprocesso *master* con `tag=2`.

Per eseguire l'algoritmo delle k-medie nei diversi sottoprocessi, come nei metodi precedenti, vengono inizialmente inviati, per mezzo di `mpi.bcast`, i medoidi iniziali, il vettore contenente l'assegnamento di ogni dato ad un *task*, la funzione per calcolare la distanza e la funzione per calcolare le k-medie `kmeans_slave`.

```
1 mpi.bcast.Robj2slave(fold)
2 mpi.bcast.Robj2slave(Mk)
3 mpi.bcast.Robj2slave(distanzaE)
4 mpi.bcast.Robj2slave(kmean_slave)
```

Ad ogni passo dell'algoritmo delle k-medie il sottoprocesso *master* esegue le seguenti istruzioni:

```
1  tasks <- vector('list')
2  for (i in 1:NTASKS) {
3     tasks[[i]] <- list(foldNumber=i,dati=dati[fold==i,])
4  }
5  mpi.bcast.Robj2slave(Mk)
6  mpi.bcast.cmd(kmean_slave())
7  junk <- 0
8  closed_slaves <- 0
9  n_slaves <- mpi.comm.size()-1
10 temp=matrix(NA,K,dim(Mk)[1])
11 while (closed_slaves < n_slaves) {
12    message <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
          ())
13    message_info <- mpi.get.sourcetag()
14    slave_id <- message_info[1]
15    tag <- message_info[2]
16    if (tag == 1) {
17       if (length(tasks) > 0) {
18          mpi.send.Robj(tasks[[1]], slave_id, 1);
19          tasks[[1]] <- NULL
20       }
21       else {
22          mpi.send.Robj(junk, slave_id, 2)
23       }
```

```
24      }
25      else if (tag == 2) {
26          ...
27      }
28      else if (tag == 3) {
29          closed_slaves <- closed_slaves + 1
30      }
31  }
32  Mk=...
```

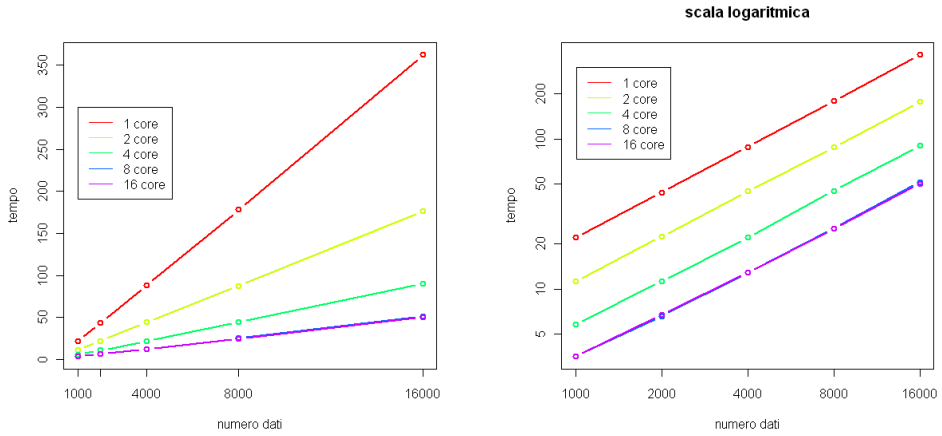
Ad ogni passo dell'algoritmo delle k-medie il sottoprocesso *master* invia i medoidi calcolati al passo precedente, riga 5, richiama la funzione `kmeans_slave`, riga 6, e crea una lista di *task* che contengono il numero del *task* e i dati da utilizzare, alla riga 3. Prima di inviare tali *task* ai sottoprocessi, il sottoprocesso *master* riceve un messaggio dai sottoprocessi, riga 12. Se il messaggio ricevuto ha `tag=1` significa che il sottoprocesso è pronto ad eseguire un nuovo *task*; se sono presenti *task* non ancora svolti, il *master* invia i dati relativi al nuovo *task* con `tag=1`, riga 18, ed elimina il *task*, se invece i *task* sono stati tutti eseguiti, il *master* invia un messaggio vuoto con `tag=2`, riga 22. Se invece il messaggio ricevuto dal sottoprocesso ha `tag=2`, significa che il messaggio contiene i risultati dell'algoritmo applicati ad un sottoinsieme di dati; i risultati vengono salvati e a partire da quelli vengono calcolati i medoidi globali. Se infine il sottoprocesso ha inviato un messaggio con `tag=3`, significa che il sottoprocesso ha terminato di eseguire la funzione `kmeans_slave`. In questo modo viene chiuso l'*handshake* tra sottoprocesso *master* e *slave*.

4.1.7 Simulazioni e confronto

Per valutare le prestazioni dei metodi descritti in precedenza abbiamo effettuato delle simulazioni con un numero di sottoprocessi *slave* pari 2, 4, 8, 16 su un dataset di 1000, 2000, 4000, 8000 e 16000 punti in $\mathbb{R}^2$. Il numero di iterazioni dell'algoritmo delle k-medie è stato mantenuto fissato in tutte le simulazioni, pari ad un valore che permette di arrivare sempre a convergenza.

Brute-Force

I risultati ottenuti utilizzando il metodo *Brute-Force* sono rappresentati in Figura 4.1 e 4.2. Si può notare in Figura 4.1b un andamento uguale in scala logaritmica del tempo impiegato ad analizzare dati di dimensioni diverse per un numero diverso di sottoprocessi. In tutte le figure si osserva inoltre che utilizzando 16 sottoprocessi non si ottiene un miglioramento ulteriore delle



(a) Tempi di esecuzione

(b) Tempi di esecuzione in scala logaritmica

Figura 4.1: Tempi di esecuzione del metodo Brute-Force per l'algoritmo delle k-medie con 2, 4, 8 e 16 sottoprocessi *slave*

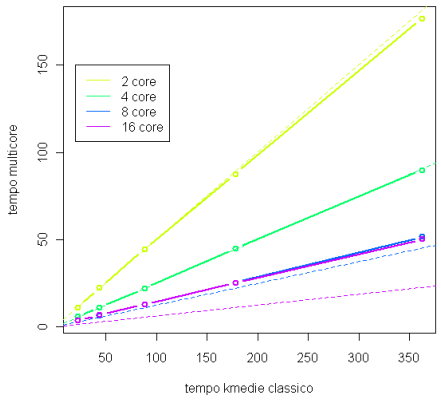


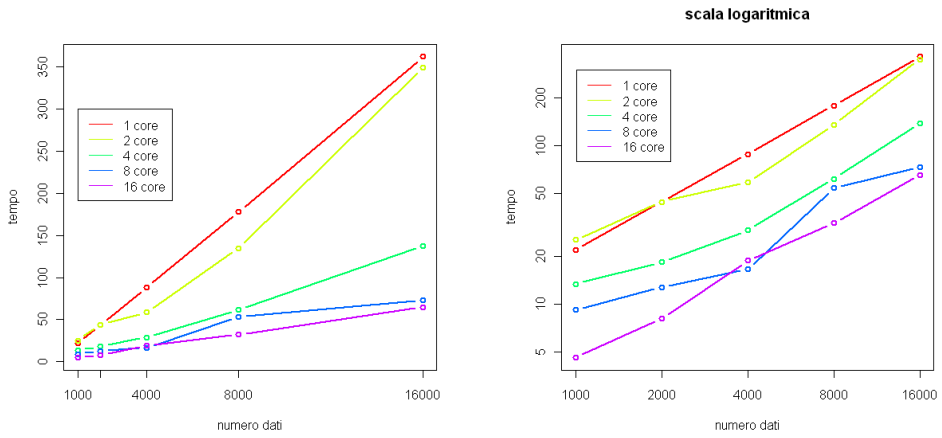
Figura 4.2: Confronto tra i tempi di esecuzione dell'algoritmo delle k-medie parallelo per mezzo del metodo Brute-Force e dell'algoritmo classico. Le linee tratteggiate rappresentano l'andamento teorico ottimale.

prestazioni dell'algoritmo con i dati considerati. La Figura 4.2 mostra invece il confronto tra il tempo effettivo impiegato ad eseguire l'algoritmo delle k-medie in parallelo con un numero diverso di sottoprocessi e il tempo impiegato ad eseguire l'algoritmo delle k-medie classico, a parità di dimensione dei dati. Si può notare che con 2 e 4 sottoprocessi *slave* l'andamento reale coincide con quello teorico mentre con 8 sottoprocessi *slave* l'andamento reale inizia

a discostarsi leggermente dall'andamento teorico. L'andamento reale con 16 sottoprocessi risulta invece essere molto distante dall'andamento teorico in quanto non migliora le prestazioni ottenute con 8 sottoprocessi.

Task-Push

I risultati ottenuti utilizzando il metodo *Task-Push*, con un numero di *task* pari a $2N$ e N pari al numero di sottoprocessi, sono rappresentati in Figura 4.3 e 4.4. Si può notare già in Figura 4.3a che nel caso si usino solo 2 sottoprocessi *slave* l'algoritmo parallelo non risulta essere efficiente rispetto all'algoritmo sequenziale classico. Anche osservando l'andamento in scala logaritmica in Figura 4.3b, si nota che l'utilizzo di 2 sottoprocessi non porta ad un miglioramento rispetto all'utilizzo dell'algoritmo sequenziale. La Figura 4.4 mostra il confronto tra il tempo effettivo impiegato ad eseguire l'algoritmo delle k-medie in parallelo con un numero diverso di sottoprocessi e il tempo impiegato ad eseguire l'algoritmo delle k-medie classico, a parità di dimensione dei dati. Si può notare che in nessun caso l'andamento reale risulta essere confrontabile con l'andamento teorico. Questo algoritmo, che rappresenta un passo intermedio tra gli algoritmi *Brute-Force* e *Task-pull*, è utile per capire lo scambio di informazioni tra sottoprocessi ma risulta essere poco efficiente.



(a) Tempi di esecuzione

(b) Tempi di esecuzione in scala logaritmica

Figura 4.3: Tempi di esecuzione del metodo Task-Push per l'algoritmo delle k-medie con 2, 4, 8 e 16 sottoprocessi *slave*

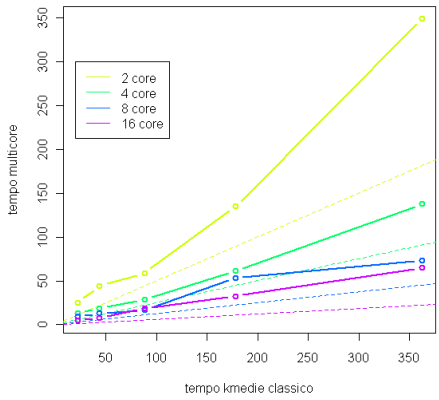
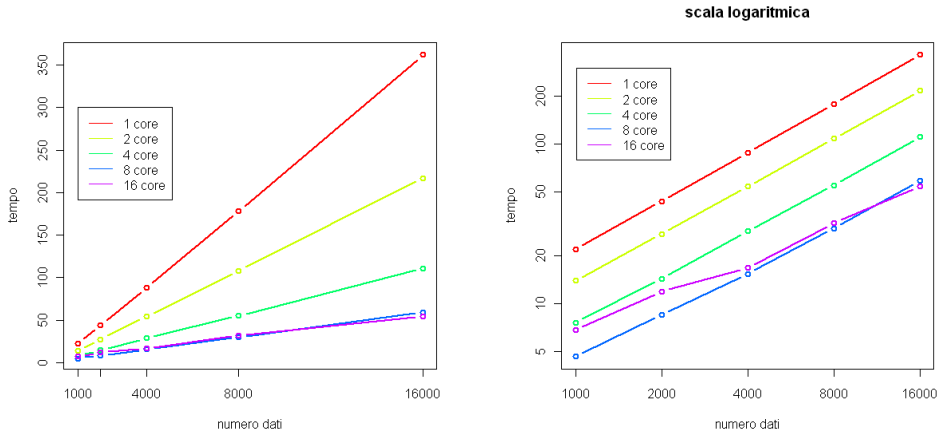


Figura 4.4: Confronto tra i tempi di esecuzione dell'algoritmo delle k-medie parallelo per mezzo del metodo Task-Push e dell'algoritmo classico. Le linee tratteggiate rappresentano l'andamento teorico ottimale.

Task-pull



(a) Tempi di esecuzione

(b) Tempi di esecuzione in scala logaritmica

Figura 4.5: Tempi di esecuzione del metodo Task-pull per l'algoritmo delle k-medie con 2, 4, 8 e 16 sottoprocessi *slave*

I risultati ottenuti utilizzando il metodo *Task-pull*, con un numero di *task* pari a $2N$ e N pari al numero di sottoprocessi, sono rappresentati in Figura 4.7 e 4.6. Come nel metodo *Brute-Force*, si può notare in Figura 4.3b un andamento in scala logaritmica del tempo impiegato ad analizzare

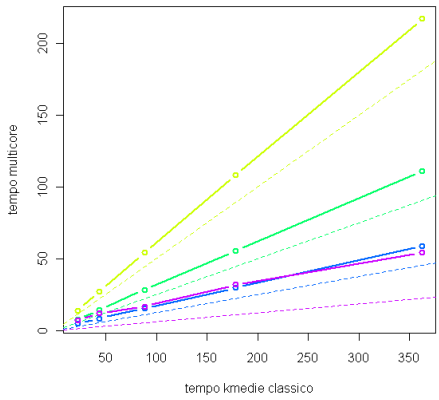
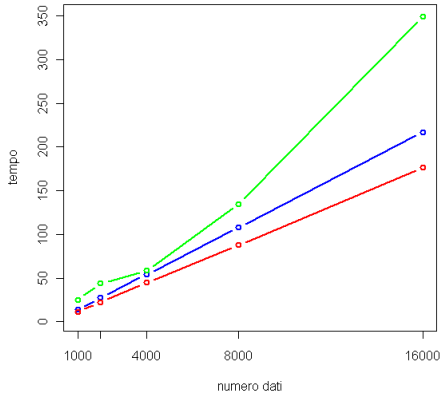


Figura 4.6: Confronto tra i tempi di esecuzione dell'algoritmo delle k-medie parallelo per mezzo del metodo Task-pull e dell'algoritmo classico. Le linee tratteggiate rappresentano l'andamento teorico ottimale.

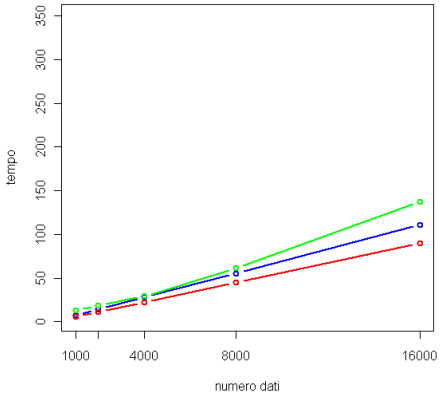
dati di dimensioni diverse simile per numeri diversi di sottoprocessi creati. Anche per il metodo *Task-pull* si osserva che utilizzando 16 sottoprocessi non si ottiene un miglioramento ulteriore delle prestazioni dell'algoritmo con i dati considerati. La Figura 4.4 mostra il confronto tra il tempo effettivo impiegato ad eseguire l'algoritmo delle k-medie in parallelo con un numero diverso di sottoprocessi e il tempo impiegato ad eseguire l'algoritmo delle k-medie sequenziale, a parità di dimensione dei dati. Si può notare che tutti gli andamenti reali si discostano leggermente dall'andamento teorico. In particolare nel caso in cui vengano creati 16 sottoprocessi *slave* non si ottiene un miglioramento rispetto ai risultati ottenuti con 8 sottoprocessi.

Confronto metodi

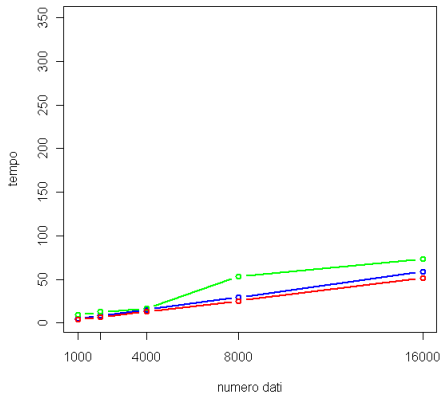
In questa sezione abbiamo infine confrontato i diversi metodi a parità di numero di sottoprocessi *slave* creati. Si può notare in tutti i casi che il metodo *Brute-Force* risulta essere il metodo più efficiente, ma all'aumentare del numero di sottoprocessi creati il divario tra i diversi metodi diminuisce. Il metodo *Task-Push*, come già notato, risulta essere sempre il meno efficiente. Tale metodo risulta infatti essere utile per comprendere scambi di informazioni più complessi tra sottoprocessi prima di affrontare il metodo *Task-pull*, ma l'assegnamento a priori dei *task* ai sottoprocessi risulta essere non ottimale. Il metodo *Task-pull* è il metodo più complesso, tra quelli presentati, per quanto riguarda lo scambio di informazioni tra sottoprocessi. Tale metodo risulta essere molto utile nel caso in cui il problema non sia divisibile



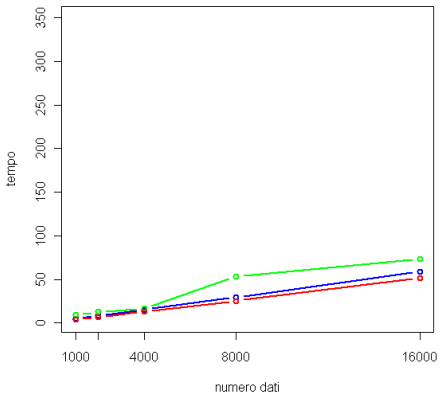
(a) Tempi di esecuzione con 2 sottoprocessi *slave*



(b) Tempi di esecuzione con 4 sottoprocessi *slave*



(c) Tempi di esecuzione con 8 sottoprocessi *slave*



(d) Tempi di esecuzione con 16 sottoprocessi *slave*

Figura 4.7: Tempi di esecuzione dei metodi Brute-Force (in rosso), Task-Push (in verde) e Task-pull (in blu).

in un numero di *task* pari al numero di sottoprocessi e nel caso in cui sia necessario effettuare un *load balancing* perché eseguito su una macchina con sottoprocessori con prestazioni diverse.

4.2 snow

Un altro pacchetto R che permette di implementare algoritmi paralleli in R è snow [8]. Questo pacchetto è basato su diversi protocolli di comunicazione, tra cui MPI, PVM, SOCK e NWS. Anche in questo caso viene creata una comunicazione di tipo *master-slave* tra sottoprocessi all'interno di un sistema a memoria distribuita.

4.2.1 Principali funzioni di snow

La libreria snow viene caricata per mezzo del comando

```
1 library("snow")
```

Una volta caricata la libreria è possibile creare un *cluster* di N sottoprocessi per mezzo dell'istruzione

```
1 cl<-makeCluster(N, type=getClusterOption("type"), ...)
```

La funzione makeCluster ha un argomento type che permette di specificare quale tipo di protocollo di comunicazione usare. Esistono inoltre delle varianti di makeCluster specifiche per i diversi protocolli: makeMPIcluster, makePVMcluster, makeSOCKcluster e makeNWScluster.

Per chiudere i sottoprocessi creati precedentemente e appartenenti al *cluster* cl si utilizza il comando:

```
1 stopCluster(cl)
```

4.2.2 Invio/ricezione di dati ed esecuzione di funzioni

Le funzioni definite nel pacchetto snow non permettono di scrivere codici flessibili come quelli basati sul pacchetto Rmpi. Sono implementate nel pacchetto solo poche funzioni e tali funzioni risultano essere molto efficienti [6]. La prima di queste funzioni è clusterSplit:

```
1 clusterSplit(cl, seq)
```

Tale funzione permette di dividere (in parti uguali) una sequenza seq tra i sottoprocessi appartenenti al *cluster* cl. Il risultato di tale funzione viene salvato all'interno di una lista di dimensione pari al numero di sottoprocessi appartenenti al *cluster* cl.

Le funzioni implementate nel pacchetto al fine di eseguire una funzione all'interno dei sottoprocessi sono 2: clusterCall e clusterApply. La prima di queste funzioni ha i seguenti argomenti:

```
1 clusterCall(cl,fun,...)
```

Questa funzione permette di eseguire la funzione `fun` su tutti i sottoprocessi appartenenti al *cluster* `cl` con argomenti uguali, passati come argomento al posto di .... I risultati vengono salvati all'interno di una lista di dimensione pari al numero di sottoprocessi.

La seconda funzione è `clusterApply`, che ha come argomenti:

```
clusterApply(cl,x,fun,...)
```

in cui `fun` rappresenta la funzione che deve essere eseguita in tutti i sottoprocessi appartenenti al *cluster* `cl`. A differenza della funzione `clusterCall`, `clusterApply` permette di passare argomenti diversi ai sottoprocessi; tali argomenti sono definiti all'interno di un `array` di dimensione pari al numero di sottoprocessi. Se la dimensione dell'`array` è superiore al numero di sottoprocessi, l'esecuzione della funzione con con argomento `x[i]` verrà assegnata al sottoprocesso $i \mod N$ con N pari al numero di sottoprocessi. Al posto di ... possono inoltre essere passati degli argomenti uguali per tutti i sottoprocessi. I risultato ottenuto da ogni sottoprocesso viene infine salvato all'interno di una lista di dimensione pari a `x`. All'interno del pacchetto `snow` è stata implementata una versione *load balancing* di `clusterApply`, definita `clusterApplyLB`. Tale funzione ha gli stessi argomenti di `clusterApply` e differisce da essa solo per il fatto che se `x` ha dimensione maggiore del numero di sottoprocessi, i dati in eccesso verranno assegnati ai cluster che terminano per primi l'esecuzione della funzione, e non in base al *rank*.

4.2.3 Struttura del codice

Le funzioni del pacchetto `snow` non sono dei semplici *wrapper* di funzioni C ma richiamano comunque tali funzioni al loro interno, come avviene nella funzione `mpi.apply`. Il codice riportato è relativo alla funzione `ClusterApply`:

```
static ClusterApply <-function(cl, fun, n, argfun) {
   checkCluster(cl)
   p <-length(cl)
   if (n > 0 && p > 0) {
      val <-vector("list", n)
   start <-1
   while (start <= n) {
      end <-min(n, start + p -1)
      jobs <-end -start + 1
      for (i in 1:jobs)
         sendCall(cl[[i]], fun, argfun(start + i -1))
         val[start:end] <-lapply(cl[1:jobs], recvResult)
         start <-start + jobs
      }
```

```
15        checkForRemoteErrors(val)
16    }
17 }
```

Tale funzione richiama alla riga 11 la funzione `sendCall` che nella versione MPI richiama al suo interno la funzione `sendData.MPInode`:

```
1 sendData.MPInode <-function(node, data)
2    mpi.send.Robj(data, node$rank, node$SENDTAG, node$comm
        )
```

La funzione `sendData.MPInode` si limita infine a richiamare a sua volta la funzione `mpi.send.Robj` contenuta nel pacchetto `Rmpi` e descritta in precedenza.

Appendice A

Codici Calcolo Parallelo

A.1 Brute-force

```
1  library("Rmpi")
2  dati=...
3  Mk=... #medoidi iniziali
4
5  #creazione sottoprocessi
6  N=... #numero di sottoprocessi
7  mpi.spawn.Rslaves(nslaves=N)
8
9  #funzione che calcola la distanza euclidea tra 2 vettori
      v1 e v2
10 distanzaE<- function(v1, v2){
11    distanzaE=sqrt(sum((v1-v2)^2))
12 }
13
14 #funzione che calcola l'appartenenza ad un cluster per i
      sottoprocessi
15 #si ipotizza di conoscere Mk=medoidi, dati
16 kmean_slave<- function(){
17    ni=dim(dati)[1]
18    p=dim(Mk)[2]
19    K=dim(Mk)[1]
20    cluster=double(ni)
21    distanza=double(K)
22    #calcolo della distanza tra i dati e i medoidi
23    for(i in 1:ni){
24       for(k in 1:K){
25          distanza[k]=distanzaE(dati[i,], Mk[k,])
```

```
26          }
27          #assegnazione del cluster
28          cluster[i]=which.min(distanza)
29      }
30      #calcolo delle medie per i dati relativi al
            sottoprocesso
31      for(k in 1:K){
32          Mk[k,]=mean(dati[cluster==k,])
33      }
34      result=list(M=Mk,C=cluster)
35      result
36  }
37
38
39  #assegnazione dei dati ai sottoprocessi
40  fold <- rep(1:N,length=n)
41
42  #creazione di una lista di dimensione pari al numero di
        sottoprocessi
43  #ogni elemento della lista contiene i dati da inviare al
        relativo sottoprocesso
44  Dati=dati
45
46  dati=list()
47  for(i in 1:N){
48      dati[[i]]=Dati[fold==i,]
49  }
50
51  # Invio dei dati ai sottoprocessi
52  mpi.scatter.Robj2slave(dati)
53
54  # Invio delle funzioni ai sottoprocessi
55  mpi.bcast.Robj2slave(kmean_slave)
56  mpi.bcast.Robj2slave(distanzaE)
57
58  NITER=10
59  for(l in 1:NITER){
60      # invio ad ogni iterazione delle medie calcolate all'
            iterazione precedente
61      mpi.bcast.Robj2slave(Mk)
62      # Esecuzione della funzione da parte di tutti i
            sottoprocessi e raccolta dei risultati
63      rssresult <- mpi.remote.exec(kmean_slave())
```

```
64
65     #calcolo delle medie globali
66     for(k in 1:K){
67        temp=matrix(NA,N,dim(Mk)[2])
68        for(i in 1:N){
69           temp[i,]=as.matrix(rssresult[[i]]$M[k,])
70        }
71        temp=data.frame(temp)
72        Mk[k,]=mean(temp)
73     }
74  }
75  #chiusura dei sottoprocessi
76  mpi.close.Rslaves()
```

A.2 Task-push

```
1  library("Rmpi")
2  dati=...
3  Mk=...#medoidi
4
5  #creazione sottoprocessi
6  N=... #numero di sottoprocessi
7  mpi.spawn.Rslaves(nslaves=N)
8
9  #funzione che calcola la distanza euclidea tra 2 vettori
      v1 e v2
10 distanzaE<- function(v1, v2){
11    distanzaE=sqrt(sum((v1-v2)^2))
12 }
13
14 #funzione che calcola l'appartenenza ad un cluster per i
      sottoprocessi
15 #si ipotizza di conoscere Mk=medoidi, dati
16 #fold=vettore di indici il cui i esimo elemento
      rappresenta il sottoprocesso da cui sarà analizzato l'
      i-esimo dato
17 #foldNumber=rank del sottoprocesso in cui si sta
      lavorando
18 kmean_slave <- function() {
19    # Ricevo un nuovo task
20    task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag())
21    task_info <- mpi.get.sourcetag()
22    tag <- task_info[2]
```

```
23     #Finché non si riceve una tag=2 (che indica che tutti
          io task sono stati effettuati) si esegue il task
24     while (tag != 2) {
25        # Eseguo il task
26        foldNumber <- task$foldNumber
27        ni=sum(fold==foldNumber)
28        p=dim(Mk)[2]
29        K=dim(Mk)[1]
30        cluster=double(ni)
31        distanza=double(K)
32        #calcolo della distanza tra i dati e i medoidi
33        for(i in 1:ni){
34           for(k in 1:K){
35              distanza[k]=distanzaE(dati[fold==foldNumber
                   ,][i,], Mk[k,])
36           }
37           #assegnazione del cluster
38           cluster[i]=which.min(distanza)
39       }
40        #calcolo delle medie per i dati relativi al
             sottoprocesso
41        for(k in 1:K){
42           Mk[k,]=mean(dati[fold==foldNumber,][cluster==k
                ,])
43        }
44        #Costruisco e invio i risultati
45        result <- list(result=Mk,C=cluster,foldNumber=
             foldNumber)
46        mpi.send.Robj(result,0,1)
47        # Ricevo un nuovo task
48        task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
             ())
49        task_info <- mpi.get.sourcetag()
50        tag <- task_info[2]
51     }
52     #Conclusi tutti i task
53     #Invio il messaggio: conclusi tutti i task
54     junk <- 0
55     mpi.send.Robj(junk,0,2)
56  }
57
58  #assegnazione dei dati ai diversi task
59  NTASKS=4
```

```
60 fold <- rep(1:NTASKS,length=n)
61 fold <- sample(fold)
62
63 #invio della funzione ai sottoprocessi
64 mpi.bcast.Robj2slave(distanzaE)
65 mpi.bcast.Robj2slave(kmean_slave)
66
67 #creazione lista task
68 tasks <- 1:NTASKS
69
70
71 # Creazione di una lista round-robin per i sottoprocessi
72 n_slaves <- mpi.comm.size()-1
73 slave_ids <- rep(1:n_slaves, length=NTASKS)
74
75 Dati=dati
76 Fold=fold
77
78 dati_temp=list()
79 dati=list()
80 fold_temp=list()
81 fold=list()
82 temp_i=rep(0,N)
83 for(i in 1:NTASKS){
84 dati_temp[[i]]=Dati[Fold==i,]
85 fold_temp[[i]]=Fold[Fold==i]
86 if(temp_i[slave_ids[i]]==0){
87 dati[[slave_ids[i]]]=dati_temp[[i]]
88 fold[[slave_ids[i]]]=fold_temp[[i]]
89 temp_i[slave_ids[i]]=temp_i[slave_ids[i]]+1
90 }
91 else{
92 dati[[slave_ids[i]]]=rbind(dati[[slave_ids[i]]],dati_temp
      [[i]])
93 fold[[slave_ids[i]]]=c(fold[[slave_ids[i]]],fold_temp[[i
      ]])
94 temp_i[slave_ids[i]]=temp_i[slave_ids[i]]+1
95 }
96 }
97
98 Tasks <- vector('list')
99 for (i in 1:NTASKS) {
100    Tasks[[i]] <- list(foldNumber=i)
```

```
101 }
102
103
104 #invio dei dati ai sottoprocessi
105 mpi.scatter.Robj2slave(dati)
106 mpi.scatter.Robj2slave(fold)
107
108 NITER=10
109 for(l in 1:NITER){
110    #invio ad ogni iterazione delle medie calcolate all'
          iterazione precedente
111    mpi.bcast.Robj2slave(Mk)
112    #Chiamo la funzione in  tutti i sottoprocessi in modo
          che sia pronta a riceve i task
113    mpi.bcast.cmd(kmean_slave())
114
115    # Invio tasks
116    for (i in 1:NTASKS) {
117       slave_id <- slave_ids[i]
118       mpi.send.Robj(Tasks[[i]],slave_id,1)
119    }
120
121    # raccolta risultati
122    temp=matrix(NA,K,dim(Mk)[1])
123    for (i in 1:length(tasks)) {
124       message <- mpi.recv.Robj(mpi.any.source(),mpi.any.
             tag())
125       foldNumber <- message$foldNumber
126       results    <- message$result
127       cluster<-message$C
128       #calcolo delle medie globali
129       for(k in 1:K){
130          temp[k,]=mean(data.frame(rbind(temp[k,],results[
                k,])),na.rm=T)
131       }
132    }
133    Mk=temp
134
135    # Chiusura handshake
136    for (i in 1:n_slaves) {
137       junk <- 0
138       mpi.send.Robj(junk,i,2)
139    }
```

```
140     for (i in 1:n_slaves) {
141         mpi.recv.Robj(mpi.any.source(),2)
142     }
143 }
144 #chiusura dei sottoprocessi
145 mpi.close.Rslaves()
```

A.3 Task-pull

```
1  library("Rmpi")
2  dati=...
3  Mk=... #medoidi
4
5  N=...
6  mpi.spawn.Rslaves(nslaves=N)
7
8  #funzione che calcola la distanza euclidea tra 2 vettori
       v1 e v2
9  distanzaE<- function(v1, v2){
10     distanzaE=sqrt(sum((v1-v2)^2))
11 }
12
13 #funzione che calcola l'appartenenza ad un cluster per i
       sottoprocessi
14 #si ipotizza di conoscere Mk=medoidi, dati
15 #fold=vettore di indici il cui i esimo elemento
       rappresenta il sottoprocesso da cui sarà analizzato l'
       i-esimo dato
16 #foldNumber=rank del sottoprocesso in cui si sta
       lavorando
17 kmean_slave <- function() {
18     # Tag per i messaggi inviati:
19     #      1=pronto per un nuovo task, 2=task compiuto, 3=
           uscita
20     # Tag per i messaggi ricevuti:
21     #      1=task, 2=tutti i task effettuati
22     junk <- 0
23     done <- 0
24     while (done != 1) {
25         # Processo pronto a ricevere un nuovo task
26         # Invio un messaggio per comunicarlo al master
27         mpi.send.Robj(junk,0,1)
28         # Ricevo un nuovo task
```

```
29        task <- mpi.recv.Robj(mpi.any.source(),mpi.any.tag
              ())
30        task_info <- mpi.get.sourcetag()
31        tag <- task_info[2]
32        if (tag == 1) { #contiene un task il messaggio
              ricevuto
33           foldNumber=task$foldNumber
34           dati <- task$dati
35           ni=dim(dati)[1]
36           p=dim(Mk)[2]
37           K=dim(Mk)[1]
38           cluster=double(ni)
39           distanza=double(K)
40           #calcolo della distanza tra i dati e i medoidi
41           for(i in 1:ni){
42              for(k in 1:K){
43                 distanza[k]=distanzaE(dati[i,], Mk[k,])
44              }
45              #assegnazione del cluster
46              cluster[i]=which.min(distanza)
47           }
48           #calcolo delle medie per i dati relativi al
                 sottoprocesso
49           for(k in 1:K){
50              Mk[k,]=mean(dati[cluster==k,])
51           }
52           # Costruisco e invio i risultati
53           results <- list(result=Mk,C=cluster,foldNumber=
                 foldNumber)
54           mpi.send.Robj(results,0,2)
55        }
56        else if (tag == 2) { #task conclusi
57           done <- 1
58        }
59        # Se il messaggio ha tag diverso da 1 o 2 viene
              ignorato
60     }
61     #conclusi tutti i task
62     #invio messaggio: ho finito tutto, sto uscendo!
63     mpi.send.Robj(junk,0,3)
64  }
65
66
```

```
67 #assegnazione dei dati ai diversi task
68 NTASKS=4
69 fold <- rep(1:NTASKS,length=n)
70 fold <- sample(fold)
71
72 # Invio dei dati ai sottoprocessi
73 mpi.bcast.Robj2slave(dati)
74 mpi.bcast.Robj2slave(fold)
75 mpi.bcast.Robj2slave(Mk)
76
77 #invio della funzione ai sottoprocessi
78 mpi.bcast.Robj2slave(distanzaE)
79 mpi.bcast.Robj2slave(kmean_slave)
80
81 NITER=10
82 for(l in 1:NITER){
83    #creazione lista task ad ogni iterazione
84    tasks <- vector('list')
85    for (i in 1:NTASKS) {
86       tasks[[i]] <- list(foldNumber=i,dati=dati[fold==i
          ,])
87    }
88    # invio ad ogni iterazione delle medie calcolate all'
       iterazione precedente
89    mpi.bcast.Robj2slave(Mk)
90    # Chiamo la funzione in  tutti i sottoprocessi in modo
        che sia pronta a riceve i task
91    mpi.bcast.cmd(kmean_slave())
92
93    junk <- 0
94    closed_slaves <- 0
95    n_slaves <- mpi.comm.size()-1
96    temp=matrix(NA,K,dim(Mk)[1])
97    while (closed_slaves < n_slaves) {
98       # Ricevo un messaggio da un sottoprocesso
99       message <- mpi.recv.Robj(mpi.any.source(),mpi.any.
          tag())
100      message_info <- mpi.get.sourcetag()
101      slave_id <- message_info[1]
102      tag <- message_info[2]
103      if (tag == 1) { #sottoprocesso pronto a rivere un
          task
```

```
104           if (length(tasks) > 0) { #se ci sono ancora task
                  invio un task al sottoprocesso
105              mpi.send.Robj(tasks[[1]], slave_id, 1);
106              tasks[[1]] <- NULL   #rimuovo il task dalla
                    lista
107           }
108           else { #comunico che non esistono più task
109              mpi.send.Robj(junk, slave_id, 2)
110           }
111        }
112        else if (tag == 2) { #il sottoprocesso ha inviato i
               risultati
113           # raccolta risultati
114           foldNumber <- message$foldNumber
115           results <- message$result
116           cluster <- message$C
117           for(k in 1:K){
118              temp[k,]=mean(data.frame(rbind(temp[k,],
                    results[k,])),na.rm=T)
119           }
120        }
121        else if (tag == 3) { #il sottoprocesso si è chiuso
122           closed_slaves <- closed_slaves + 1 #aumento il
                 numero di sottoprocessi chiusi di 1
123        }
124     }
125     Mk=temp
126  }
127  #chiusura dei sottoprocessi
128  mpi.close.Rslaves()
```

Bibliografia

[1] R Development Core Team (2009). *R: A language and environment for statistical computing. R Foundation for Statistical Computing*, Vienna, Austria. URL http://www.R-project.org

[2] W. N. Venables, D. M. Smith and the R Development Core Team, *An Introduction to R* (2010). URL http://cran.r-project.org/doc/manuals/R-intro.pdf

[3] R Development Core Team, *R language definition* (2010). URL http://cran.r-project.org/doc/manuals/R-lang.pdf

[4] J.M. Chambers, *Software for data analysis: programming with R.* Springer Science+Business Media (2008).

[5] J.M. Chambers, D.T. Lang, *Object-Oriented Programming in R.* The Newsletter of the R Project (2001)

[6] Schmidberger, Morgan, Eddelbuettel, Yu, Tierney, Mansmann,*State of the Art in Parallel Computing with R.* Journal of Statistical Software, August 2009, Volume 31, Issue 1. (2009)

[7] Hao Yu (2001). *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface).* R package version 0.5-8. http://www.stats.uwo.ca/faculty/yu/Rmpi

[8] Luke Tierney, A. J. Rossini, Na Li and H. Sevcikova. *snow: Simple Network ofWorkstations.* R package version 0.3-3.