DOTTORATO DI RICERCA

in

MATEMATICA APPLICATA E INFORMATICA

Ciclo XVI

Consorzio tra Università di Catania, Università di Napoli Federico II,
Seconda Università di Napoli, Università di Palermo, Università di Salerno

SEDE AMMINISTRATIVA: UNIVERSITÀ DI NAPOLI FEDERICO II

Francesco Gregoretti

Applicazioni parallele (multi-sito) in ambiente Grid

TESI DI DOTTORATO DI RICERCA

· *The law of life* ·
· *The law of love* ·

Ringraziamenti

Ringrazio il Prof. Almerico Murli che mi ha formato ed avviato alla ricerca e che, quale mio tutore nel corso di dottorato, mi ha guidato e consigliato.

Ringrazio poi la Prof.ssa Luisa D'Amore per la disponibilità dimostrata e i suggerimenti fornitimi; i miei colleghi, Gennaro Oliva e Laura Antonelli, e i ricercatori del Condor Team, Alain Roy e Jamie Frey, per aver collaborato attivamente alla realizzazione di questo lavoro.

Desidero inoltre ringraziare i colleghi con cui ho condiviso le giornate in laboratorio e tutti gli amici che mi sono stati vicini, moralmente e materialmente.

Un ringraziamento particolare va a Seri per avermi incoraggiato nei momenti più difficili, spronandomi sempre ad andare avanti.

Un grazie di cuore anche a mia madre che ha sempre riposto fiducia cieca in quello che faccio, subendo spesso il mio brontolare davanti allo schermo del computer.

Rivolgo infine un sentito ringraziamento e un saluto ai membri del mio gruppo musicale (One Starving Day), Pippo, Dario, Andrea e Marco, perchè la nostra musica è espressione della mia sensibilità.

Introduzione

Nel campo delle architetture di calcolo ad alte prestazioni, il Grid Computing rappresenta ad oggi la frontiera delle attività di ricerca, come è anche testimoniato dall'elevato numero di progetti dedicati a tale tecnologia.

Il concetto di Grid si è evoluto considerevolmente nel corso di questi anni. La crescente popolarità del Grid ha avuto come risultato la nascita di vari tipi di "Grids": Grid di dati, Grid Computazionali, Grid di applicazioni e servizi, Grid di strumentazione, Grid di sensori, ... [22].

La tendenza attuale è far convergere i concetti relativi alle architetture, protocolli, e applicazioni di queste "Grids" allo scopo di formulare un unico paradigma: il Grid.

Nel 1998, Ian Foster e Carl Kesselman formularono la seguente definizione [24]:

> "una griglia computazionale è un'infrastruttura hardware e software che fornisce un accesso affidabile, consistente, pervasivo ed economico a risorse high-end di calcolo".

Successivamente, nel 2000, gli stessi Kasselman e Foster, insieme a Steve Tuecke, rividero [25] tale definizione per porre l'accento sulle questioni relative alle politiche di accesso, affermando che il concetto di Grid si riferisce alla:

> "condivisione coordinata di risorse e problem solving nell'ambito di Organizzazioni Virtuali (*Virtual Organizations*) multi-istituzionali e dinamiche".

Tale condivisione è altamente controllata: i fornitori delle risorse e gli utenti si accordano per definire chiaramente e accuratamente quali sono le risorse condivise, a chi è consentito l'accesso e sotto quali condizioni. Un insieme di individui e/o istituzioni che rispettano tali regole di condivisione viene denominato Virtual Organization.

In [22] sono elencate le caratteristiche significative che possono essere adottate per determinare se un sistema di calcolo distribuito soddisfa i requisiti per essere una Grid.

Un tale sistema, si definisce una Grid, se:

- coordina risorse che non sono soggette ad un controllo centralizzato - una Grid aggrega e coordina risorse che provengono da domini amministrativi diversi, e affronta le problematiche relative alla sicurezza e alle politiche di utilizzo.

- utilizza protocolli e interfacce standard, aperti e universali - una Grid è formata da protocolli e interfacce universali per la gestione di questioni fondamentali quali la sicurezza, l'autorizzazione, la scoperta delle risorse, l'accesso alle risorse; l'adozione di protocolli e interfacce standard e aperti consente l'interoperabilità e la realizzazione di un'infrastruttura comune.

- assicura la fornitura di qualità di servizio (*Quality of Service* - QoS) non comuni - una Grid consente alle risorse che la costituiscono di essere utilizzate in modo coordinato per fornire diverse qualità di servizio, per esempio, in relazione al tempo di risposta[1], al throughput, alla disponibilità e/o alla co-allocazione di molteplici tipi di risorse, per incontrare le più complesse richieste degli utenti.

Gli esperimenti condotti in diversi campi scientifici e tecnologici, come la visualizzazione scientifica, l'esplorazione spaziale, le simulazioni computazionali nei vari campi delle scienze e dell'ingegneria, la ricerca medica, generano

[1] il tempo che trascorre tra la sottomissione del processo e l'ottenimento della prima risposta.

immense quantità di dati, la cui analisi richiede la disponibilità di un'enorme potenza computazionale.

Le tecnologie Grid, consentendo la condivisione sicura e l'utilizzo coordinato di risorse distribuite geograficamente, mettono a disposizione degli utenti un'enorme potenza di calcolo, attraverso l'acquisizione di un elevato numero di risorse computazionali difficilmente disponibile localmente. In particolare l'interconnessione dei supercalcolatori e dei cluster disponibili presso i diversi centri realizza una struttura globale che consente l'esecuzione delle applicazioni candidate alla risoluzione di complessi problemi "computationally intensive" e "data intensive".

Più in generale l'aggregazione di risorse high-end (software, processori, memorie, sistemi di memorizzazione) resa possibile dalle tecnologie Grid, è finalizzata a fornire a un insieme di utenti una enorme potenza di calcolo *on-demand*.

Tuttavia lo sviluppo e la gestione dell'esecuzione di un'applicazione Grid evidenzia alcune problematiche legate all'eterogeneità e dinamicità dell'ambiente.

Tra i requisiti dell'ambiente software necessari all'esecuzione di una computazione in una Grid costituita da risorse dislocate geograficamente, rivestono particolare rilevanza:

- l'abilità di scoprire, acquisire e gestire con affidabilità le risorse computazionali in maniera dinamica;
- la capacità di utilizzare le risorse
  - ignorandone struttura e localizzazione e ignorando i meccanismi dell'infrastruttura sottostante che ne consentono l'utilizzo (trasparenza),
  - senza perdere di vista lo stato dei task computazionali distribuiti,
  - evitando di intervenire direttamente in caso di fallimenti (*fault-tolerance*).

Il calcolo su Grid impone quindi la realizzazione di strumenti di sviluppo adeguati e di nuovi sistemi per la gestione dell'esecuzione delle applicazioni in modo da soddisfare i requisiti succitati.

D'altra parte l'uso efficiente di una Grid impone spesso la realizzazione di nuove applicazioni distribuite o modifiche sostanziali alle applicazioni parallele tradizionali.

Infatti, fondamentalmente, il modello computazionale utilizzato per la programmazione su Grid è il tradizionale modello a scambio di messaggi.

In particolare le applicazioni parallele multi-sito consistono di più componenti (subjob) eseguiti in parallelo su uno o più processori di uno o più cluster o supercalcolatori presso differenti siti. Tali applicazioni possono così accedere efficacemente ad un numero di risorse computazionali molto più ampio di quello a disposizione utilizzando un qualsiasi supercalcolatore o cluster singolo. Un'opportuna distribuzione dei subjob tra le risorse disponibili consente a queste applicazioni di beneficiare in termini di prestazioni di questa potenza computazionale aggregata nonostante l'overhead addizionale introdotto dalle comunicazioni (tra i subjob) che utilizzano le reti più lente (per esempio WAN).

In questo lavoro è stata realizzata un'implementazione parallela multi-sito dell'algoritmo del Gradiente Coniugato a Blocchi (BCG) (utilizzato come testbed di applicazioni parallele multi-sito) ed è stato progettato e realizzato un sistema (denominato MPI jobs management system) per la gestione dell'esecuzione di applicazioni parallele (secondo le specifiche MPI) multi-sito sui cluster distribuiti di una Grid.

È stata inoltre utilizzata una libreria denominata MGF (MPI Globus Forwarder) [34, 32, 33], per la realizzazione di applicazioni per il calcolo distribuito ad elevate prestazioni secondo le specifiche MPI.

L'algoritmo del BCG rispetto all'algoritmo standard del CG presenta caratteristiche che lo rendono più consono ad un'elaborazione in ambiente Grid. La relativa implementazione parallela multi-sito fa uso di due livelli di parallelismo: il primo riflette la decomposizione in task dell'algoritmo a blocchi (gra-

na grossa), il secondo è realizzato suddividendo i task computazionalmente più onerosi tra processi paralleli (grana fine) (Capitolo 1).

L'MPI jobs management system consente di sottomettere le applicazioni parallele multi-sito ad un sistema Grid, sollevando l'utente dall'individuazione e acquisizione simultanea delle risorse necessarie alla loro esecuzione. Esso consente di schedulare i subjob paralleli, eseguirli sui calcolatori più appropriati, tenendo conto sia dei requisiti e preferenze dei subjob, sia delle caratteristiche delle risorse, e gestirne la sincronizzazione. Requisiti e preferenze dovranno riflettere le esigenze dei diversi subjob paralleli in termini di complessità computazionale e complessità delle comunicazioni, in modo da ottimizzare il mapping con le risorse disponibili (Capitolo 2).

L'utilizzo della libreria MGF consente di eseguire in modo trasparente ed efficiente le applicazioni multi-sito realizzate secondo le specifiche MPI su più calcolatori paralleli di una Grid, compresi i cluster a rete privata (Capitolo 3).

L'MPI jobs management system e MGF costituiscono un ulteriore strato middleware basato su strumenti presistenti e consolidati quali il Globus Toolkit, Condor-G e MPICH-G2 descritti brevemente nell'Appendice A.

Indice

Capitolo 1

Implementazione parallela Grid (multi-sito) dell'algoritmo del Gradiente Coniugato a Blocchi

L'algoritmo del Gradiente Coniugato [30, 50] è ampiamente utilizzato per la risoluzione di sistemi lineari.

La sua struttura non risulta però consona ad un'elaborazione in ambiente di calcolo parallelo e più in generale in ambiente Grid.

La formulazione standard (algoritmo 1) dell'algoritmo parallelo, comune ad alcune librerie "di qualità" come *Aztec* [56] e *PETSc* [12], presenta infatti numerosi punti di sincronizzazione e un numero elevato di messaggi di piccole dimensioni da scambiare.

In particolare, distribuendo la matrice dei coefficienti per blocchi di righe (figura 1.1), i nuclei computazionali eseguiti in parallelo per ogni iterazione possono essere così riassunti (algoritmo 1):

1. Prodotto matrice-vettore (eseguito una volta);
2. Aggiornamento di vettori mediante operazione di tipo AXPY (eseguito tre volte);

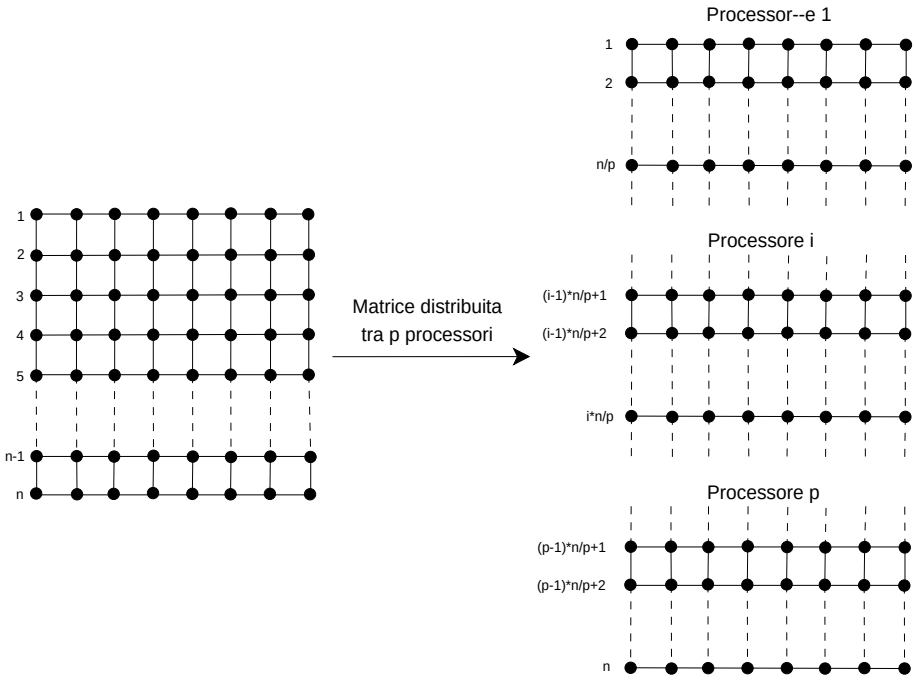

Figura 1.1: Distribuzione Matrice per blocchi di righe.

3. Prodotto scalare (eseguito due volte).

I nuclei computazionali 1. e 3. richiedono comunicazioni globali e quindi sincronizzazioni tra i processi.

I costi di sincronizzazione possono incidere maggiormente in sistemi Grid, dove le risorse computazionali hanno diverse tipologie di interconnessioni (WAN, LAN, Reti ad elevate prestazioni, ...) caratterizzate da latenze e ampiezze di banda estremamente variabili.

La versione a blocchi dell'algoritmo del Gradiente Coniugato [44] consente di ridurre il numero dei punti di sincronizzazione e aumentare le dimensioni dei messaggi da scambiare.

Eseguendo più operazioni floating-point tra i diversi punti di sincronizzazione tale algoritmo risulta più tollerante alla latenza e pertanto si adatta

Algoritmo 1 Formulazione standard dell'algoritmo parallelo del Gradiente Coniugato

applicato al sistema $Ax = b$, con A matrice di dimensione $n \times n$ simmetrica e definita positiva, $r = b$ - Ax residuo di x e $\{p_1,\ldots,p_n\}$ vettori non nulli A-coniugati (cioè soddisfacenti ${p_i}^T A p_j = 0, \forall i \neq j$).
$k = 0$; $p_0 = 0$; $\beta_0 = 0$; $r_0 = b$; $rrdot_0 = <r_0, r_0>$
while $r_k \geq AZ\_tol$
  $k = k + 1$
  $p_k = r_{k-1} + \beta_{k-1} p_{k-1}$
  $q_k = A p_k$
  $pqdot_k = {p_k}^T q_k$
  $\alpha_k = rrdot_{k-1} \,/\, pqdot_k$
  $x_k = x_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1}$ - $\alpha_k q_k$
  $rrdot_k = {r_k}^T r_k$
  $\beta_k = rrdot_k \,/\, rrdot_{k-1}$
end

meglio ai sistemi distribuiti e ai sistemi Grid rispetto all'algoritmo standard del Gradiente Coniugato.

L'algoritmo a blocchi risulta decomposto in sottoproblemi (task) che possono essere risolti concorrentemente e che non hanno la stessa complessità computazionale, pertanto si presta all'introduzione di due livelli di parallelismo: a grana grossa e a grana fine.

Il primo livello riflette la decomposizione in task, il secondo corrisponde alla decomposizione dei task computazionalmente più onerosi tra processi paralleli.

Nei paragrafi successivi verranno descritti la versione a blocchi dell'algoritmo del Gradiente Coniugato e la relativa implementazione.

1.1 Versione a blocchi dell'algoritmo del Gradiente Coniugato

La versione a blocchi dell'algoritmo del Gradiente Coniugato può essere considerata in due contesti.

Se dobbiamo risolvere un sistema di equazioni lineari di ordine n con b vettori dei termini noti differenti, il Gradiente Coniugato a blocchi produce simultaneamente un vettore soluzione per ognuno di loro, in al più $\lceil n/b \rceil$ passi e richiedendo eventualmente meno operazioni di quelle che richiederebbe applicare l'algoritmo del Gradiente Coniugato b volte.

Consideriamo il problema di risolvere il sistema di equazioni lineari

$$AX = B \tag{1.1}$$

dove A è una matrice simmetrica di ordine n e B di dimensioni $n \times b$.

1.1.1 Algoritmo di Lanczos a Blocchi

L'algoritmo del Gradiente Coniugato a blocchi sfrutta una trasformazione di similitudine della matrice A. Il punto di partenza per la sua costruzione è l'algoritmo di Lanczos a Blocchi [15, 31, 58].

Data la matrice B di dimensioni $n \times b$, scelto ν_1 di dimensioni $b \times b$ tale che $Z_1 = B\nu_1$ soddisfa $Z_1^T Z_1 = I_{(b \times b)}$ e dato $Z_0 = 0_{(n \times b)}$, l'iterazione di base dell'algoritmo di Lanczos a Blocchi è la seguente: per $k = 1, 2, \ldots$

$$Z_{k+1} = (AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T})\nu_{k+1}, \tag{1.2}$$

dove

$$\rho_k = Z_k^T A Z_k$$

e ν_{k+1} è scelto in modo che $Z_{k+1}^T Z_{k+1} = I_{(b \times b)}$.

Per calcolare la 1.2 osserviamo che posto

$$\widetilde{Z}_{k+1} = AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T} \Rightarrow Z_{k+1} = \widetilde{Z}_{k+1}\nu_{k+1},$$

le matrici (di Lanczos) Z_{k+1} possono essere generate tramite fattorizzazione ortogonale delle matrici $\widetilde{Z}_{k+1}$ (utilizzando la fattorizzazione QR di Householder o di Givens o l'algoritmo di Gram-Schmidt modificato):

$$\widetilde{Z}_{k+1} = Z_{k+1}\nu_{k+1}^{-1}.$$

Ovvero le matrici Z_{k+1} e ν_{k+1}^{-1} sono i fattori Q ed R di tale fattorizzazione; le matrici Z_1 e ν_1^{-1} sono i fattori Q ed R della fattorizzazione:

$$B = Z_1\nu_1^{-1}. \tag{1.3}$$

Dopo

$$k \leq \overline{k} = \lceil n/b \rceil$$

passi la 1.2 produce una decomposizione della matrice A come:

$$A\mathcal{Z}_k = \mathcal{Z}_k T_k + \widetilde{\mathcal{Z}}_k \tag{1.4}$$

dove

$$\mathcal{Z}_k = [Z_1, Z_2, \ldots, Z_k],$$

$\widetilde{\mathcal{Z}}_k$ è una matrice di dimensioni $n \times kb$

$$\widetilde{\mathcal{Z}}_k = [0_{(n\times(k-2)b)}, \widetilde{Z}_{k-1}]$$

e T_k è una matrice tridiagonale

$$T_k = \begin{bmatrix} \rho_1 & \nu_2^{-T} & & & \\ \nu_2^{-1} & \rho_2 & \nu_3^{-T} & & \\ & & \ddots & & \\ & & & \ddots & \nu_k^{-T} \\ & & & \nu_k^{-1} & \rho_k \end{bmatrix}.$$

Le colonne di $\mathcal{Z}_{\overline{k}}$ formano una base ortonormale per lo spazio dei vettori n-dimensionali, cosicchè è possibile esprimere la soluzione $X^\star$ del sistema 1.1 come

$$X^\star = \mathcal{Z}_{\overline{k}}\hat{\psi}_{\overline{k}}, \tag{1.5}$$

dove $\Psi_{\overline{k}}$ è un vettore n-dimensionale.
Il sistema 1.1 diventa

$$A\mathcal{Z}_{\overline{k}}\hat{\psi}_{\overline{k}} \overset{(1.1)}{=} B \overset{(1.3)}{=} Z_1\nu_1^{-1},$$

e, moltiplicando per $\mathcal{Z}_{\overline{k}}^T$ (essendo $\mathcal{Z}_{\overline{k}}^T A\mathcal{Z}_{\overline{k}} = T_{\overline{k}}$ per la 1.4),

$$T_{\overline{k}}\hat{\psi}_{\overline{k}} = \mathcal{Z}_{\overline{k}}^T B = \begin{bmatrix} \nu_1^{-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{1.6}$$

Consideriamo la fattorizzazione QR di T_k

$$T_k\mathcal{V}_k^T = L_k, \tag{1.7}$$

dove
$\mathcal{V}_k^T\mathcal{V}_k = I$ e

$$L_k = \begin{bmatrix} L_{1,1} & & & & & & \\ L_{2,1} & L_{2,2} & & & & & \\ L_{3,1} & L_{3,2} & L_{3,3} & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & L_{k-2,k-4} & L_{k-2,k-3} & L_{k-2,k-2} & & \\ & & & L_{k-1,k-3} & L_{k-1,k-2} & L_{k-1,k-1} & \\ & & & & L_{k,k-2} & L_{k,k-1} & \overline{L}_{k,k} \end{bmatrix}.$$

Le matrici $L_{i,j}$ hanno dimensioni $b \times b$.
Introducendo le variabili

$$W_k = \mathcal{Z}_k \mathcal{V}_k^T, \tag{1.8}$$

$$\psi_k = \mathcal{V}_k \hat{\psi}_k, \tag{1.9}$$

il sistema 1.6 diventa

$$T_{\overline{k}} \hat{\psi}_{\overline{k}} \overset{(1.7)}{=} L_k \mathcal{V}_k \hat{\psi}_k \overset{(1.9)}{=} L_{\overline{k}} \psi_{\overline{k}} = \begin{bmatrix} \nu_1^{-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

e

$$X_{\overline{k}} \overset{(1.5)}{=} \mathcal{Z}_{\overline{k}} \hat{\psi}_{\overline{k}} \overset{(1.9)}{=} \mathcal{Z}_{\overline{k}} \mathcal{V}_{\overline{k}}^T \psi_{\overline{k}} \overset{(1.8)}{=} W_{\overline{k}} \psi_{\overline{k}}$$

fornisce la soluzione al passo $\overline{k}$.

1.1.2 Algoritmo del Gradiente Coniugato a Blocchi (BCG)

Seguendo [45] l'algoritmo del BCG risulta decomposto in otto task che possono essere eseguiti concorrentemente (algoritmo 2).

- **Fase di inizializzazione**: dati B e X_0 di dimensioni $n \times b$, siano

$$R_0 = B - AX_0,$$

$$\hat{X}_0 = X_0,$$

$$Z_0 = 0_{(n \times b)}$$

e

$$Z_1 = R_0 \nu_1,$$

Algoritmo 2 Algoritmo del BCG

Fase di inizializzazione $R_0 = B - AX_0$
$\hat{X}_0 = X_0$
$Z_0 = 0_{(n\times b)}$
fattorizzazione QR di R_0: $R_0 = Z_1\nu_1^{-1}$
$\overline{W}_1 = Z_1$
$\rho_0 = 0_{(b\times b)}$
$\rho_1 = Z_1^T A Z_1$
$\overline{L}_{1,1} = \rho_1$
$V_1 = I_{(2b\times 2b)}$

do $k = 1, 2, \ldots, \overline{k} = \lceil n/b \rceil$

Task 1 $\widetilde{Z}_{k+1} = AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T}$
fattorizzazione QR di $\widetilde{Z}_{k+1}$: $\widetilde{Z}_{k+1} = Z_{k+1}\nu_{k+1}^{-1}$

Task 2 AZ_{k+1}
$\rho_{k+1} = Z_{k+1}^T AZ_{k+1}$

Task 3 $[L_{k+1,k-1}, \overline{L}_{k+1,k}] = [0_{(b\times b)}, \nu_{k+1}^{-1}]V_k^T$

Task 4 fatt. LQ di $[\overline{L}_{k+1,k-1}, \nu_{k+1}^{-T}]$: $[\overline{L}_{k+1,k-1}, \nu_{k+1}^{-T}] = [L_{k,k}, 0_{(b\times b)}]V_{k+1}$

Task 5 $[L_{k+1,k}, \overline{L}_{k+1,k+1}] = [\overline{L}_{k+1,k}, \rho_{k+1}]V_{k+1}^T$

Task 6 $[W_k, \overline{W}_{k+1}] = [\overline{W}_k, Z_{k+1}]V_{k+1}^T$

Task 7 **if** $(k = 1)$
$\psi_1 = L_{1,1}^{-1}\nu_1^{-1}$
elseif $(k > 1)$
$\psi_k = -L_{k,k}^{-1}(L_{k,k-2}\psi_{k-2} + L_{k,k-1}\psi_{k-1})$
endif

Task 8 $\hat{X}_k = \hat{X}_{k-1} + W_k\psi_k$

enddo

Fase finale $\overline{\psi}_{k+1} = -\overline{L}_{k+1,k+1}(L_{k+1,k-1}\psi_{k-1} + L_{k+1,k}\psi_k)$
$X_{k+1} = X_k + \overline{W}_{k+1}\overline{\psi}_{k+1}$

dove ν_1 è tale che $Z_1^T Z_1 = I_{(b\times b)}$.

Z_1 può essere calcolato come fattore Q della fattorizzazione QR di R_0:

$$R_0 = Z_1 \nu_1^{-1}.$$

Siano, inoltre,

$$\overline{W}_1 = Z_1,$$

$$\rho_0 = 0_{(b\times b)},$$

$$\rho_1 = Z_1^T A Z_1,$$

$$\overline{L}_{1,1} = \rho_1$$

e

$$V_1 = I_{(2b\times 2b)}.$$

Al passo k ($k = 1, 2, \ldots, \overline{k} = \lceil n/b \rceil$):

- **Task 1** - si calcola la matrice

$$\widetilde{Z}_{k+1} = AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T}$$

e la fattorizzazione QR di $\widetilde{Z}_{k+1}$:

$$\widetilde{Z}_{k+1} = Z_{k+1}\nu_{k+1}^{-1}.$$

In questo modo si ottiene la matrice triangolare superiore ν_{k+1}^{-1} e la matrice di Lanczos Z_{k+1},

$$Z_{k+1} = (AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T})\nu_{k+1},$$

tale che $Z_{k+1}^T Z_{k+1} = I_{(b\times b)}$.

- **Task 2** - si calcolano:

$$AZ_{k+1}$$

e

$$\rho_{k+1} = Z_{k+1}^T A Z_{k+1}.$$

- **Task 3** - si calcolano i due blocchi di L

$$[L_{k+1,k-1}, \overline{L}_{k+1,k}] = [0_{(b\times b)}, \nu_{k+1}^{-1}] V_k^T.$$

- **Task 4** - si calcola la fattorizzazione LQ di $[\overline{L}_{k+1,k-1}, \nu_{k+1}^{-T}]$ (di dimensioni $b \times 2b$):

$$[\overline{L}_{k+1,k-1}, \nu_{k+1}^{-T}] = [L_{k,k}, 0_{(b\times b)}] V_{k+1}.$$

In questo modo si ottiene la matrice triangolare inferiore $L_{k,k}$ e la matrice ortogonale V_{k+1} tali che:

$$[\overline{L}_{k+1,k-1}, \nu_{k+1}^{-T}] V_{k+1}^T = [L_{k,k}, 0_{(b\times b)}]$$

e

$$V_{k+1}^T V_{k+1} = I_{(2b\times 2b)}.$$

- **Task 5** - si calcolano i due blocchi di L

$$[L_{k+1,k}, \overline{L}_{k+1,k+1}] = [\overline{L}_{k+1,k}, \rho_{k+1}] V_{k+1}^T.$$

- **Task 6** - si calcolano

$$[W_k, \overline{W}_{k+1}] = [\overline{W}_k, Z_{k+1}]V_{k+1}^T.$$

- **Task 7** - si calcola ψ_k in modo che sia soddisfatta

$$L_{1,1}\psi_1 = \nu_1^{-1}, \text{ se } k = 1,$$

$$L_{k,k-2}\psi_{k-2} + L_{k,k-1}\psi_{k-1} + L_{k,k}\psi_k = 0, \text{ se } k > 1.$$

- **Task 8** - si calcola la soluzione al passo k

$$\hat{X}_k = \hat{X}_{k-1} + W_k\psi_k.$$

- Infine, al termine del passo $\overline{k}$, si calcolano $\overline{\psi}_{k+1}$ e X_{k+1} da

$$L_{k+1,k-1}\psi_{k-1} + L_{k+1,k}\psi_k + \overline{L}_{k+1,k+1}\overline{\psi}_{k+1} = 0,$$

$$X_{k+1} = X_k + \overline{W}_{k+1}\overline{\psi}_{k+1}.$$

1.2 Implementazione: il parallelismo a 2 livelli nel BCG

L'algoritmo del BCG suggerisce l'introduzione di due livelli di parallelismo: a grana grossa e a grana fine.

Il primo livello (grana grossa) riflette la decomposizione in task 1-8 dell'algoritmo descritto nel paragrafo 1.1.2. Ogni task viene eseguito da uno o più processi indipendenti.

Il secondo livello (grana fine) corrisponde alla decomposizione dei task computazionalmente più onerosi tra processi paralleli. In particolare dalla tabella 1.1, che mostra il numero di operazioni floating-point richiesto da ciascun task, considerando che $n \gg b$, risulta che i task 1, 2, 6 ed 8 hanno una complessità computazionale molto maggiore rispetto agli altri.

Ai due livelli di parallelismo corrispondono due classi di comunicazioni:

Task	Totale operazioni floating-point
1	$4nb^2 + k(4nb^2)\ (1 \leq k \leq 2b)$
2	$4nb^2$
3	$4b^3$
4	$5.5b^3 + 7b^2 + 4.5b\,(*, /) + 5.5b^3 + 1.5b\,(+, -)$
5	$8b^3$
6	$8nb^2$
7	$6b^3$
8	$2nb^2$

Tabella 1.1: Costo computazionale richiesto da ciascun task.

- inter-task: tra processi coinvolti in diversi task;
- intra-task: tra processi coinvolti nello stesso task.

Nei paragrafi successivi verranno descritti in dettaglio il parallelismo a grana grossa e il parallelismo a grana fine.

1.2.1 Parallelismo a grana grossa

Il parallelismo a grana grossa riflette la decomposizione in task dell'algoritmo descritto nel paragrafo 1.1.2. Ogni task viene eseguito da uno o più processi indipendenti utilizzando le risorse Grid disponibili.

I task vengono assegnati alle risorse di calcolo disponibili in base alla:

- complessità computazionale
  - i task computazionalmente più onerosi verranno assegnati alle risorse con maggiore potenza di calcolo in modo da bilanciare il carico computazionale complessivo;
- complessità delle comunicazioni inter-task
  - i task verranno assegnati alle risorse in base alla quantità di dati scambiati tra gli stessi, tenendo conto del fatto che le comunica-

zioni possono avvenire su diverse tipologie di reti (WAN, LAN, Reti ad alte prestazioni).

Al fine di definire un'opportuna distribuzione dei task che tenga conto di questi criteri, consideriamo il grafo delle dipendenze tra i task rappresentato in figura 1.2.

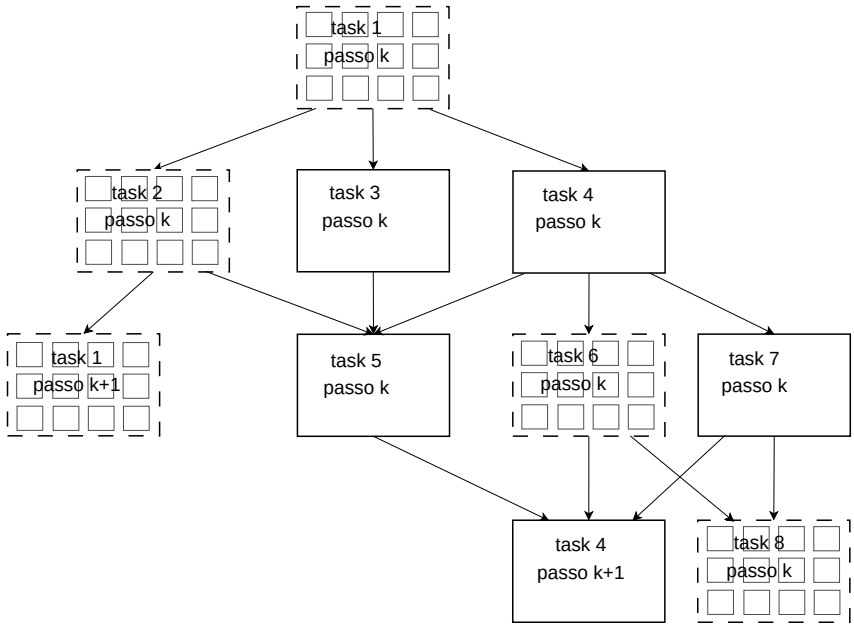

Figura 1.2: Grafo delle dipendenze dei task dell'algoritmo del BCG.

Nel grafo in questione (più precisamente un DAG), i nodi rappresentano i task mentre gli archi rappresentano le dipendenze tra i task.

Nella tabella 1.2 è invece rappresentato il flusso dei dati in ingresso e in uscita per ognuno dei task. Se si attribuisce come peso ad ogni arco del grafo la quantità di dati scambiati tra i task agli estremi, si ottiene il grafo pesato rappresentato in figura 1.3.

La distribuzione dei task viene fatta in modo da bilanciare la complessità delle comunicazioni (inter-task) tenendo conto del peso degli archi: i task

2 (k-1) 2 (k-1)	$\rho_k(b \times b) \rightarrow$ $AZ_k(n \times b) \rightarrow$	1 (k)	$\nu_{k+1}(b \times b) \rightarrow$ $\nu_{k+1}(b \times b) \rightarrow$ $Z_{k+1}(n \times b) \rightarrow$ $Z_{k+1}(n \times b) \rightarrow$	3 (k) 4 (k) 2 (k) 6 (k)
1 (k)	$Z_{k+1}(n \times b) \rightarrow$	2 (k)	$\rho_{k+1}(b \times b) \rightarrow$ $AZ_{k+1}(n \times b) \rightarrow$ $\rho_{k+1}(b \times b) \rightarrow$	5 (k) 1 (k+1) 1 (k+1)
4 (k-1) 1 (k)	$V_k^T(2b \times 2b) \rightarrow$ $\nu_{k+1}(b \times b) \rightarrow$	3 (k)	$\overline{L}_{k+1,k+1}(b \times b) \rightarrow$ $L_{k,k-2}(b \times b) \rightarrow$	5 (k) 7 (k)
1 (k) 5 (k-1)	$\nu_{k+1}(b \times b) \rightarrow$ $\overline{L}_{k,k}(b \times b) \rightarrow$	4 (k)	$V_{k+1}^T(2b \times 2b) \rightarrow$ $V_{k+1}^T(2b \times 2b) \rightarrow$ $L_{k,k}^{-1}(b \times b) \rightarrow$	5 (k) 6 (k) 7 (k)
2 (k) 3 (k) 4 (k)	$\rho_{k+1}(b \times b) \rightarrow$ $\overline{L}_{k+1,k}(b \times b) \rightarrow$ $V_{k+1}^T(2b \times 2b) \rightarrow$	5 (k)	$L_{k,k-1}(b \times b) \rightarrow$ $\overline{L}_{k+1,k+1}(b \times b) \rightarrow$	7 (k) 4 (k+1)
1 (k) 4 (k)	$Z_{k+1}(n \times b) \rightarrow$ $V_{k+1}^T(2b \times 2b) \rightarrow$	6 (k)	$W_k(n \times b) \rightarrow$	8 (k)
3 (k-1) 5 (k-1) 4 (k)	$L_{k,k-2}(b \times b) \rightarrow$ $L_{k,k-1}(b \times b) \rightarrow$ $L_{k,k}^{-1}(b \times b) \rightarrow$	7 (k)	$\psi_k(b \times b) \rightarrow$	8 (k)
6 (k) 7 (k)	$W_k(n \times b) \rightarrow$ $\psi_k(b \times b) \rightarrow$	8 (k)		

Tabella 1.2: Flusso dei dati per ogni task.

vengono distribuiti in modo tale che le comunicazioni corrispondenti ad archi di peso maggiore utilizzano reti con migliori prestazioni.

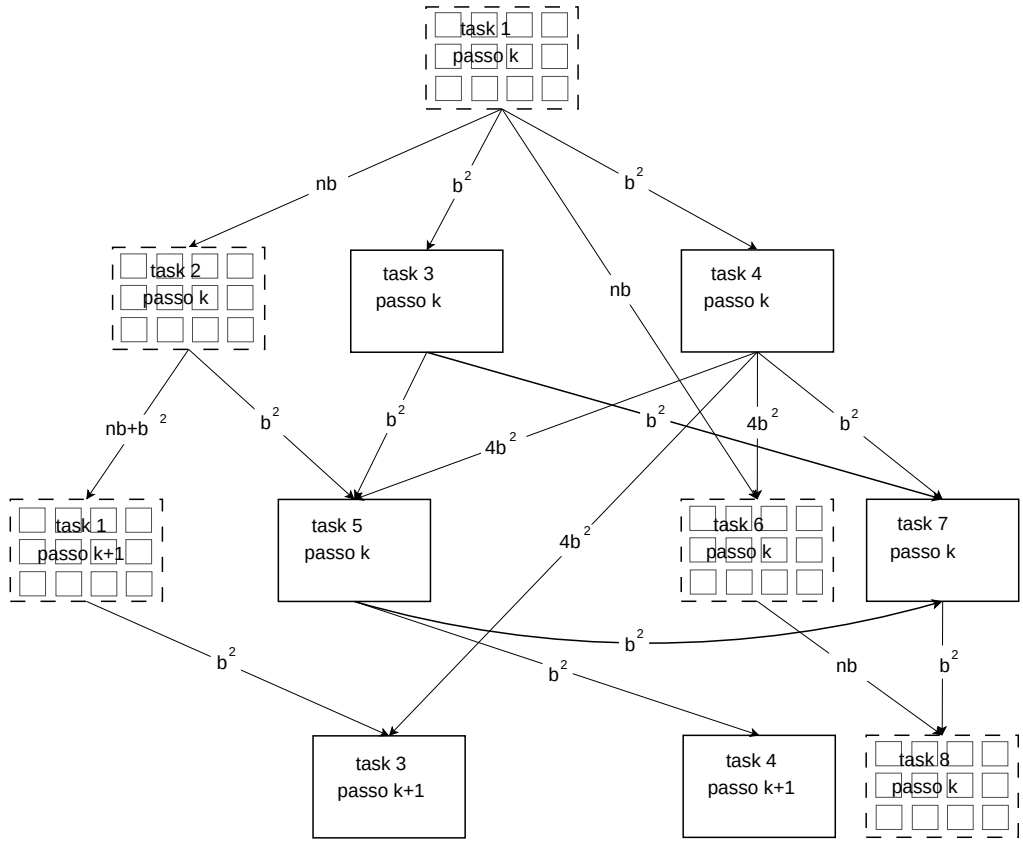

Figura 1.3: Grafo pesato delle dipendenze dei task dell'algoritmo del BCG.

Ad esempio, considerando che la quantità di dati scambiati tra i task 1 e 2 ad ogni passo è maggiore rispetto a quella di qualsiasi altra coppia, i due task verranno eseguiti su risorse computazionali "vicine" (appartenenti alla stessa LAN o a diverse CPU di uno stesso calcolatore).

1.2.2 Parallelismo a grana fine

Il parallelismo a grana fine si realizza distribuendo il calcolo all'interno dei singoli task tra più processi.

I task più onerosi dal punto di vista computazionale sono il task 1 (ag-

giornamento delle matrici di Lanczos) e il il task 2 (prodotto tra la matrice del sistema e la matrice di Lanczos).

Se $b \ll n$ i successivi due task che hanno una complessità computazionale maggiore sono il task 6 (aggiornamento delle matrici ausiliarie W, $\overline{W}$) e il task 8 (aggiornamento dell'approssimazione della soluzione).

In questo caso, poichè le matrici L e V sono rispettivamente di dimensioni $b \times b \ll n \times b$ e $2b \times 2b \ll n \times b$, non effettuiamo ulteriori distribuzioni del calcolo all'interno dei task responsabili del loro aggiornamento (task 3, 4, 5, 7).

La distribuzione del calcolo all'interno dei task 1, 2, 6 e 8 si può ottenere attraverso la partizione delle matrici di Lanczos in blocchi di righe o in blocchi di colonne.

Si è scelto di utilizzare la partizione in blocchi di righe in quanto, in questo modo, solo la fattorizzazione QR all'interno del task 1 e il prodotto AZ_k e il calcolo di ρ_{k+1} all'interno del task 2, richiedono comunicazioni tra i processi.

I processi di uno stesso task vengono eseguiti sui nodi di uno stesso calcolatore parallelo sfruttando per le comunicazioni (intra-task) reti di interconnessione veloci.

Nei successivi paragrafi verrà descritto il parallelismo a grana fine all'interno dei task 1, 2, 6 e 8.

Task 1

Nel task 1 viene calcolata la matrice

$$\widetilde{Z}_{k+1} = AZ_k - Z_k\rho_k - Z_{k-1}\nu_k^{-T}$$

e successivamente viene eseguita la fattorizzazione QR di $\widetilde{Z}_{k+1}$

$$\widetilde{Z}_{k+1} = Z_{k+1}\nu_{k+1}^{-1}.$$

Le matrici AZ_k, Z_k e Z_{k-1} sono distribuite per blocchi di righe tra i processi coinvolti nel task. Se P_1 è il numero di processi ed n è la dimensione del

Algoritmo 3 Algoritmo di Gram-Schmidt modificato in parallelo con una distribuzione dei dati per blocchi di righe

do $i = 1, b$
  Locale $\widetilde{Z}(:,i) \cdot \widetilde{Z}(:,i)$
  Somma Globale $dotp = \widetilde{Z}(:,i) \cdot \widetilde{Z}(:,i)$
  Locale $\nu(i,i) = 1/\sqrt{dotp}$
  Locale $Z(:,i) = \widetilde{Z}(:,i)/\nu(i,i)$
  if $(i < b)$
    Locale $\widetilde{Z}^T(:, i+1:b)Z(:,i)$
    Somma Globale $\nu(i, i+1:b) = \widetilde{Z}^T(:, i+1:b)Z(:,i)$
    Locale $\widetilde{Z}(:, i+1:b) = \widetilde{Z}(:, i+1:b) - \nu(i, i+1:b)Z(:,i)$
  endif
enddo

problema (numero di righe di AZ_k, Z_k e Z_{k-1}), il processo i, $i = 0, \ldots, P_1 - 1$, memorizza le righe $i * n/P_1, \ldots, (i+1) * n/P_1 - 1$ delle suddette matrici. Le matrici ρ_k e ν_k^{-T} sono invece memorizzate interamente in ognuno dei processi.

Il processo i calcola le righe $i * n/P_1, \ldots, (i+1) * n/P_1 - 1$ delle matrici $\widetilde{Z}_{k+1}$ e Z_{k+1} e la matrice ν_{k+1}^{-1}.

Per la fattorizzazione QR è stato utilizzato l'algoritmo di Gram-Schmidt modificato (algoritmo 3).

Nell'algoritmo 3 le operazioni con la notazione "Locale" vengono eseguite dal processo i utilizzando i dati locali (distribuiti): le righe $i * n/P_1, \ldots, (i+1) * n/P_1 - 1$ delle matrici $\widetilde{Z}_{k+1}$ e Z_{k+1}. La notazione "Somma Globale" rappresenta l'operazione di comunicazione collettiva che somma i dati distribuiti e distribuisce il risultato a tutti i processi. Al passo i, $i = 1, \ldots, b$, la dimensione del vettore coinvolto nell'operazione di "Somma Globale" è $b - i$. Supponendo che l'operazione venga eseguita secondo uno schema di comunicazione ad albero la complessità delle comunicazioni per ogni processo è $O(b^2 log_2 p)$ [43].

In generale un'unica applicazione dell'algoritmo 3 può non bastare a produrre una matrice Z_{k+1} sufficientemente ortogonale ($\left\| Z_{k+1}^T Z_{k+1} - I \right\|$ trascu-

rabile)[1]. Per questo motivo la fattorizzazione viene eseguita in modo iterativo e ripetuta, se necessario, fino a $2 \times b$ volte [27].

Task 2

Nel task 2 viene calcolata la matrice

$$AZ_{k+1}$$

e successivamente la matrice

$$\rho_{k+1} = Z_{k+1}^T AZ_{k+1}.$$

Le matrici A e Z_{k+1} sono distribuite per blocchi di righe tra i processi coinvolti nel task.

Se P_2 è il numero di processi, il processo i calcola le righe $i*n/P_2, \ldots, (i+1)*n/P_2 - 1$ della matrice AZ_{k+1} e la matrice ρ_{k+1}, tramite operazione di riduzione, come somma dei prodotti locali $Z_{k+1}^T AZ_{k+1}$. La dimensione del vettore coinvolto in questa "Somma Globale" è $b \times b$.

Supponendo che le operazioni di comunicazione collettive vengano eseguite secondo uno schema di comunicazione ad albero la complessità delle comunicazioni per ogni processo è $O(nblog_2p)$ per il calcolo di AZ_{k+1} e $O(b^2log_2p)$ per il il calcolo di ρ_{k+1} [43].

Task 6

Nel task 6 viene calcolata la matrice

$$W_k = [\overline{W}_k, Z_{k+1}]V_{k+1}^T(1 : 2 \times b, 1 : b)$$

e successivamente

$$\overline{W}_{k+1} = [\overline{W}_k, Z_{k+1}]V_{k+1}^T(1 : 2 \times b, b + 1 : 2 \times b).$$

[1] causa errori di round-off.

Le matrici $\overline{W}_k$ e Z_{k+1} sono distribuite per blocchi di righe tra i processi coinvolti nel task. La matrice V_{k+1} è invece memorizzata interamente in ognuno dei processi.

Se P_6 è il numero di processi, il processo i calcola le righe $i*n/P_6, \ldots, (i+1)*n/P_6 - 1$ delle matrici W_k e $\overline{W}_{k+1}$.

Task 8

Nel task 8 viene calcolata la soluzione al passo k

$$\hat{X}_k = \hat{X}_{k-1} + W_k \psi_k.$$

Le matrici $\hat{X}_{k-1}$ e W_k sono distribuite per blocchi di righe tra i processi coinvolti nel task. La matrice ψ_k è invece memorizzata interamente in ognuno dei processi.

Se P_8 è il numero di processi, Il processo i calcola le righe $i*n/P_8, \ldots, (i+1)*n/P_8 - 1$ della matrice $\hat{X}_k$.

Capitolo 2

Un sistema per la gestione dell'esecuzione di applicazioni MPI: MPI jobs management system

I job[1] che eseguono le applicazioni parallele su Grid consistono di più componenti, denominati subjob, eseguiti in parallelo su uno o più processori di uno o più calcolatori presso differenti siti. Ci si può riferire a queste applicazioni come ad applicazioni parallele multi-sito.

L'esecuzione dei job (e quindi l'esecuzione concorrente dei subjob) dell'applicazione del BCG (Capitolo 1) e di una qualsiasi altra applicazione parallela multi-sito, in ambiente Grid, richiede essenzialmente la co-allocazione, cioè l'allocazione simultanea di più risorse.

Inoltre è auspicabile l'esistenza di un sistema che consenta la sottomissione di tali applicazioni, fornendo funzionalità di selezione delle risorse (brokering) e di scheduling dei job, e che sia tollerante ai fallimenti (*fault-tolerant*).

MPICH-G2 (vedi paragrafo A.2) utilizza per la co-allocazione la compo-

[1]Uno o più processi corrispondenti alla richiesta di esecuzione di una computazione.

nente Globus chiamata Dynamically Updated Request Online Co-allocator (DUROC) (vedi paragrafo A.1.4).

DUROC, pur asservendo allo scopo principale di consentire la co-allocazione per l'esecuzione di job paralleli "multi-componente", presenta alcune limitazioni: non fornisce strumenti per la selezione delle risorse al momento della sottomissione dei job, richiedendo che venga esplicitamente specificata ogni risorsa destinata all'esecuzione dei singoli subjob, nè è in grado di gestire situazioni di fallimenti.

Ad esempio, se nel momento in cui vengono sottomessi i job le risorse specificate non sono disponibili, DUROC, potrebbe attendere che le stesse divengano di nuovo disponibili per un periodo di tempo imprecisato. Oppure, nell'eventualità in cui l'esecuzione del job fallisca è richiesto l'intervento diretto dell'utente.

Ad oggi non sono stati sviluppati e implementati molti sistemi di selezione, co-allocazione e monitoraggio delle risorse per l'esecuzione di job paralleli in ambienti multi-cluster e Grid.

La mancanza di tali sistemi limita di fatto lo sviluppo di applicazioni parallele multi-sito[2].

Un sistema che è in grado di consentire l'allocazione di risorse appartenenti a differenti domini amministrativi, per un singolo job, è Condor (vedi paragrafo A.3.1) con il suo DAG Manager. Il DAG Manager di Condor prende in input job descritti sotto forma di grafi orientati aciclici (Directed Acyclic Graphs, DAG), schedulando di volta in volta i task del grafo in funzione delle dipendenze. Un nuovo task viene inserito nella coda dei job non appena tutti i task da cui dipende sono stati eseguiti.

[2]L'altro motivo per cui vengono sviluppate poche applicazioni di questo tipo è da ricercarsi nel fatto che gli sviluppatori e utenti temono che il tempo di computazione sia notevolmente penalizzato dalle limitate ampiezza di banda e latenza delle reti WAN.

Questo overhead è tanto maggiore quanto maggiori sono le comunicazione su WAN tra i subjob dell'applicazione multi-componente. A questo punto è chiaro che al migliorare delle prestazioni delle reti WAN, quell'overhead tenderà a ridursi e, conseguentemente, verranno sviluppate più applicazioni parallele multi-sito.

In ogni caso, in Condor, non è implementato nessun meccanismo di co-allocazione che consenta l'acquisizione simultanea di più risorse per l'esecuzione di job paralleli multi-componente; tuttavia Condor-G (vedi paragrafo A.3.2) presenta le funzionalità di scheduling e di tolleranza ai fallimenti auspicate.

È possibile pertanto utilizzare servizi e sistemi Grid già esistenti (DUROC per la co-allocazione delle risorse, Condor-G per lo scheduling dei job, il Servizio di Informazioni di Globus per il monitoraggio delle risorse), e integrarli con nuove componenti, progettate e sviluppate ex-novo per fornire funzionalità supplementari.

Per consentire la sottomissione dei job paralleli multi-componente attraverso Condor-G è stato quindi realizzato il *Coordinator*. Tale servizio interagisce con lo Scheduler di Condor-G, sfruttando le funzionalità del sistema Condor-G, e utilizza i servizi forniti da DUROC, per gestire la sincronizzazione dei subjob.

La funzionalità di brokering è svolta dalla componente che è stata chiamata *GangMatchMaker* (un vero e proprio Resource Broker) combinando informazioni ottenute dall'*Advertiser* attraverso interrogazioni al Servizio di Informazioni (MDS) del Globus Toolkit (vedi paragrafo A.1.3). L'allocazione presso le diverse risorse remote è gestita dai GateKeeper (del Globus Toolkit, vedi paragrafo A.1.4) locali.

In definitiva è stato sviluppato un sistema per la gestione dell'esecuzione di applicazioni parallele (secondo le specifiche MPI) sui cluster distribuiti di un ambiente Grid (*MPI jobs management system*).

Questo sistema consente di schedulare ed eseguire i subjob paralleli sui calcolatori più appropriati ad eseguirli, tenendo conto sia dei requisiti e preferenze dei job, sia delle caratteristiche, stato e preferenze delle risorse. Requisiti e preferenze dovranno riflettere tra l'altro le esigenze dei diversi subjob paralleli in termini di complessità computazionale e complessità delle comunicazioni, in modo da ottimizzare il mapping con le risorse disponibili.

Problemi affrontati nello sviluppo di questo sistema sono stati il brokering

delle risorse, lo scheduling dei job e la co-allocazione delle risorse richieste per l'esecuzione.

L'architettura di tale sistema è rappresentata in figura 2.1.

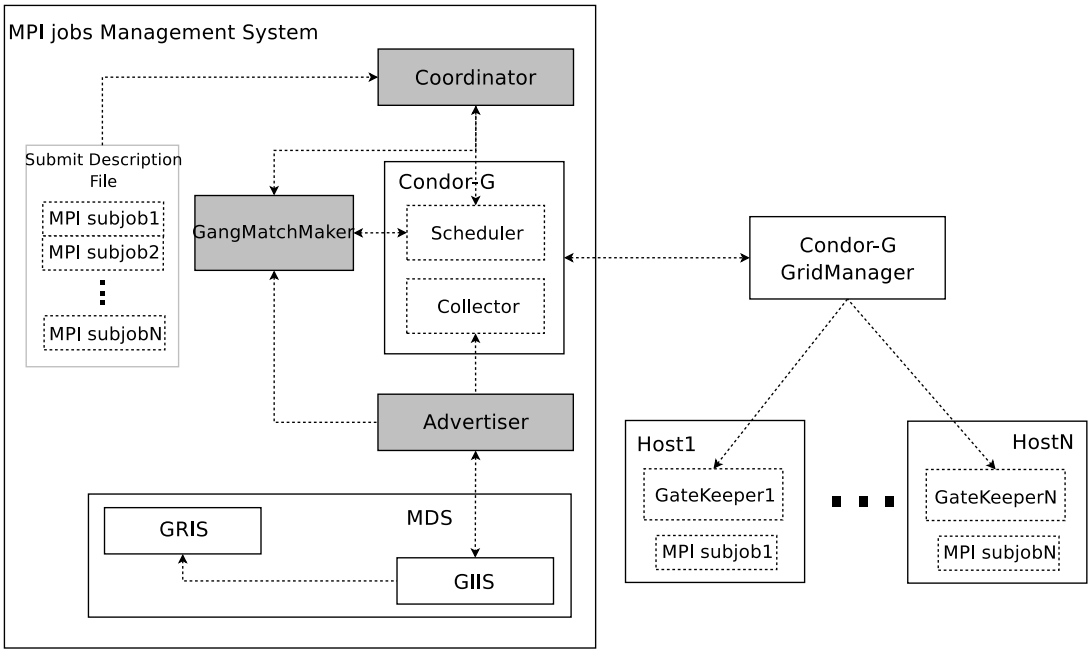

Figura 2.1: MPI jobs management system - un job parallelo, descritto mediante un submit description file, viene sottomesso al sistema Condor-G attraverso il *Coordinator*. Quest'ultimo interagisce con lo Scheduler Condor-G per la sottomissione del job e con il *GangMatchMaker* per la selezione dei calcolatori più appropriati per l'esecuzione del job. L'*Advertiser* comunica le risorse Grid disponibili sia al Collector di Condor per la fase di sottomissione, sia al *GangMatchMaker* per essere prese in considerazione nell'operazione di brokering.

2.1 Il Coordinator

Il Coordinator consente di sottomettere i job di un'applicazione parallela (secondo le specifiche MPI) attraverso Condor-G sulle risorse Grid più appropriate ad eseguirli.

Esso consente la co-allocazione dei subjob appartenenti all'applicazione parallela interagendo con Condor-G, l'Advertiser e il GangMatchMaker, seguendo la seguente procedura a sette passi:

- Nel primo passo sottomette tutti i subjob allo Scheduler Condor-G posizionandoli nella coda in stato di *hold*. I job sottomessi in questo stato rimangono nella coda finchè non vengono rilasciati attraverso il comando **condor_release**.
- Nel secondo passo fa partire e inizializza i servizi di DUROC.
- Nel terzo passo modifica le variabili di ambiente dei subjob in modo che DUROC possa gestirne la sincronizzazione: i subjob sono bloccati da una barriera DUROC.
- Nel quarto passo crea una ClassAd che rappresenta requisiti e preferenze di tutti i subjob, sottomettendola al GangMatchMaker, e richiede all'Advertiser l'aggiornamento delle ClassAd che descrivono le risorse. Il GangMatchMaker, implementando il modello del *Gangmatching* [49], utilizza la ClassAd che descrive requisiti e preferenze dei subjob e le ClassAd che descrivono le risorse per trovare i calcolatori più appropriati ad eseguire i subjob. Il Coordinator riceve comunicazione di tali calcolatori.
- Nel quinto passo modifica per ogni subjob l'attributo *GlobusResource* assegnandogli l'URL del GateKeeper della risorsa individuata dal GangMatchMaker per quel subjob.
- Nel sesto passo rilascia i subjob dalla coda di Condor-G.

- Nel settimo passo rilascia la barriera DUROC: questo avviene solo quando tutti i subjob sono stati assegnati ad un risorsa.

Dopo questa sincronizzazione viene avviata l'esecuzione di tutti i subjob, monitorata fino al completamento.

Per ogni job parallelo (secondo le specifiche MPI) multi-componente sottomesso al Coordinator è necessario specificare il numero dei subjob e la loro dimensione, cioè il numero di processori da utilizzare per ogni subjob.

La descrizione dei job avviene attraverso un submit description file (vedi paragrafo A.3.1) multi-componente che contiene un submit description file per ogni subjob.

Per avvalersi del servizio di brokering del GangMatchMaker, e consentire l'allocazione dinamica dei subjob in base ai requisiti specificati, nei submit description file non va indicato nessun *globusscheduler* specifico, mentre vanno specificati i *Requirements*.

Se invece si vuole che per l'esecuzione di un determinato subjob venga utilizzata una risorsa specifica allora per quel subjob andrà indicato un *globusscheduler* specifico includendo la riga:

```
globusscheduler = beocomp.dma.unina.it/jobmanager
```

2.2 Il GangMatchMaker (Resource Broker) e l'Advertiser

Il GangMatchMaker "trova" i calcolatori che meglio corrispondono ai requisiti e preferenze dei subjob.

Tale servizio di brokering è implementato utilizzando il modello del Gangmatching, un'estensione del modello del Matchmaking di Condor-G, e combinando le informazioni sulle caratteristiche e lo stato delle risorse.

Il modello del Gangmatching consente di superare una pesante limitazione del modello del Matchmaking: tale modello si limita a individuare un unico (possibile) matching tra una singola richiesta di esecuzione e le risorse

disponibili; il risultato sarà un'unica risorsa considerata compatibile con la richiesta di esecuzione. Il modello del Matchmaking è pertanto puramente bilaterale.

Il Gangmatching consente invece di trovare i matching tra un insieme di richieste di esecuzione e le risorse disponibili, tenendo conto contemporaneamente dei requisiti e preferenze di tutte le richieste e consentendo ad ogni risorsa la capacità di matching con una sola richiesta. Il risultato sarà rappresentato dalle diverse risorse compatibili con i diversi subjob.

Per ogni subjob, identificato dal proprio $ClusterID$, il GangMatchMaker comunica al Coordinator il JobManager ($gatekeeper\_url$) del calcolatore individuato per l'esecuzione.

Le informazioni sulle caratteristiche e sullo stato corrente delle risorse sono ottenute interrogando il Servizio di Informazioni del Globus Toolkit (vedi paragrafo A.1.3). Allo scopo di fornire una rappresentazione delle risorse adeguata alle necessità del sistema è stato progettato e implementato uno schema *ad hoc* per la pubblicazione delle informazioni relative ai calcolatori paralleli.

Requisiti e preferenze dei subjob sono espressi attraverso gli attributi *Requirements* e *Rank* nel submit description file utilizzato per la sottomissione.

Affinchè il sistema Condor sia a conoscenza delle risorse disponibili per la sottomissione e il GangMatchMaker sia in grado di trovare le compatibilità tra richieste di subjob e risorse offerte è necessario che i siti Grid disponibili siano "pubblicizzati".

L'Advertiser comunica le risorse disponibili sia al Collector di Condor per la fase di sottomissione, sia al GangMatchMaker per essere prese in considerazione nell'operazione di Gangmatching.

Le caratteristiche delle risorse sono espresse sotto forma di ClassAd. A tale scopo l'Advertiser crea delle ClassAd per ogni risorsa utilizzando informazioni statiche e informazioni dinamiche, ottenute interrogando il Servizio di Informazioni di Globus.

2.2.1 Il linguaggio Classified Advertisements (ClassAds) e il Matchmaking

Il linguaggio Classified Advertisements è un linguaggio semistrutturato che consente di rappresentare le caratteristiche di servizi arbitrari (nel caso specifico richieste di job e calcolatori) e di specificare i requisiti richiesti per la loro allocazione.

Una ClassAd è una lista di corrispondenze tra *nomi di attributi* e *espressioni*.

Ad esempio in figura 2.2 è mostrata una ClassAd che rappresenta una workstation, mentre in figura 2.3 è mostrata una ClassAd che descrive un job.

```
[
    MyType = ‘‘Machine’’;
    TargetType = ‘‘Job’’;
    Name = ‘‘lamu2-9.cps.na.cnr.it’’;
    Machine = ‘‘lamu2-9.cps.na.cnr.it’’;
    Rank = 0.000000;
    VirtualMemory = 516492;
    Disk = 252528;
    KeyboardIdle = 8848;
    Memory = 249;
    Cpus = 1;
    Arch = ‘‘INTEL’’;
    OpSys = ‘‘LINUX’’;
    TotalVirtualMemory = 516492;
    TotalDisk = 252528;
    KFlops = 82717;
    Mips = 492;
    State = ‘‘Unclaimed’’;
    Activity = ‘‘Idle’’;
    Friends = {‘‘oliva’’, ‘‘frangreg’’};
    Requirements = member(other.name, Friends);
]
```

Figura 2.2: Esempio di ClassAd che descrive una workstation.

```
[
   MyType = ``Job'';
   TargetType = ``Machine'';
   QDate = 1123783047;
   CompletionDate = 0;
   Owner = ``gregoretti'';
   Iwd = ``/home/gregoretti/'';
   Cmd = ``/home/gregoretti/sim.g2'';
   WantRemoteSyscalls = FALSE;
   WantCheckpoint = FALSE
   DiskUsage = 0;
   Args = ``'';
   Requirements =
        (other.Type ==``Machine'')
        && (other.Arch == ``INTEL'')
        && (other.OpSys == ``LINUX'')
        && (other.Disk >= DiskUsage)
        && (other.Memory > Memory);
]
```

Figura 2.3: Esempio di ClassAd che descrive un job.

Le espressioni possono essere semplici valori costanti come numeri interi o reali, valori booleani (**true** o **false**), date, stringhe di caratteri, i valori speciali **error** e **undefined**, oppure possono essere espressioni composite costruite attraverso operatori applicati a uno o più operandi.

Tali operatori includono gli usuali operatori aritmetici, logici e di confronto (come $+$, && e $\leq$), costruttori di liste ($\{espressione_1, \ldots, espressione_n\}$) e di record ($[nome_1 = espressione_1, \ldots, nome_n = espressione_n]$) e funzioni integrate (come $member(constx, stringl)$ che ritorna un booleano: se x non è un valore costante o l non è una lista, allora il risultato è **error**; altrimenti se uno degli elementi di l è uguale a x secondo l'operatore $==$, allora il risultato è **true**, altrimenti è **false**).

Per trovare i matching tra le richieste dei job e le offerte di risorse, il modello del Matchmaking, opera in un ambiente di valutazione nel quale ogni ClassAd può accedere agli attributi delle altre ClassAd. Il riferimento della

forma $self.attributo$ si riferisce ad un attributo della ClassAd che contiene tale riferimento. Il riferimento $other.attributo$, invece, si riferisce all'attributo di un'altra ClassAd. Se non è specificato nè $self$ nè $other$ il meccanismo di valutazione assume il prefisso $self$.

Il protocollo e l'algoritmo di Matchmaking attribuiscono un significato speciale alle parole chiave *Requirements*, *Rank* e *other*.

Se i *Requirements* di due ClassAd sono valutati entrambi **true** allora le le due ClassAd sono considerate compatibili (è stato trovato un "match").

A questo punto viene utilizzato l'attributo *Rank* per effettuare una scelta all'interno di tutti i possibili match. Tra tutte le ClassAd di offerta di risorse compatibili con la ClassAd di richiesta di un job, il Matchmaking sceglie quella con il più alto valore per l'attributo *Rank*. Quest'ultimo viene valutato congiuntamente al valore di *Rank* della ClassAd di richiesta.

Il riferimento ad un attributo inesistente viene valutato con la costante **undefined**.

L'algoritmo di matchmaking tratta il valore **undefined** come **false**, in altre parole il matching fallisce se l'attributo *Requirements* viene valutato **undefined**.

2.2.2 Schema per la definizione delle informazioni relative ai cluster

In questo paragrafo viene descritta l'estensione realizzata a partire da uno schema utilizzato dall'MDS per la pubblicazioni delle informazioni relative ai cluster.

In un'applicazione parallela multi-sito, i diversi subjob paralleli possono avere un diverso costo delle comunicazioni e, pertanto, possono avere richieste diverse rispetto alle prestazioni della rete interna dei cluster sui quali dovranno essere eseguiti.

A tale scopo è auspicabile che il sistema informativo utilizzato dall'in-

frastruttura Grid pubblichi, consentendone l'accesso, le informazioni relative alle prestazioni succitate.

Il *GLUE schema* è uno degli schemi adottati dal Servizio di Informazioni di Globus per la pubblicazione delle informazioni (vedi paragrafo A.1.3). Le componenti fondamentali di tale schema sono:

- *Computing Element* che rappresentano le code dei sistemi di scheduling;
- *Cluster* che rappresentano un raggruppamento di subcluster o nodi;
- *Subcluster* che rappresentano un gruppo di calcolatori con caratteristiche di base comuni.

Uno schema alternativo utilizzato per la rappresentazione di informazioni su calcolatori di tipo cluster deriva dal *NorduGrid Information System* [39] dove, a differenza di quanto accade per il GLUE schema, le caratteristiche delle code del sistema di sottomissione sono attributi del cluster.

Entrambi gli schemi succitati non prevedono la pubblicazioni di informazioni sul sottosistema di rete mediante il quale sono interconnessi i nodi di un cluster con riferimento alle sue caratteristiche e prestazioni. Tali informazioni risultano di notevole importanza nella selezione della risorsa più appropriata per l'esecuzione di un'applicazione parallela o di un subjob parallelo.

Per superare questa limitazione si è scelto di ampliare uno dei due schemi con informazioni dettagliate sul sottosistema di rete. Si è utilizzato come base lo schema NorduGrid in quanto, a differenza del GLUE Schema, esso fornisce esclusivamente una visione di insieme del cluster, senza dettagli sui singoli nodi. Tali dettagli infatti risultano maggiormente utili nella selezione di risorse per l'esecuzione di applicazioni sequenziali. L'estensione dello schema NorduGrid è mostrata in figura 2.4. In tale figura è rappresentato il record LDIF di un cluster.

Gli attributi in corsivo sono quelli introdotti per il sistema proposto.

I valori per gli attributi

cluster-network-mpi-latency, *cluster-network-mpi-bandwidth*, *cluster-network-tcp-latency* e *cluster-network-tcp-bandwidth*,

```
dn: nordugrid-cluster-name=beocomp.dma.unina.it,Mds-Vo-name=local,o=grid
objectClass: nordugrid-cluster
objectClass: Mds
nordugrid-cluster-name: beocomp.dma.unina.it
nordugrid-cluster-aliasname: Beocomp
nordugrid-cluster-comment: Hello there!
nordugrid-cluster-owner: ICAR CNR Sezione di Napoli
nordugrid-cluster-location: Dipartimento di Matematica e Applicazioni
nordugrid-cluster-issuerca: /O=ICAR-NA/CN=ICAR-NA Certification Authority
nordugrid-cluster-contactstring: beocomp.dma.unina.it:2122
cluster-network-name: Fast Ethernet
cluster-network-vendor: 3Com
cluster-network-model: Super Stack
cluster-network-version: II
cluster-network-mpi-latency: 65
cluster-network-mpi-bandwidth: 82
cluster-network-tcp-latency: 34
cluster-network-tcp-bandwidth: 90
cluster-cpus-specint: 17
cluster-cpus-specfloat: 13
nordugrid-cluster-support: almerico.murli@dma.unina.it
nordugrid-cluster-lrms-type: OpenPBS
nordugrid-cluster-lrms-version: 2.3
nordugrid-cluster-lrms-config: FIFO scheduler, single job per processor
nordugrid-cluster-architecture: i686
nordugrid-cluster-opsys: LINUX
nordugrid-cluster-opsys-version: 2.4.20-20.7
nordugrid-cluster-homogeneity: True
nordugrid-cluster-nodecpu: Pentium II (Deschutes)
nordugrid-cluster-nodememory: 256
nordugrid-cluster-nodeaccess: inbound
nordugrid-cluster-nodeaccess: outbound
nordugrid-cluster-totalcpus: 14
nordugrid-cluster-cpudistribution: 1cpu:14
nordugrid-cluster-usedcpus: 0
nordugrid-cluster-queuedjobs: 0
nordugrid-cluster-totaljobs: 0
nordugrid-cluster-sessiondir-free: 0
nordugrid-cluster-sessiondir-total: 0
Mds-validfrom: 200509161427562
Mds-validto: 200509161428262
```

Figura 2.4: Schema NorduGrid esteso.

per i cluster del sistema Grid di supporto (vedi appendice B), sono stati ottenuti misurando le prestazioni delle reti interne con il programma di benchmark NetPIPE [52]. Tale programma è stato utilizzato per misurare le prestazioni della sottostruttura di rete con riferimento al protocollo TCP e all'implementazione MPI presente sul cluster.

In figura 2.5 è mostrata la latenza unidirezionale (*one-way*) per MPI per pacchetti di piccole dimensioni per i tre cluster del sistema Grid di supporto, denominati Beocomp, Altair e Vega.

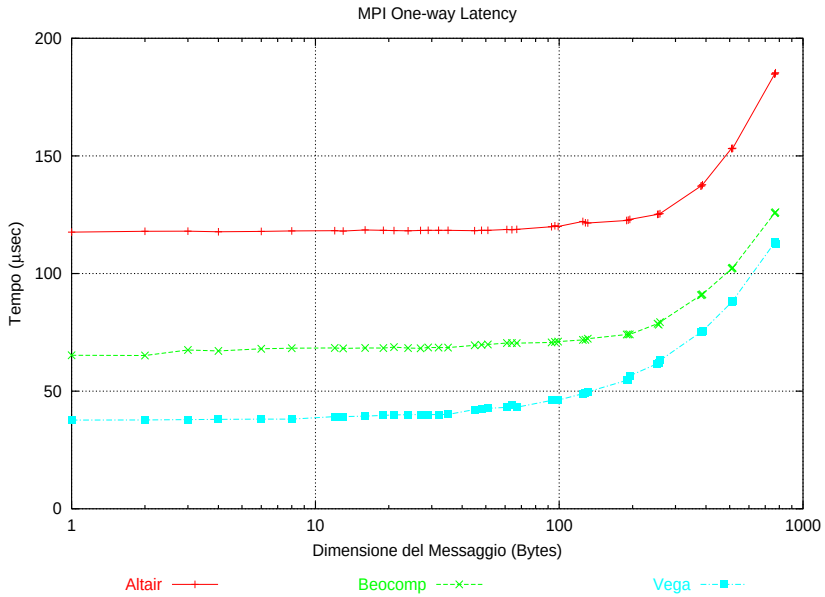

Figura 2.5: Latenza One-way MPI - La latenza one-way è metà del tempo di round trip.

La latenza di startup per i suddetti cluster è rispettivamente di 65, 117 e 38 μsec.

In figura 2.6 è mostrata l'ampiezza di banda unidirezionale (*one-way*) per MPI per gli stessi cluster per dimensioni dei pacchetti fino a 8MB.

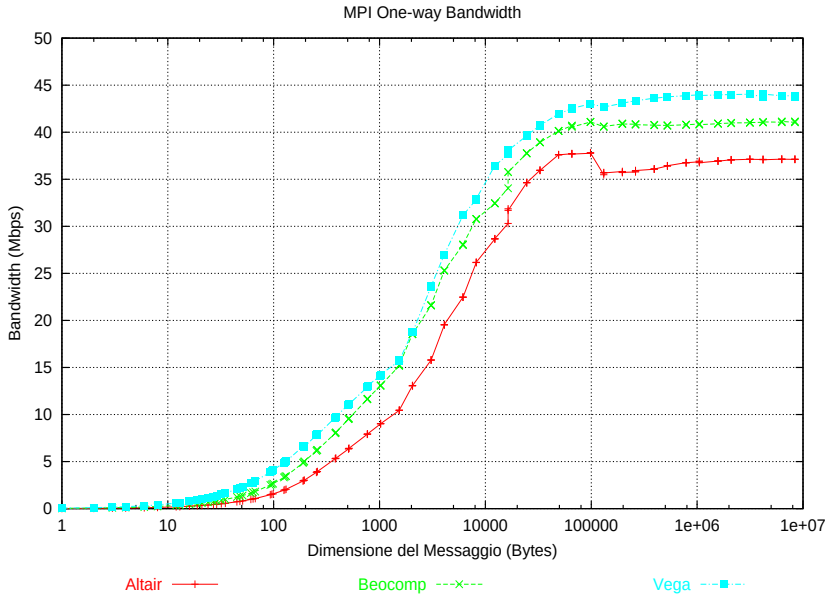

Figura 2.6: Ampiezza di Banda One-Way MPI.

2.2.3 L'Advertiser

L'Advertiser (vedi figura 2.1) ha lo scopo di "informare" sulle risorse Grid disponibili il Collector di Condor (per la fase di sottomissione) ed il GangMatchMaker (per il matching con le richieste di job).

Periodicamente l'Advertiser comunica al Collector e al GangMatchMaker una ClassAd per ogni risorsa Grid disponibile. Tale ClassAd è costituita da una lista minima di informazioni statiche comuni a tutte le risorse:

```
MyType                = ``Machine''
TargetType            = ``Job''
Name                  = ``Nome_Gatekeeper''
gatekeeper_url        = ``nome.dma.unina.it/jobmanager''
Rank                  = 0.000000
CurrentRank           = 0.000000
UpdateSequenceNumber  = 4
CurMatches            = 0
```

UpdateSequencenumber è un intero positivo incrementato ogni volta che

il sito viene pubblicizzato. Infatti il Collector scarta la ClassAd di un sito dopo un ben definito intervallo di tempo dall'ultimo aggiornamento. Tale intervallo di tempo può essere specificato attraverso la macro

$$CLASSAD\_LIFETIME$$

definita nel file di configurazione di Condor. Il suo valore di default è 900 (quindici minuti). Alternativamente, l'intervallo di tempo dopo il quale la ClassAd di un sito viene considerata "scaduta", può essere specificato attraverso un attributo definito all'interno della ClassAd stessa.

La ClassAd viene completata con informazioni statiche e dinamiche più dettagliate attraverso interrogazioni (LDAP query) al GIIS (vedi paragrafo A.1.3) relativo ai siti che si intende pubblicizzare. Ad ogni interrogazione l'Advertiser riceve dati dal GIIS in formato LDIF secondo gli schemi GLUE e NorduGrid esteso e converte tali informazioni in una lista di ClassAd.

La ClassAd viene aggiornata e comunicata al Collector (tramite il comando **condor_advertise**) periodicamente e ogni volta che un job parallelo multi-componente è sottomesso, tramite il Coordinator, allo Scheduler Condor-G.

La stessa ClassAd viene comunicata al GangMatchMaker ad ogni sottomissione di un job parallelo.

2.2.4 Il Gangmatching

La co-allocazione in ambiente Grid impone la necessità di un modello di matchmaking multilaterale in grado di trovare le compatibilità, attraverso un'unica operazione atomica, all'interno di un'aggregazione di ClassAd dipendenti tra loro.

Il modello del Gangmatching è una soluzione al problema del matchmaking multilaterale che soddisfa il seguente requisito: le caratteristiche dei candidati (richieste e offerte) e le loro interdipendenze sono definite esclusivamente dai candidati stessi.

Mediante tale modello vengono create "gang" di ClassAd "unendo" tra loro ClassAd singole attraverso un'operazione di matching.

Intuitivamente il Gangmatching estende il matchmaking bilaterale rimpiazzando l'obbligo di un singolo (implicito) match bilaterale con la richiesta di un'esplicita lista di match bilaterali, fornendo ad ogni ClassAd la capacità supplementare di accedere alle informazioni relative ai match bilaterali che coinvolgono ClassAd differenti.

L'operazione di matching avviene tra *port* di ClassAd differenti. L'astrazione dei *port* ha differenti funzioni:

I *port* fungono da interfaccia di matchmaking consentendo ai candidati di accedere alle informazioni indipendentemente dal modo con cui siano state generate (come valori costanti o informazioni provenienti da caratteristiche di altri candidati nella "gang").

L'astrazione dei *port* separa gli spazi dei nomi dei diversi tipi di candidati e permette di accedere alle caratteristiche dei candidati da altri match.

L'astrazione dei *port* infine impone una struttura sulla "gang" aggregata che semplifica la definizione e l'implementazione degli algoritmi.

Viene ora presentato il modello del Gangmatching trattando il problema specifico della co-allocazione dei subjob che compongono un'applicazione parallela.

In questo contesto, è necessario trovare i matching tra le richieste di esecuzione dei singoli subjob e le offerte di risorse disponibili, tenendo conto simultaneamente dei requisiti e preferenze di tutti i subjob e delle loro interdipendenze, e consentendo che ad ogni subjob possa essere assegnata una risorsa differente.

Rappresentazione delle ClassAd nel problema di co-allocazione di un job multi-componente

Un esempio di ClassAd che rappresenta la richiesta di esecuzione di più subjob facenti parte di un'applicazione parallela è illustrata in figura 2.7.

```
[ Type = ‘‘Job’’;
  Ports = {
    [ Label=‘‘subjob1’’;
      Requirements = subjob1.type == ‘‘Machine’’ &&
                     subjob1.Arch == ‘‘i686’’ &&
                     subjob1.OpSys == ‘‘LINUX’’ &&
                     subjob1.ClusterNodeCount >= 17;
      Rank = 10000 / subjob1.ClusterNetMPILatency +
                10 * subjob1.ClusterCPUsSpecFloat;
    ],
    [ Label=‘‘subjob2’’;
      Requirements = subjob2.type == ‘‘Machine’’ &&
                     subjob2.Arch == ‘‘i686’’ &&
                     subjob2.OpSys == ‘‘LINUX’’ &&
                     subjob2.ClusterNodeCount >= 4;

    ],
    [ Label=‘‘subjob3’’;
      Subnet1 = subjob1.Subnet;
      Requirements = subjob3.type == ‘‘Machine’’ &&
                     subjob3.Arch == ‘‘i686’’ &&
                     subjob3.OpSys == ‘‘LINUX’’ &&
                     subjob3.ClusterNodeCount >= 14;
      Rank = 10 * subjob3.ClusterCPUsSpecFloat;
    ]
  }
]
```

Figura 2.7: Esempio di ClassAd che descrive un job parallelo multi-componente.

La caratteristica più rilevante di questo esempio è l'attributo *Ports*. I *port* di una ClassAd definiscono il numero e le caratteristiche delle richieste per le quali occorre trovare i matching affinchè l'intera ClassAd sia soddisfatta. Nel modello del Gangmatching l'algoritmo bilaterale di Matchmaking viene applicato tra i *port* di ClassAd diverse piuttosto che tra intere ClassAd.

Ogni *port* definisce una *Label* che sostituisce l'attributo *other* del Matchmaking bilaterale. Il campo di azione di una *label* si estende dal *port* in

cui è definita fino all'ultimo *port* della lista. In questo modo, le espressioni definite nel *port* con *label* "subjob2" possono fare riferimento alle espressioni definite nel *port* con *label* "subjob1", ma non viceversa.

Nell'esempio proposto i requisiti per i matching di ogni *port* non dipendono da nessun attributo definito in qualche altro *port*. Potrebbe invece essere richiesto che i *Requirements* di qualche *port* dipendano da qualche attributo definito in un altro *port* che lo precede nella lista. La relazione di dipendenza tra *port* è infatti limitata dal fatto che ogni *port* non può dipendere da nessun *port* che lo segue nella lista.

Nelle figure 2.8, 2.9, 2.10, sono illustrate tre ClassAd parziali che rappresentanti i cluster del sistema Grid di supporto.

```
[ MyType = ‘‘Machine’’;
  Name = ‘‘vega.na.icar.cnr.it’’;
  gatekeeper_url = ‘‘vega.na.icar.cnr.it:2122’’;
  Arch = ‘‘i686’’;
  OpSys = ‘‘LINUX’’;
  Subnet = ‘‘140.164.14’’
  ClusterNodeCount = 17;
  ClusterCPUType = ‘‘Intel(R) Pentium(R) 4 CPU 1500MHz’’;
  ClusterNetMPIBandwidth = ‘‘88’’;
  ClusterCPUsSpecFloat = ‘‘558’’;
  ClusterCPUsSpecInt = ‘‘535’’;
  ClusterNetiMPILatency = ‘‘38’’;
  Ports = {
    [ Label = ‘‘subjob’’
    ]
  }
]
```

Figura 2.8: ClassAd parziale che descrive il cluster Vega.

Le ClassAd che rappresentano le risorse differiscono dalla corrispondenti ClassAd del modello bilaterale per la presenza dei *port*. Tale presenza impone alla ClassAd della risorsa l'obbligo che il matching possa avvenire con una sola entità, cioè con un unico *port* della ClassAd di richiesta di esecuzione.

Nelle ClassAd di esempio non sono definiti *Requirements*. Quest'ultimi

```
[ MyType = ‘‘Machine’’;
  Name=‘‘altair.dma.unina.it’’;
  gatekeeper_url=‘‘altair.dma.unina.it:2122’’;
  Arch=‘‘i686’’;
  OpSys=‘‘LINUX’’;
  Subnet = ‘‘192.167.11’’
  ClusterNodeCount=11;
  ClusterCPUType=‘‘Pentium Pro’’;
  ClusterNetMPIBandwidth=‘‘74’’;
  ClusterCPUsSpecFloat=‘‘6’’;
  ClusterCPUsSpecInt=‘‘8’’;
  ClusterNetMPILatency=‘‘117’’;
  Ports = {
    [ Label = ‘‘subjob’’
    ]
  }
]
```

Figura 2.9: ClassAd parziale che descrive il cluster Altair.

```
[ MyType = ‘‘Machine’’;
  Name=‘‘beocomp.dma.unina.it’’;
  gatekeeper_url=‘‘beocomp.dma.unina.it:2122’’;
  Arch=‘‘i686’’;
  OpSys=‘‘LINUX’’;
  Subnet = ‘‘192.167.11’’
  ClusterNodeCount=14;
  ClusterCPUType=‘‘Pentium II (Deschutes)’’;
  ClusterNetMPIBandwidth=‘‘82’’;
  ClusterCPUsSpecFloat=‘‘13’’;
  ClusterCPUsSpecInt=‘‘17’’;
  ClusterNetMPILatency=‘‘65’’;
  Ports = {
    [ Label = ‘‘subjob’’
    ]
  }
]
```

Figura 2.10: ClassAd parziale che descrive il cluster Beocomp.

possono essere definiti allo scopo di implementare sofisticate politiche di gestione delle risorse. Le espressioni definite nei *Requirements* delle ClassAd dei calcolatori possono fare riferimento ad attributi definiti nei *port* della ClassAd del job (per i quali vanno trovati i matching).

Il modello del Gangmatching consente alla ClassAd del job di rendere note alle ClassAd che rappresentano le risorse le informazioni su una risorsa specifica attraverso un appropriato *port*. Ogni ClassAd che rappresenta un calcolatore non può infatti accedere agli attributi definiti nelle ClassAd degli altri calcolatori.

Il *port* *subjob*3 dichiarato nella ClassAd del job funziona come un'interfaccia astratta con le potenziali ClassAd dei calcolatori. La ClassAd del job implementa tale interfaccia nel modo seguente: nel *port* *subjob*3 si può fare riferimento ad un attributo della ClassAd di un calcolatore compatibile con un altro *port* che lo precede nella lista; il port *subjob*3 rende noto alle ClassAd dei calcolatori il valore di tale attributo attraverso la sintassi $Ports[2].attributo$. La ClassAd del job in figura 2.7 rende noto alle ClassAd dei calcolatori il valore dell'attributo *Subnet* della ClassAd compatibile con il *port* "subjob1", attraverso la sintassi $Ports[2].Subnet1$.

Ad esempio potrebbe essere richiesto che il *subjob*1 e il *subjob*3 vengano eseguiti su calcolatori della stessa LAN aggiungendo tra i *Requirements* per il *subjob*3 l'espressione $subjob3.Subnet == Subnet1$.

L'algoritmo di Matchmaking nel modello del Gangmatching

La funzione dell'algoritmo di Matchmaking nel modello del Gangmatching è quella di schierare, per ogni ClassAd di richiesta di esecuzione di un job, una "gang" consistente di ClassAd.

La "gang" consistente viene costruita a partire da una "gang" degenere costituita da un'unica ClassAd *radice*. Successivamente ogni *port* "libero" della "gang" viene "unito" ad un *port* compatibile. Quest'ultimo andrà ricercato tra i *port* delle ClassAd non ancora schierate nella "gang" consistente.

Questa operazione viene ripetuta fino a quando tutti i *port* non sono "uniti" e la "gang" non è consistente (cioè tutte i requisiti sono soddisfatti).

In generale un match consiste di un *tree* di ClassAd. Ogni coppia di ClassAd adiacenti è collegata verificando che gli attributi di un *port* della prima soddisfino i *Requirements* di un *port* della seconda, e viceversa.

Nell'esempio del paragrafo precedente la "gang" consistente è costituita da quattro ClassAd: un job e tre calcolatori (vedi figura 2.11).

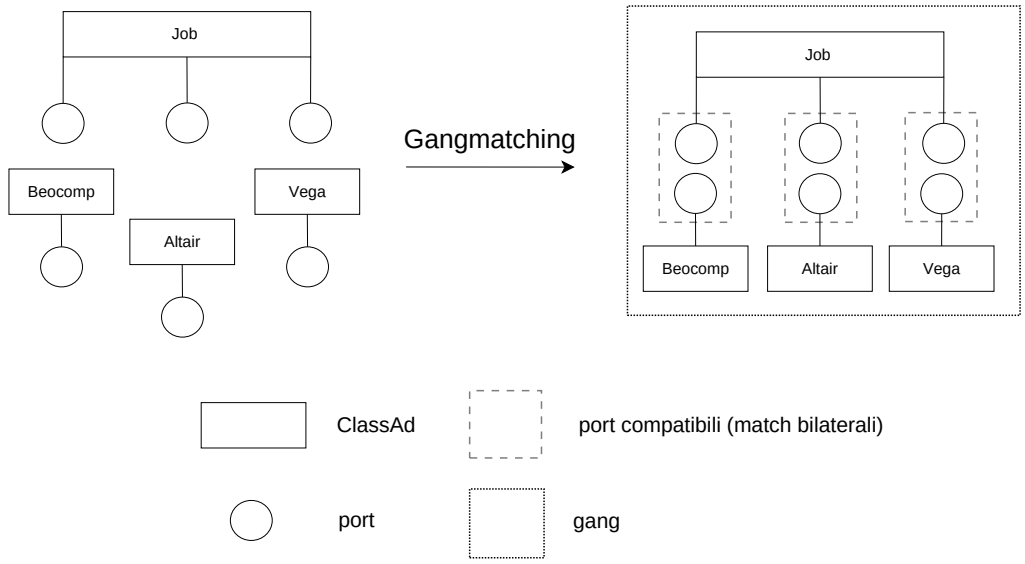

Figura 2.11: Operazione di Gangmatching.

Capitolo 3

La libreria MGF (MPI Globus Forwarder)

MGF (MPI Globus Forwarder) [34, 32, 33] è un'implementazione dello standard MPI-1 per Griglie Computazionali che estende MPICH-G2. Essa consente agli utenti di eseguire in modo trasparente applicazioni MPI su più calcolatori paralleli di una Grid, compresi i cluster a rete privata.

Infatti l'implementazione delle routine di comunicazione di MPICH-G2 non contempla il caso in cui i processi coinvolti siano in esecuzione su nodi interni di cluster a rete privata. Questo rappresenta un limite, in quanto nei cluster utilizzati per il calcolo ad alte prestazioni generalmente c'è un unico nodo che ha accesso alle reti WAN e LAN, il front-end, sia per la scarsa disponibilità di indirizzi IP pubblici, sia per semplificare l'amministrazione e la gestione della sicurezza del cluster.

L'assenza di un meccanismo di instradamento di messaggi tra le reti private, limita di fatto il numero di applicazioni che possono essere portate da un calcolatore parallelo convenzionale ad un sistema Grid costituito da più calcolatori paralleli, senza apportare modifiche al codice sorgente.

Per superare tale limitazione, MGF, utilizza processi "nascosti" all'utente, in esecuzione sui front-end dei cluster a rete privata, per consentire le comunicazioni che coinvolgono processi in esecuzione su nodi che non hanno

accesso alla rete WAN.

In conclusione i principali obiettivi di MGF sono:

- consentire l'esecuzione di applicazioni MPI su Grid senza apportare modifiche al codice sorgente;
- rendere accessibili all'utente informazioni dettagliate sulla topologia di rete del sistema durante l'esecuzione;
- consentire l'utilizzo del canale di comunicazione più efficiente tra quelli a disposizione per eseguire una qualsiasi comunicazione punto-punto;
- implementare comunicazioni collettive efficienti.

3.1 Canali di comunicazione

In questo contesto si definisce *canale di comunicazione* il percorso di rete che un messaggio deve compiere per arrivare da un processo MPI ad un altro. MGF distingue due classi di canali di comunicazione:

- *canali diretti* - realizzati utilizzando esclusivamente dispositivi di rete;
- *canali indiretti* - realizzati utilizzando processi intermedi.

I processi in esecuzione su nodi che accedono alla stessa rete utilizzano per comunicare i canali diretti; esempi sono: i nodi di uno stesso cluster, che comunicano attraverso la sua rete di interconnessione o i front-end di cluster differenti, che comunicano attraverso Internet.

I canali indiretti sono necessari quando i processi coinvolti nella comunicazione sono in esecuzione su nodi appartenenti a reti private differenti. In questo caso la comunicazione avviene mediante processi intermedi che si occupano di instradare i messaggi tra le reti. MGF utilizza MPICH-G2 nelle comunicazioni sui canali diretti mentre gestisce direttamente le comunicazioni sui canali indiretti.

3.2 Processi Forwarder

Quando si utilizza la libreria MGF per eseguire applicazioni MPI su Grid, oltre ai processi di calcolo possono essere allocati dei processi aggiuntivi detti *Forwarder*.

I Forwarder sono processi di servizio in esecuzione, su richiesta dell'utente, sui nodi front-end dei cluster a rete privata e gestiscono le comunicazioni indirette. Ogni comunicazione esterna che ha come destinatario un processo in esecuzione su un nodo interno di un cluster a rete privata viene gestita dal Forwarder: quando un processo in esecuzione su un altro calcolatore vuole comunicare con il processo suddetto, le primitive di comunicazione della libreria MGF fanno si che il messaggio venga spedito al Forwarder che si occupa di recapitarlo al destinatario. Analogamente il Forwarder può gestire le comunicazioni provenienti dai nodi interni del cluster a rete privata destinate a processi in esecuzione su altri calcolatori.

I Forwarder utilizzano l'implementazione MPI proprietaria (quando esiste) nelle comunicazioni con i nodi del proprio cluster ed il TCP nelle comunicazioni esterne.

L'utente abilita l'esecuzione del Forwarder definendo la variabile di ambiente **MGF_PRIVATE_SLAN** nello script RSL utilizzato per lo start-up del job.

Quando si utilizza MGF il comunicatore **MPI_COMM_WORLD** raggruppa processi di calcolo e Forwarder, anche se questi ultimi non svolgono attività di calcolo. Per evitare che il progamma conteggi erroneamente anche i Forwarder come processi coinvolti nel calcolo, MGF introduce un nuovo comunicatore denominato **MGF_COMM_WORLD** a cui appartengono soltanto i processi di calcolo. In fase di compilazione il comunicatore **MPI_COMM_WORLD** viene sostituito automaticamente con **MGF_COMM_WORLD**. In questo modo le routine MPI che accedono ai comunicatori (come *MPI_Comm_Size()* e *MPI_Comm_Rank()*), invocate con l'argomento **MPI_COMM_WORLD**, restituiscono le informazioni relative a **MGF_COMM_WORLD** nascondendo all'utente la presenza

dei Forwarder. Ad esempio richiedendo il numero dei processi in esecuzione, si ottiene il numero dei processi di calcolo, escludendo dal conto eventuali Forwarder. Quindi tale sostituzione consente all'utente di effettuare una corretta suddivisione del carico senza dover necessariamente modificare il proprio codice.

3.3 Gestione della topologia

L'esecuzione di un'applicazione MPI su un sistema complesso come quello Grid coinvolge differenti tipologie di interconnessioni.

La topologia descrive il tipo di interconnessione tra i nodi su cui sono in esecuzione i processi e consente di ricavare le informazioni sui canali di comunicazione disponibili.

MPICH-G2 organizza processi e connessioni in una struttura multilivello in cui ad ogni livello corrisponde un canale di comunicazione (vedi paragrafo A.2.4). Il livello 0 è il TCP su WAN, il livello 1 è il TCP su rete LAN, il livello 2 è il TCP sulla rete interna del calcolatore e il livello 3 è la libreria MPI proprietaria.

Per descrivere la topologia MPICH-G2 assegna a tutti i processi un *colore* (intero non negativo) per ogni livello. Due processi che hanno lo stesso colore ad un dato livello possono comunicare sul corrispondente canale di comunicazione.

MPICH-G2 assume che tutti i processi MPI siano in grado di comunicare sulla rete WAN e quindi assegna loro lo stesso colore al livello 0. L'assegnazione del colore non tiene però conto dell'eventuale presenza di cluster a rete privata, nei quali solo il front-end accede alla reti WAN e LAN, mentre gli altri nodi hanno accesso alla sola rete interna (vedi paragrafo A.2.4).

Per superare questa limitazione, MGF, nel caso in cui nella computazione siano conivolti cluster a rete privata, integra le informazioni sulla topologia ricostruite da MPICH-G2 con nuove informazioni. In particolare utilizza una nuova struttura dati per descrivere la topologia di rete WAN. Questa

struttura è un vettore di N interi, dove N è il numero dei processi di calcolo, nel quale l'i-mo elemento descrive l'accesso dell'i-mo processo alla rete WAN. Se l'i-mo elemento ha valore 0, questo vuol dire che l'i-mo processo è in esecuzione su un nodo che ha accesso diretto alla rete WAN; se ha valore -1, questo vuol dire che l'i-mo processo è in esecuzione su un nodo interno di un cluster a rete privata.

3.4 Comunicazioni punto-punto

MGF gestisce direttamente le comunicazioni punto-punto sui canali di comunicazione indiretti, mentre invoca le primitive di comunicazione di MPICH-G2 su quelli diretti.

In una comunicazione punto-punto MGF individua la disponibilità di canali diretti analizzando le informazioni topologiche proprie di MPICH-G2 e la struttura che descrive la topologia della rete WAN propria di MGF. Se due processi hanno lo stesso colore ai livelli 2 o 3 della struttura multilivello di MPICH-G2, questo vuol dire che essi sono in esecuzione su nodi di uno stesso cluster e che quindi possono comunicare su un canale diretto. In caso contrario MGF accede alle informazioni sulla topologia della rete WAN: se entrambi i processi hanno valore 0 nel vettore che descrive tale topologia, essi possono utilizzare un canale diretto poichè hanno entrambi accesso ad Internet, altrimenti utilizzano un canale indiretto

Se i processi possono comunicare attraverso un canale diretto, MGF invoca le corrispondenti routine MPICH-G2 che automaticamente selezionano il canale di comunicazione più efficiente: vMPI o TCP (vedi paragrafo A.2.5).

MGF gestisce le comunicazioni punto-punto sui canali indiretti utilizzando i processi Forwarder. I Forwarder utilizzano la libreria MPI proprietaria (se disponibile) per le comunicazioni con i nodi del proprio cluster e il TCP per le comunicazioni su LAN e WAN. Quando un processo in esecuzione su un nodo interno di un cluster a rete privata deve inviare un messaggio ad un processo in esecuzione all'esterno del cluster, esso invia il messaggio ed il

rango del destinatario al Forwarder in esecuzione sul front-end dello stesso cluster. Il Forwarder si occupa di recapitare il messaggio al destinatario.

Analogamente, quando un processo esterno al cluster a rete privata deve inviare un messaggio ad un processo in esecuzione su un nodo interno, esso invia il messaggio ed il rango del destinatario al Forwarder. Quest'ultimo si occupa di recapitare il messaggio al destinatario.

L'utilizzo simultaneo di due Forwarder rende possibile le comunicazioni punto-punto anche tra nodi interni di cluster a rete privata differenti (entrambi hanno valore -1 nella struttura che descrive la topologia di rete WAN di MGF).

L'operazione di forwarding è completamente trasparente all'utente.

3.5 Comunicazioni collettive

MGF eredita l'implementazione delle comunicazioni collettive di MPICH-G2. Tale implementazione utilizza le informazioni sulla topologia per minimizzare le comunicazioni sui canali più lenti. MGF inoltre consente la corretta esecuzione delle comunicazioni collettive nei casi nei quali ciò non è possibile con MPICH-G2 (vedi paragrafo A.2.5).

Si consideri ad esempio la funzione di broadcast. L'algoritmo di broadcast di MPICH-G2 è strutturato in tre fasi (vedi paragrafo A.2.5): nella prima fase la radice invia il messaggio di broadcast ai processi master di ogni LAN; nella seconda fase, presso ogni sito, il master invia il messaggio di broadcast a tutti i rappresentati dei calcolatori della stessa LAN; nella terza ed ultima fase i master dei calcolatori inviano il messaggio a tutti i nodi dei rispettivi calcolatori.

La presenza di un cluster a rete privata, può pregiudicare il corretto svolgimento dell'algoritmo: quando il processo radice è in esecuzione su un nodo interno (la prima fase non può essere eseguita correttamente), oppure quando il comunicatore del broadcast contiene tutti i nodi di un cluster a rete privata escluso il front-end (il master del cluster è in esecuzione su un nodo interno

e non può ricevere il messaggio dal processo master della corrispondente rete LAN).

L'implementazione della routine di broadcast della libreria MGF supera queste limitazioni utilizzando i processi Forwarder.

Se il processo radice è in esecuzione su un nodo interno di un cluster a rete privata, esso spedisce il messaggio di broadcast e i ranghi dei processi master al livello 0 al processo Forwarder in esecuzione sullo stesso cluster. Il Forwarder si occupa di spedire il messaggio ai processi master di ogni LAN. L'utilizzo del Forwarder consente di memorizzare il messaggio sul front-end e inviarlo ai vari processi processi master, evitando che il messaggio venga comunicato dalla radice al front-end per ogni comunicazione con l'esterno del cluster.

Se un processo master di una LAN è in esecuzione su un nodo interno di un cluster a rete privata, la seconda fase dell'algoritmo viene completata dal Forwarder in esecuzione su quel cluster, il quale si occupa di spedire il messaggio agli altri processi master della stessa LAN.

Nelle comunicazioni punto-punto vengono utilizzate le routine di MGF e pertanto anche se il processo master di un cluster a rete privata è in esecuzione su un nodo interno, il messaggio gli viene recapitato tramite il Forwarder in esecuzione sul front-end dello stesso cluster.

3.6 Usare MGF

Per utilizzare la libreria MGF è sufficiente compilare i programmi MPI con il wrapper per la compilazione di MPICH-G2 e linkare i file oggetto così ottenuti con la libreria MGF.

MGF utilizza gli strumenti del Globus Toolkit per avviare l'esecuzione di un programma MPI su più calcolatori. L'utente realizza uno script RSL che contiene le informazioni sull'esecuzione allo stesso modo con cui farebbe utilizzando MPICH-G2 (vedi paragrafo A.2.2) e avvia la computazione attraverso il comando **globusrun**:

```
% globusrun -w -f nomefile.rsl
```

Supponiamo di voler eseguire un programma MPI su tre cluster (a rete privata) del sistema Grid di supporto (vedi appendice B). In figura 3.1 è mostrato un esempio di script RSL per l'esecuzione del programma.

```
+
( &(resourceManagerContact=‘‘vega.na.icar.cnr.it:2122’’)
   (count=17)
   (label=‘‘subjob 0’’)
   (jobtype=mpi)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (P4_SETS_ALL_ENVVARS 1)
                (MGF_PRIVATE_SLAN 1)
                (LD_LIBRARY_PATH /opt/globus-2.4.3/lib/))
   (directory=‘‘/home/frangreg/’’)
   (executable=‘‘gsiftp://beocomp.dma.unina.it/users/home3/frangreg/hostname.g2’’)
( &(resourceManagerContact=‘‘altair.dma.unina.it:2122’’)
   (count=4)
   (label=‘‘subjob 17’’)
   (jobtype=mpi)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (P4_SETS_ALL_ENVVARS 1)
                (MGF_PRIVATE_SLAN 1)
                (LD_LIBRARY_PATH /opt/globus-2.4.3/lib/))
   (directory=‘‘/users/home3/frangreg/’’)
   (executable=‘‘gsiftp://beocomp.dma.unina.it/users/home3/frangreg/hostname.g2’’)
( &(resourceManagerContact="beocomp.dma.unina.it:2122")
   (count=14)
   (label="subjob 21")
   (jobtype=mpi)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
                (P4_SETS_ALL_ENVVARS 1)
                (MGF_PRIVATE_SLAN 1)
                (LD_LIBRARY_PATH /opt/globus-2.4.3/lib/))
   (directory="/users/home3/frangreg")
   (executable="/users/home3/frangreg/hostname.g2")
)
```

Figura 3.1: Esempio di script RSL per l'esecuzione di un programma MPI che utilizza la libreria MGF.

MGF attiva un processo Forwarder su ogni cluster per cui è stata dichiarata la variabile d'ambiente MGF_PRIVATE_SLAN.

Attraverso lo script RSL in figura 3.1, su Vega viene richiesta l'esecuzione di 17 processi, 16 processi di calcolo e un processo Forwarder, su Altair viene richiesta l'esecuzione di 4 processi, 3 processi di calcolo e un processo Forwarder, infine su Beocomp viene richiesta l'esecuzione di 14 processi, 13 processi di calcolo e un processo Forwarder.

Capitolo 4

Test preliminari

In questo capitolo sono descritti i risultati di alcuni test di esecuzione dell'implementazione parallela multi-sito dell'algoritmo del BCG.

Tali test sono stati effettuati utilizzando un sistema Grid di supporto (vedi appendice B) che comprende tre cluster: Beocomp, costituito da 15 processori Pentium II 450MHz con 256MB di memoria connessi tramite rete Ethernet, Altair, costituito da 16 processori Pentium Pro 200MHz con 128MB di memoria connessi tramite due reti Ethernet e Vega, costituito da 19 processori Pentium 4 1500MHz con 512MB di memoria connessi tramite rete Ethernet. Vega è installato presso i locali dell'Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR-CNR), mentre Beocomp ed Altair risiedono sulla stessa LAN presso i locali del Dipartimento di Matematica e Applicazioni "R. Caccioppoli" dell'Università degli Studi di Napoli Federico II.

Per l'esecuzione su più cluster (Grid) è stata utilizzata la libreria MGF (vedi Capitolo 3) per consentire la corretta esecuzione delle primitive di comunicazione punto-punto anche nel caso in cui i processi coinvolti fossero in esecuzione su nodi interni con accesso alla sola rete privata. Per l'esecuzione su un solo cluster è stata utilizzata la libreria MPICH.

Per la sottomissione del job su più cluster è stato utilizzato l'MPI jobs management system (vedi Capitolo 2). Un esempio di submit description file utilizzato per la sottomissione è illustrato in figura 4.1.

```
(subjob1)
universe = globus
executable = bcg_dist
arguments = s3rmt3m3.mtx 3 bs3rmt3m3.mtx 7 15 16 17 18 24 25 31
machine_count = 17
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = s3rmt3m3.mtx,s3rmt3m3_rhs.mtx
output = outfile.$(Cluster)
error = errfile.$(Cluster)
log = logfile.$(Cluster)
environment = P4_SETS_ALL_ENVVARS=1; MGF_PRIVATE_SLAN=1; LD_LIBRARY_PATH=/opt/globus-2.4.3/lib/
globusrsl = (jobtype=mpi) (count=17) (label=subjob 0)
requirements = (OpSys == ``LINUX'' && Arch == ``i686'') && (ClusterNodeCount >= 17)
rank = 10000/ClusterNetLatency + 10*ClusterCPUsSpecFloat
queue

(subjob2)
universe = globus
executable = bcg_dist
arguments = s3rmt3m3.mtx 3 bs3rmt3m3.mtx 7 15 16 17 18 24 25 31
machine_count = 4
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = s3rmt3m3.mtx,s3rmt3m3_rhs.mtx
output = outfile.$(Cluster)
error = errfile.$(Cluster)
log = logfile.$(Cluster)
environment = P4_SETS_ALL_ENVVARS=1; MGF_PRIVATE_SLAN=1; LD_LIBRARY_PATH=/opt/globus-2.4.3/lib/
globusrsl = (jobtype=mpi) (count=4) (label=subjob 17)
requirements = (OpSys == ``LINUX'' && Arch == ``i686'') && (ClusterNodeCount >= 4)
queue

(subjob3)
universe = globus
executable = bcg_dist
arguments = s3rmt3m3.mtx 3 bs3rmt3m3.mtx 7 15 16 17 18 24 25 31
machine_count = 14
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = s3rmt3m3.mtx,s3rmt3m3_rhs.mtx
output = outfile.$(Cluster)
error = errfile.$(Cluster)
log = logfile.$(Cluster)
environment = P4_SETS_ALL_ENVVARS=1; MGF_PRIVATE_SLAN=1; LD_LIBRARY_PATH=/opt/globus-2.4.3/lib/
globusrsl = (jobtype=mpi) (count=14) (label=subjob 21)
requirements = (OpSys == ``LINUX'' && Arch == ``i686'') && (ClusterNodeCount >= 14)
rank = 10*ClusterCPUsSpecFloat
queue
```

Figura 4.1: Esempio di submit description file per la sottossione del Gradiente Coniugato a Blocchi parallelo multi-componente.

Il subjob1 richiede l'esecuzione dei task 1 e 2. Quest'ultimi sono i più

onerosi dal punto di vista computazionale, sono suddivisi in processi paralleli e richiedono comunicazioni intra-task. Per questo subjob, quindi, le preferenze sono per i calcolatori paralleli con migliori prestazioni sia per la rete di interconnessione sia per la potenza di calcolo floating point ($rank = 10000/ClusterNetLatency + 10 * ClusterCPUsSpecFloat$).

I due task verranno eseguiti da CPU dello stesso calcolatore. In questo modo, si è tenuto conto del fatto che la quantità di dati scambiati tra i task 1 e 2, ad ogni passo, è maggiore rispetto a quella scambiata da qualsiasi altra coppia.

Il subjob3 richiede l'esecuzione dei task 6, 7 e 8. I task 6 e 8 sono i task, che al pari dell'1 e del 2, sono i più onerosi dal punto di vista computazionale. Sono suddivisi tra processi paralleli ma non richiedono comunicazioni intra-task. Per questo subjob, quindi, le preferenze sono per i calcolatori paralleli con migliori prestazioni per il calcolo floating point, senza esprimere preferenze in merito alle prestazioni della rete di interconnesione ($rank = 10 * ClusterCPUsSpecFloat$).

Il subjob 4 richiede infine l'esecuzione dei task 3, 4 e 5. Quest'ultimi hanno una complessità computazionale molto inferiore rispetto agli altri ed ognuno viene eseguito da un singolo processo. Per questo subjob non sono espresse preferenze specifiche.

A partire dal submit description file di figura 4.1 il Coordinator crea una ClassAd definendo un *port* per ogni subjob e i corrispondenti requisiti e preferenze all'interno di ogni *port*. Tale ClassAd viene comunicata al GangMatchMaker. L'operazione di Gangmatching ha come risultato il seguente mapping con le risorse del sistema Grid di supporto (vedi appendice B): la gang consistente è costituita dalla ClassAd succitata e le ClassAd di Vega, Altair e Beocomp; in particolare si stabiliscono i matching bilaterali tra il subjob1 e Vega, il subjob2 e Altair, il subjob3 e Beocomp.

In questo modo l'applicazione viene eseguita su Vega utilizzando 16 processi di calcolo, (8 per ognuno dei task 1 e 2) su Altair utilizzando 3 processi di calcolo (1 per ognuno dei task 3, 4 e 5) e su Beocomp utilizzando 13 pro-

cessi di calcolo (6 per ognuno dei task 6 e 8, 1 per il task 7), per un totale di 32 processi di calcolo (figura 4.2).

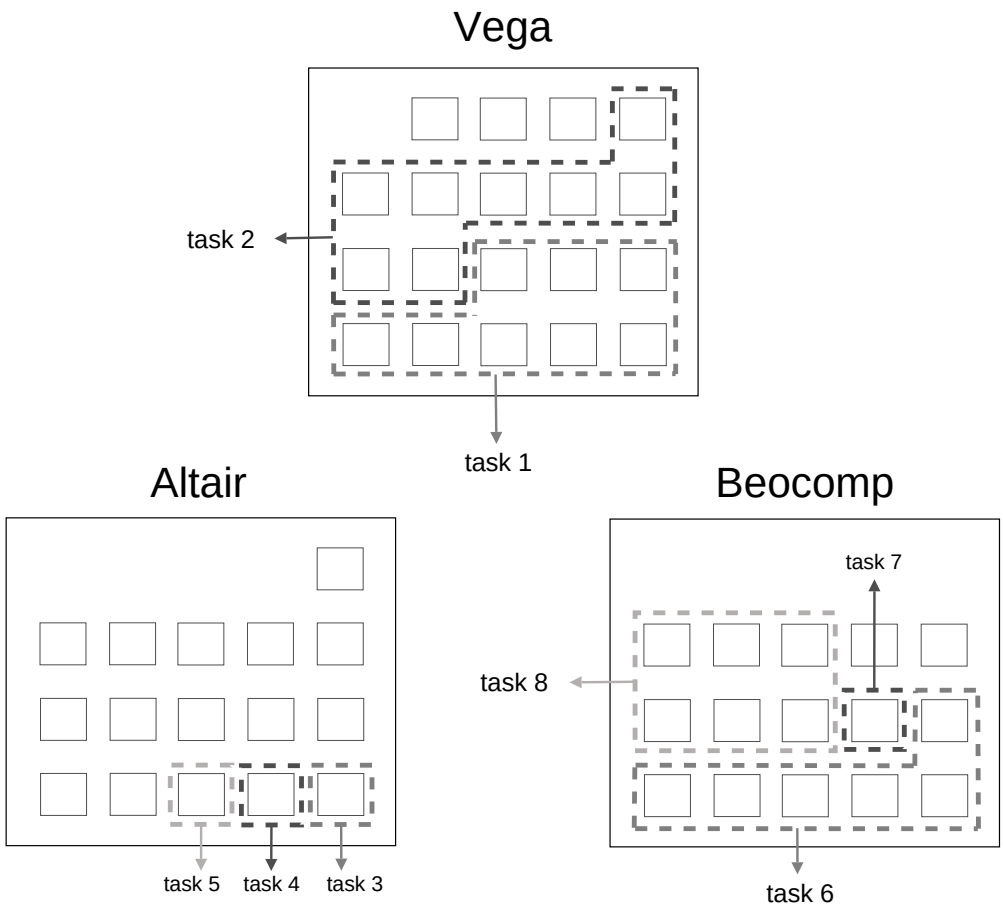

Figura 4.2: Mapping con le risorse del sistema Grid di supporto.

Per l'esecuzione su un solo cluster, Vega, sono stati utilizzati invece 16 processi di calcolo.

Sono stati effettuati dei test di esecuzione dell'applicazione parallela multi-sito del Gradiente Coniugato a Blocchi per problemi di dimensioni variabili. In figura 4.3 sono illustrati i tempi di esecuzione dell'applicazione per proble-

mi di dimensione: $(n = 5357, b = 3)$, $(n = 10974, b = 3)$, $(n = 13681, b = 3)$, $(n = 15439, b = 3)$.

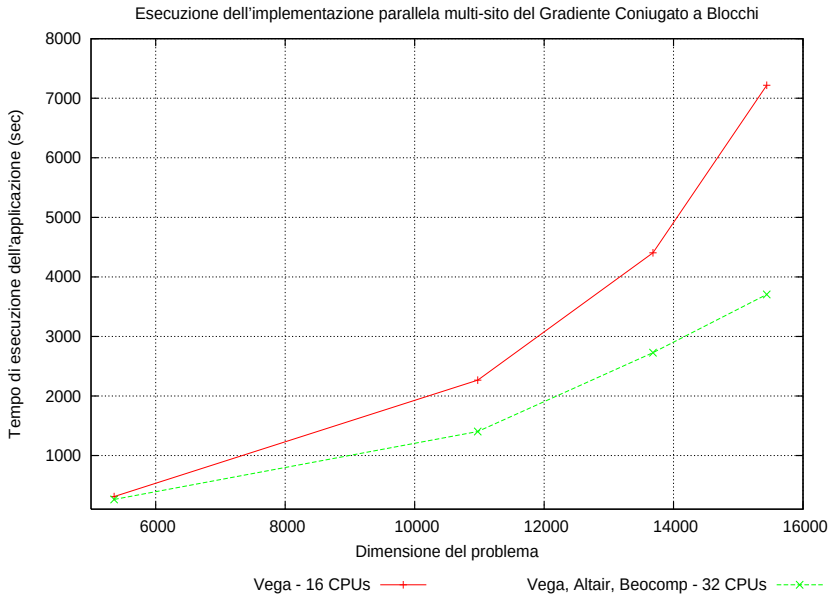

Figura 4.3: Esecuzione dell'implementazione parallela multi-sito del Gradiente Coniugato a Blocchi: Il tempo di esecuzione dell'applicazione corrisponde all'intervallo di tempo che intercorre tra l'inizio della fase di inizializzazione e il termine della computazione (calcolo della soluzione al passo $\overline{k}$).

Dal grafico appare evidente come la potenza di calcolo aggregata resa possibile dalla disponibilità di più calcolatori consente di ottenere dei benefici rispetto all'utilizzo di un singolo cluster (Vega[1]). Il tempo di esecuzione dell'applicazione si riduce per dimensioni del problema superiori a $n = 5000$, nonostante l'overhead introdotto dalle comunicazioni su WAN e LAN (figura 4.4), e tale riduzione aumenta considerevolmente all'aumentare delle stesse dimensioni.

La tabella 4.1 mostra lo speedup del tempo di esecuzione sul sistema

[1]È stato utilizzato come confronto Vega perchè è quello con le prestazioni migliori.

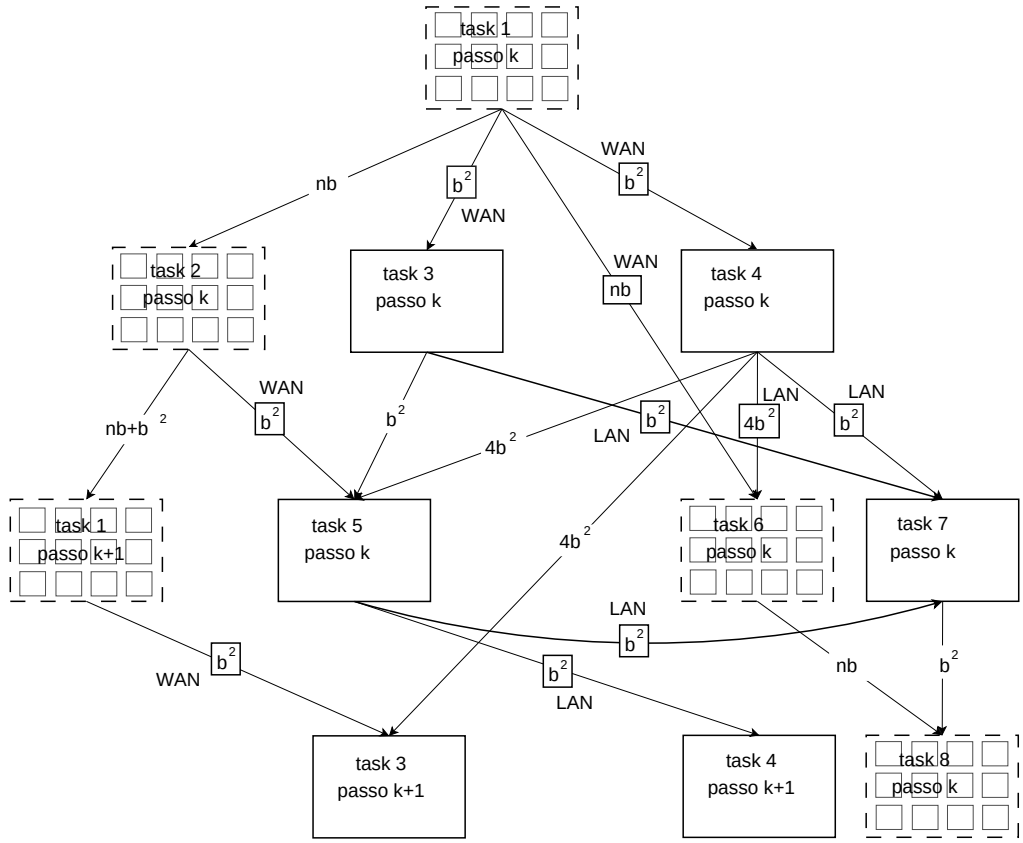

Figura 4.4: Grafo pesato delle dipendenze dei task dell'algoritmo del BCG: Indicazione delle comunicazioni che avvengono su WAN e LAN per effetto del mapping con le risorse del sistema Grid di supporto ottenuto a partire dal file di figura 4.1.

Grid di supporto rispetto al tempo di esecuzione sul singolo cluster più performante al variare di n.

In particolare si noti che per $n = 15439$ il tempo di esecuzione sul sistema Grid è poco più della metà di quello sul singolo cluster Vega (speedup 1.95).

Il sistema Grid ha 32 CPU e una potenza computazionale aggregata di $\approx$ 30000 Mflop/s (5850 Mflop/s con 13 CPU di Beocomp, 600 Mflop/s con 3 CPU di Altair, 24000 Mflop/s con 16 CPU di Vega) significativamente infe-

n	speedup
5357	1.18
10974	1.61
13681	1.61
15439	1.95

Tabella 4.1: Speedup del tempo di esecuzione sul sistema Grid di supporto rispetto al tempo di esecuzione sul singolo cluster più performante.

riore a quella disponibile se Vega avesse avuto 32 CPU pari a 48000 Mflop/s. Pertanto, l'utilizzo del sistema Grid, grazie ad un miglior bilanciamento del carico a livello delle risorse, consente di dimezzare il tempo di esecuzione dell'applicazione (rispetto all'utilizzo del singolo cluster Vega) senza raddoppiare la potenza computazionale.

Appendice A

Grid Middleware

Il middleware è uno strato software atto a mascherare l'eterogeneità delle risorse consentendo all'utente di astrarsi dalla complessità dell'ambiente Grid. La sua funzione è agevolare la progettazione, la realizzazione e la gestione dell'esecuzione di applicazioni distribuite, fornendo un ambiente di sviluppo distribuito, integrato e consistente.

In questo capitolo verranno descritti i seguenti software di middleware:

- Globus Toolkit [23, 28] - una collezione di strumenti open-source modulari, affermatosi come standard *de facto* per il Grid Computing.
- MPICH-G2 [38] - implementazione grid-enabled dello standard MPI-1, basata sulla libreria MPICH.
- Condor-G [26] - un agente per la gestione della computazione in una Grid, che integra le tecnologie di Condor [14, 40, 21] e del Globus Toolkit.

A.1 Il Globus Toolkit

Il Globus Toolkit è un software *open source* che rappresenta uno standard *de facto* per il Grid Computing [23, 28].

Esso comprende un insieme di servizi e librerie sviluppati per realizzare la tecnologia Grid, consentendo la condivisione sicura di risorse eterogenee e distribuite geograficamente, appartenenti a domini di sicurezza e gestione non omogenei.

È confezionato sotto forma di un insieme di componenti modulari che possono essere utilizzati sia indipendentemente che in modo congiunto per sviluppare applicazioni e tool in ambiente Grid. In sostanza il Toolkit offre una cosiddetta *bag of services*, da cui gli sviluppatori di applicazioni o strumenti che necessitano dell'infrastruttura Grid, possono attingere secondo le proprie necessità.

Il Globus Toolkit è stato realizzato dalla Globus Alliance, una collaborazione internazionale che comprende:

- Argonne National Laboratory, University of Chicago [1].
- Information Sciences Institute (University of Southern California) [9].
- EPCC, University of Edinburgh [2].
- National Center for Supercomputing Applications (NCSA) [4].
- Northern Illinois University, High Performance Computing Laboratory [5].
- Royal Institute of Technology, Sweden [6].
- Univa Corporation [8].
- Swedish Center for Parallel Computers [7].

La condivisione delle risorse viene realizzata senza apportare modifiche all'infrastruttura sottostante. Questo è di fondamentale importanza in una realtà multi-istituzionale, poichè consente ai possessori delle risorse di conservarne il controllo locale.

I componenti di Globus sono organizzati secondo la struttura a livelli rappresentata in figura A.1. I componenti di ogni livello condividono caratteristiche e completano le funzionalità dei componenti dei livelli inferiori, in maniera del tutto analoga a quanto accade nel modello a livelli del TCP/IP [51].

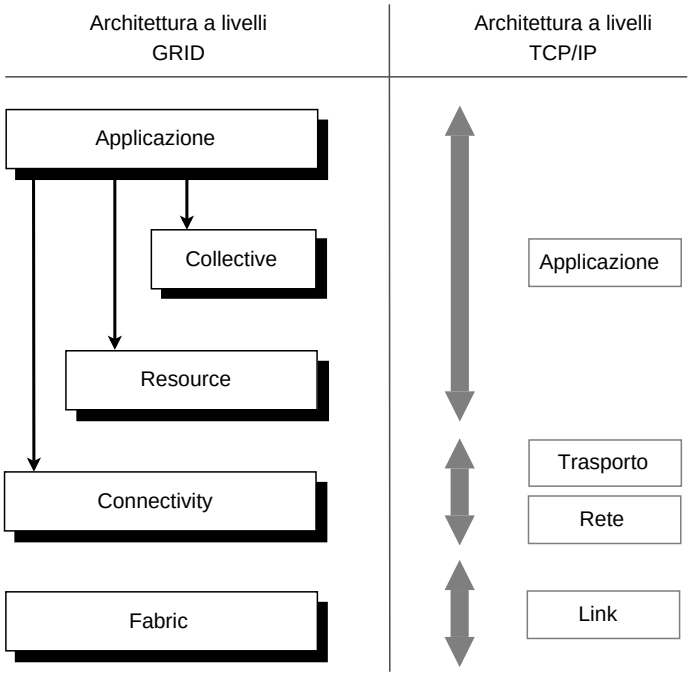

Figura A.1: L'architettura a livelli del Grid e la sua correlazione con il modello TCP/IP.

Complessivamente si può individuare, nella sovrapposizione dei livelli, una struttura a "clessidra", nella quale la strozzatura centrale rappresenta un piccolo insieme di astrazioni e protocolli che consentono un elevato livello di connettività fra le applicazioni e i servizi di più alto livello che accedono alle risorse condivise (alla sommità della clessidra) e le tecnologie sottostanti (alla base della clessidra).

La strozzatura centrale consiste dei protocolli di *Resource* e di *Connectivity*, che facilitano la condivisione delle singole risorse.

I protocolli a questi livelli sono progettati in modo che possono essere implementati su un'ampia varietà di tipi di risorse, definite al livello *Fabric*. I suddetti protocolli vengono a loro volta utilizzati per costruire un'ampia varietà di servizi globali e applicazioni posizionati al livello *Collective* (così chiamato perchè coinvolge l'uso coordinato di più risorse).

Il punto di forza di questo modello architetturale consiste nella possibilità che molteplici applicazioni e servizi di gestione collettivi (ai livelli più alti), e molteplici servizi di gestione locali alle singole risorse (ai livelli più bassi), siano connettibili tramite pochi protocolli di trasporto.

L'importanza di definire un minimo insieme di protocolli *intergrid* scaturisce dalla necessità di consentire l'interoperabilità tra i diversi sistemi Grid.

I servizi forniti dal Globus Toolkit si posizionano nella strozzatura centrale della clessidra, fornendo un'interfaccia attraverso la quale le applicazioni di alto livello utilizzano in modo trasparente i sistemi sottostanti.

Attraverso tali servizi un sistema distribuito di più calcolatori funziona come se si trattasse di un unico *metacalcolatore*[1] accessibile mediante un minimo insieme ben definito di API (*Application Programming Interfaces*)[2] e SDK (*Software Development Kit*)[3] che rendono disponibile un ambiente di runtime comune, favorendo la portabilità delle applicazioni.

I servizi forniti dal Toolkit possono essere raggruppati in quattro categorie:

[1]Sistema formato dall'aggregazione dinamica di nodi computazionali non dedicati al metacalcolatore, interconnessi attraverso una rete non dedicata (anche Internet) e appartenenti a domini di sicurezza e gestione non omogenei.

[2]Specifica di un insieme di routine per facilitare lo sviluppo di applicazioni; si riferisce alla loro definizione e non ad una particolare implementazione.

[3]Consiste di librerie e strumenti che forniscono un'implementazione delle specifiche di un API e di programmi di base per il loro utilizzo.

- **Security**: servizi di sicurezza, autenticazione, autorizzazione e delega delle credenziali all'interno delle Virtual Organizations.
- **Data Management**: servizi per la gestione dei dati.
- **Execution Management**: servizi per l'allocazione delle risorse, per il controllo, lo scheduling e la coordinazione delle computazioni remote.
- **Information Services**: servizi che forniscono informazioni sui componenti del sistema.

A.1.1 Security

Le applicazioni Grid, durante la loro esecuzione, possono acquisire, allocare e rilasciare le risorse dinamicamente.

A tal fine, in questo ambiente è auspicabile l'esistenza di un sistema che consenta ai processi creati dagli utenti di utilizzare risorse Grid, attaverso meccanismi di delega, senza richiedere l'intervento dell'utente stesso.

Inoltre è auspicabile che i diversi possessori delle risorse, pur condividendole mediante la tecnologia Grid, possano continuare ad adottare le proprie politiche di sicurezza.

Il Globus Toolkit fornisce una serie di strumenti per constatare l'identità degli utenti e dei servizi (autenticazione), per garantire l'integrità dei dati e la riservatezza nelle comunicazioni e per stabilire quali utenti sono abilitati a eseguire quali azioni (autorizzazione).

Tali strumenti sono basati sulla crittografia a chiave pubblica ed il protocollo SSL/TLS (*Secure Sockets Layer/ Transport Layer Security*)[4], e sono sotto la denominazione *Grid Security Infrastructure* (GSI) [55].

In definitiva GSI è stato sviluppato per soddisfare le seguenti esigenze:

[4]Secure Sockets Layer (SSL) è un protocollo progettato per realizzare comunicazioni cifrate su Internet; la versione 3.0, rilasciata nel 1996, è stata utilizzata come base di sviluppo per il protocollo Transport Layer Security (TLS) [20], standard IETF [36].

- consentire comunicazioni sicure (autenticate e riservate) tra elementi della Griglia Computazionale;
- consentire ai proprietari delle risorse di adottare i propri meccanismi di sicurezza e strumenti di autenticazione locale;
- consentire con una singola operazione di autenticazione, *single sign-on*, attraverso delega delle credenziali, di utilizzare tutti i servizi e le risorse Grid per i quali si è autorizzati.

Nei successivi paragrafi verranno descritti in breve alcune caratteristiche fondamentali della GSI:

- l'utilizzo di certificati per l'autenticazione di utenti e risorse;
- il processo di mutua autenticazione tra le parti;
- la delega delle credenziali e sign-on singolo.

I certificati per l'autenticazione di utenti e servizi

GSI utilizza per autenticare utenti e servizi certificati a chiave pubblica conformi allo standard X.509 [13], stabilito dall'Internet Engineering Task Force (IEFT).

I certificati vengono rilasciati da autorità certificanti (*Certificate Authority* - CA) che verificano e registrano l'identità dell'entità da certificare.

Essi contengono quattro blocchi principali di informazioni:

- un *subject name* che identifica l'entità certificata;
- la chiave pubblica dell'entità certificata;
- l'identità della Certificate Authority (CA) che ha firmato il certificato (tramite firma digitale con la propria chiave privata) allo scopo di garantire l'associazione tra la chiave pubblica e l'identità dell'entità certificata;

- la firma digitale della CA designata.

Un'entità che riconosce una CA deve essere in possesso della sua chiave pubblica. In questo modo potrà verificare la firma digitale apposta sui certificati rilasciati dalla CA.

Mutua autenticazione

Se due entità sono in possesso di un certificato e entrambe riconoscono le CA che hanno firmato i rispettivi certificati, esse possono accertarsi delle rispettive identità tramite il processo di *mutua autenticazione* tra due parti.

GSI utilizza SSL (Secure Sockets Layer, anche noto come Transport Layer Security, TLS) per implementare il proprio protocollo di mutua autenticazione alla base della delega e del single sign-on.

Si descrive brevemente il processo di mutua autenticazione tra due parti A e B.

A stabilisce una connessione con B inviando il proprio certificato. Utilizzando tale certificato B risale all'identità di A, alla sua chiave pubblica e alla CA designata per certificarlo. B controlla la validità del certificato di A, controllando la firma digitale apposta su di esso dalla CA.

A questo punto B deve assicurarsi che A sia realmente l'entità identificata dal certificato. A tal fine B invia ad A un messaggio generato in maniera casuale ed A restituisce tale messaggio dopo averlo cifrato tramite la propria chiave privata.

B decifra il messaggio utilizzando la chiave pubblica contenuta nel certificato di A e verifica che il risultato coincida con il messaggio casuale originario.

La stessa operazione viene ripetuta a parti invertite, dopodichè il processo di mutua autenticazione è concluso.

Delega delle credenziali e sign-on singolo

GSI fornisce funzionalità di delega delle credenziali: un'estensione del protocollo standard SSL.

Esempi nei quali è richiesta la delega delle credenziali includono: il rilascio delle credenziali a processi "incustoditi", da eseguire sulle risorse Grid, senza intervento diretto dell'utente; la condivisione di file per un periodo di tempo limitato; l'impiego di servizi di brokering delle risorse che si preoccupano di acquisire risorse per conto dell'utente.

La delega delle credenziali avviene attraverso la creazione di un *proxy*.

Quest'ultimo consiste di un nuovo certificato (certificato proxy X.509), che contiene una nuova chiave pubblica, e di una nuova chiave privata. Il nuovo certificato contiene l'identità dell'utente, con l'indicazione che il certificato è un certificato proxy, ed è firmato dall'utente stesso. Inoltre il certificato contiene l'indicazione sul periodo di validità (limitato) del proxy.

Il proxy e la nuova chiave privata vengono memorizzati localmente, in un file nella directory **/tmp**, protetto dall'accesso degli altri utenti mediante i permessi del filesystem.

Una volta che il proxy è stato creato e memorizzato, il certificato proxy e la chiave privata vengono utilizzati per la mutua autenticazione sulle risorse Grid, sollevando l'utente dalla pratica ripetuta dell'inserimento della *pass phrase* (password utilizzata per cifrare la chiave privata).

I certificati proxy consentono quindi il sing-on singolo: l'utente si autentica una sola volta, attraverso il comando **grid-proxy-init** e digitando la *pass phrase*, allo scopo di creare il proxy. Quest'ultimo viene poi utilizzato per ripetute autenticazioni per un periodo di tempo limitato senza compromettere la protezione della chiave privata dell'utente.

In presenza del proxy, il processo di mutua autenticazione differisce leggermente. La parte remota riceve sia il certificato proxy (firmato dall'utente), che il certificato utente. La chiave pubblica del certificato utente viene utilizzata per verificare la validità della firma sul certificato proxy, mentre la

chiave pubblica della CA viene utilizzata per verificare la validità della firma sul certificato utente.

In questo modo si stabilisce una catena di fiducia dalla CA al proxy, attraverso l'utente.

Ogni proxy può essere delegato, a sua volta, a produrre un nuovo proxy di livello inferiore con un sottoinsieme di privilegi.

A.1.2 Data Management

Le applicazioni di calcolo ad alte prestazioni data-intensive utilizzano collezioni di dati, librerie digitali, e database distribuiti geograficamente e richiedono la gestione efficiente e il trasferimento su rete geografica di enormi quantità di dati, dell'ordine dei terabyte o petabyte.

Esempi di applicazioni di questo tipo sono le analisi sperimentali e le simulazioni in discipline scientifiche come la fisica delle alte energie, la modellistica climatica e l'astronomia. Queste applicazioni utilizzano enormi quantità di dati condivise da centinaia o migliaia di ricercatori distribuiti geograficamente.

Essi necessitano di trasferire grandi sottoinsieme di questi dati da risorse locali a risorse computazionali remote affinchè vengano elaborati. Altre volte possono creare copie locali (repliche) dei dati presso i propri siti o presso altre risorse remote, per evitare che la latenza dei trasferimenti su rete geografica rallenti l'elaborazione dei dati.

In definitiva, gli obiettivi principali del Data Management in Globus sono:

- consentire l'accesso ai dati e il loro trasferimento in modo efficiente e sicuro;
- gestire la replica dei dati all'interno dalla Grid.

L'accesso e traferimento di dati è implementato dal GridFTP e dal *Globus Access to Secondary Storage* (GASS), mentre la replica dei dati è implementata attraverso delle API come *globus_ replica_ manager()* e *globus_ replica_ catalog()*.

GridFTP è un'estensione del protocollo File Transfer Protocol (FTP). Il protocollo FTP è un protocollo standard IETF, ampiamente utilizzato per il trasferimento dei dati in Internet e implementato da numerosi software di pubblico dominio. FTP fornisce un'architettura ben definita che consente di supportare nuove estensioni e di rilevare le estensioni supportate da una particolare implementazione.

GridFTP consiste di un lato client e di un lato server. Il lato server è implementato dal demone *in.ftpd* mentre il lato client dal comando *globus-url-copy* e altre API associate.

Estensioni del protocollo FTP supportate da GridFTP sono:

- Supporto per l'autenticazione GSI, e quindi per il sign-on singolo.
- Trasferimento dei dati controllato da una terza parte, che fornisce la possibilità di iniziare, controllare e monitorare il trasferimento dei dati tra due server GridFTP remoti.
- Trasferimento parallelo di dati, che consente il trasferimento di più stream TCP di dati (anche quando il trasferimento avviene da un singolo server) in modo da aumentare la banda aggregata rispetto all'uso di un singolo stream [47].
- Trasferimento parziale dei file, che consente il trasferimento di una parte arbitraria di dati presenti all'interno di un file, una caratteristica necessaria quando si opera con file di grosse dimensioni.
- Supporto per un trasferimento dati affidabile e riavviabile, che fornisce un meccanismo di verifica e di ripristino dei trasferimenti interrotti in seguito a fallimenti: problemi di rete, server fuori servizio, ecc.

Il trasferimento di file tra server può avvenire anche attraverso GASS. A differenza di GridFTP, usato per il trasferimento di file dati di grandi dimensioni, GASS è utilizzato per trasferire gli eseguibili (associati ai job utente), e i file di input, output e di errore a loro associati. Quindi l'utilizzo di GASS è più strettamente collegato alla fase di sottomissione dei job.

Il trasferimento dei file gestito da GASS è anche detto *staging* dei file. Il processo di trasferimento dell'eseguibile e dei file di input dall'host dove sono memorizzati all'host dove l'eseguibile verrà eseguito (host di esecuzione), prende il nome di *staging-in*. Dall'altra parte, il processo di trasferimento dei file di output dall'host di esecuzione all'host da cui è stato sottomesso il job o all'host dove era inizialmente memorizzato l'eseguibile, prende il nome di *staging-out*.

Anche GASS, come GridFTP, ha un lato server e un lato client. Un client GASS verrà avviato automaticamente su ogni host di esecuzione ogniqualvolta verrà sottomesso un job. Il GASS server và invece avviato manualmente, solo quando è richiesto lo staging dei file, sul calcolatore dove sono memorizzati l'eseguibile e i file di input.

Un GASS server ha il proprio URL, di solito col prefisso *https://* per abilitare le proprietà di GSI, cosicchè i client possano identificare la sua locazione. L'eseguibile e i file dati verranno memorizzati sull'host di esecuzione in una directory, chiamata *cache storage*, che verrà rimossa una volta completata l'esecuzione dei job.

A.1.3 Information Services

In ambiente Grid la scoperta e il monitoraggio delle risorse e dei servizi divengono problemi complessi per la diversità, il comportamento dinamico il vasto numero e la distribuzione geografica delle entità alle quali un utente potrebbe essere interessato.

Di conseguenza, i Servizi di Informazioni, (*Information Services*), fornendo gli strumenti per la scoperta e il monitoraggio delle risorse, e quindi per la progettazione e l'adattamento all'ambiente dell'applicazione, risultano essere una parte fondamentale di qualsiasi software di infrastruttura Grid.

Il servizio MDS

Il Globus Toolkit include un insieme di *Information Services* che collettivamente vanno sotto il nome di *Monitoring and Discovery System* (MDS) [42].

MDS è stato progettato allo scopo di fornire uno standard per la pubblicazione e accesso alle informazioni relative allo stato e alla configurazione delle risorse dell'infrastruttura Grid [16].

Esso consente di aggregare informazioni provenienti da più fonti, e di renderle accessibili attraverso uno o più punti di accesso.

MDS è stato progettato per ottenere:

- un accesso efficiente e scalabile a dati dinamici;
- un accesso uniforme e flessibile alle informazioni;
- buoni tempi di risposta a frequenti e complessi aggiornamenti;
- un accesso a sorgenti di informazioni multiple;
- un mantenimento decentralizzato delle informazioni.

MDS rappresenta un'interfaccia tra i programmi che forniscono le informazioni e le applicazioni o i servizi di più alto livello (quali servizi di broker, monitoraggio, individuazione dei guasti) che ne fruiscono.

MDS2, sviluppato con il Globus Toolkit 2.x, è un'implementazione dell'MDS che utilizza Lightweight Directory Access Protocol (LDAP) [35] come back-end per ottenere tutte le informazioni. Esso è presente anche nelle versioni successive del Globus Toolkit (3.0, 3.2 e 4.0) allo scopo di supportare tutti i deployment esistenti.

MDS2 ha una struttura gerarchica (figura A.2) che si basa su tre componenti principali:

- *Information Provider* (IP): è un servizio locale ad una singola risorsa e fornisce informazioni sulla risorsa stessa.

- *Grid Resource Information Service* (GRIS): è un servizio distribuito che può rispondere a richieste di informazioni sullo stato e la configurazione di una particolare risorsa.

- *Grid Index Information Service* (GIIS): è un servizio che aggrega le informazioni provenienti da un insieme di GRIS server per fornire informazioni sullo stato delle risorse di un'intera organizzazione.

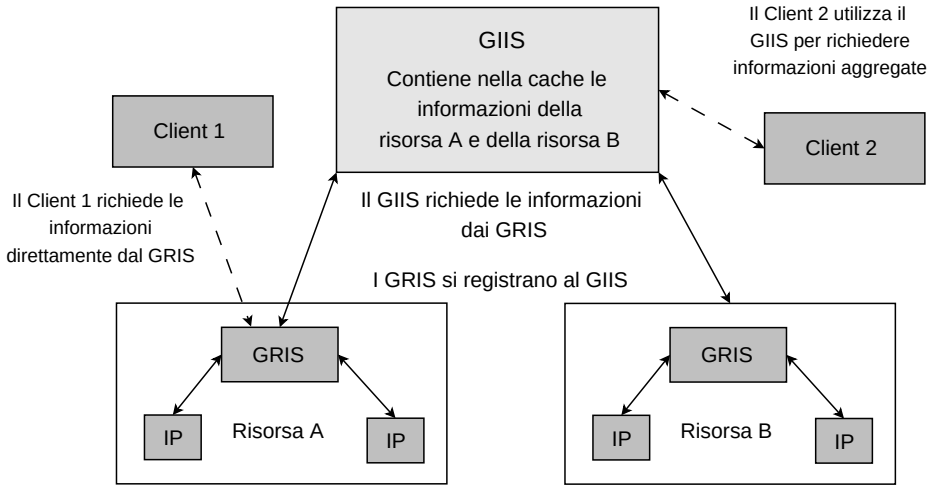

Figura A.2: Struttura gerarchica dell'MDS2: possono esserci più livelli di GIIS e ogni GRIS può registrarsi a qualsiasi GIIS e a sua volta ogni GIIS può registrarsi a qualsiasi altro GIIS creando un'architettura modulare ed estensibile.

Le interazioni all'interno del sistema informativo sono definite in termini di due protocolli base:

- Il *Grid Information Protocol* (GRIP) attraverso il quale, un utente o più frequentemente i componenti del Servizio di Informazioni, recuperano le informazioni dagli altri componenti.

- Il *Grid Registration Protocol* (GRRP) attraverso il quale un componente del Servizio di Informazioni notifica la sua esistenza agli altri componenti.

MDS2 utilizza il protocollo LDAP per implementare il protocollo di interrogazione (enquery) GRIP. LDAP, implementando il GRIP, definisce un modello per le informazioni, un linguaggio di interrogazione e il protocollo di trasporto.

Il GRRP è un protocollo di registrazione *soft-state*, che, in questo contesto, si traduce nel fatto che le notifiche vengono ripetute periodicamente e quindi i riferimenti alle risorse possono essere scartati se non vengono confermati da notifiche successive.

Ogni messaggio GRRP contiene il nome del servizio (cioè un URL al quale vengono reindirizzati i messaggi GRIP), il tipo di messaggio di notifica e l'indicazione sull'intervallo di tempo per il quale la notifica è considerata valida. La definizione di GRRP non specifica il "mezzo di trasporto" del messaggio. In MDS2 è stato utilizzato LDAP per implementare il protocollo di trasporto.

Nei paragrafi successivi verranno descritti i componenti del sistema informativo MDS2, come tali componenti utilizzano i protocolli descritti per interagire tra loro e il modello utilizzato per rappresentare i dati.

Information Provider (IP) Un *Information Provider* (IP) utilizza il protocollo di registrazione (GRRP) per notificare i servizi di più alto livello della sua esistenza; un servizio di più alto livello usa il protocollo di interrogazione (GRIP) per recuperare le informazioni su una risorsa monitorata da un IP.

MDS2 fornisce una serie di IP detti *Core Information Provider* che forniscono informazioni statiche (versione del sistema operativo, tipo di CPU,...) e dinamiche (carico medio, coda dello scheduler,...) relative gli host, informazioni sul sistema di memorizzazione (spazio disponibile sul filesystem, la dimensione della memoria centrale e di quella virtuale,...) e informazioni sulla rete attraverso il Network Weather Service (larghezza di banda e latenza della rete, sia misurate che previste).

MDS2 consente agli sviluppatori di implementare Information Provider

personalizzati detti *Custom Information Provider* per il monitoraggio di informazioni non contemplate dai Core IP.

Grid Resource Information Service (GRIS) Il *Grid Resource Information Service* (GRIS) è solitamente in esecuzione su ogni risorsa, fornendo tutte le informazioni relative a quella risorsa.

A tal scopo utilizza il modello dei dati, il linguaggio di interrogazione e il protocollo del GRIP per funzionare come un IP che contiene le informazioni suddette.

Il GRIS autentica ed analizza ogni richiesta di informazioni ricevuta e poi spedisce le richieste ad uno o più IP locali, a seconda del tipo di informazioni presenti nella richiesta. Per ridurre efficientemente l'elaborazione della ricerca, il GRIS filtra, prima di inviarli a un client o a un GIIS, i risultati restituiti da un IP, allo scopo di eliminare tutti gli oggetti che non corrispondono ai filtri di ricerca.

Il GRIS utilizza un meccanismo di salvataggio (*caching*) per le informazioni ricevute dagli IP in modo da ridurre il numero di interrogazioni. L'intervallo di tempo di validità per le informazioni salvate, detta time-to-live (TTL), viene comunicata al GRIS dagli IP e viene specificata attraverso la configurazione di questi ultimi. Il GRIS quando viene interrogato da un client, utilizza le informazioni salvate, a meno che il time-to-live non sia terminato (cache invalidata). In tal caso il GRIS interroga gli IP sottostanti.

Il protocollo GRRP viene invece usato da un GRIS per registrarsi ai componenti di più alto livello in modo da realizzare la struttura gerarchica.

Grid Index Information Service (GIIS) MDS2 fornisce un servizio per la costruzione di insiemi di informazioni aggregate (*aggregate directory*) denominato *Grid Index Information Service* (GIIS). Un'istanza di questo servizio fornisce un insieme di informazioni aggregate più semplice che descrive la struttura gerarchica dell'intera rete di informazioni aggregate. Il GIIS accetta i messaggi di registrazione da un GRIS o altre istanze del GIIS e fonde le informazioni ricevute in modo da creare uno spazio di informazioni unificato.

Quando un server GIIS viene interrogato per recuperare informazioni su diverse risorse grid, questi a sua volta interroga i GRIS in esecuzione sulle risorse in questione, raccoglie i risultati delle query e li restituisce all'applicazione.

Anche il GIIS, adotta il meccanismo di salvataggio delle informazioni basato sui TTL: estrae le informazioni dalla cache quando viene effettuata una richiesta da un client. Se le informazioni non sono più valide, perchè il time-to-live è terminato, invia la richiesta ai GIIS o ai GRIS sottostanti.

Il modello di rappresentazione dei dati e gli schemi di MDS2 MDS2, utilizzando la tecnologia LDAP, adotta un modello per le informazioni che deriva direttamente dal protocollo succitato [59].

Di conseguenza le informazioni sono organizzate in collezioni che prendono il nome di *entry* e che sono identificate da un proprio e unico *Distinguished Name* (DN).

Ogni entry è un'istanza a un determinato tipo di oggetto (un calcolatore, una rete, un'organizzazione,...). L'informazione riguardante una entry è rappresentata da uno o più attributi, ognuno consistente di un nome ed un corrispondente valore. Ad ogni entry è associata una *object class* che definisce gli attributi ad esso associati ed i tipi di dato ed i valori che tali attributi possono assumere.

Per semplificare il processo di individuazione di una entry queste ultime sono organizzate secondo un "namespace" gerarchico, strutturato ad albero, chiamato *Directory Information Tree* (DIT).

La definizione di una object class consiste di tre parti: una classe padre (che permette una sorta di ereditarietà consentendo la definizione di una object class come estensione di una esistente), una lista di attributi necessari ed una lista di attributi opzionali.

Gli *schemi* contengono le definizioni degli object class e dei tipi di attributi.

Schemi adottati da MDS2 sono l'MDS Core Schema [41] e il Grid Laboratory Uniform Environment (GLUE) Schema [29].

Il GLUE Schema è nato dalla collaborazione dei team dei progetti EU-DataTAG [19] e US-iVDGL [37]. Lo scopo era quello di produrre uno schema per la rappresentazione delle informazioni sulle risorse che permettesse l'interoperabilità tra differenti middleware di Griglia.

Lo schema viene descritto con diagrammi di classi UML [57] per favorire una struttura di informazione comune, indipendente dalla specifica tecnologia e dal modello dei dati.

A.1.4 Execution Management

L'utilizzo remoto e concorrente di risorse computazionali distribuite geograficamente, solleva alcune problematiche, che possono essere così sintetizzate:

- Autonomia del sito: le risorse appartengono a organizzazioni e domini di gestione differenti. In questo scenario deve essere comunque consentito ai proprietari delle risorse di mantenerne il pieno controllo, gestendo le politiche di utilizzo e di scheduling e i meccanismi di sicurezza.

- Eterogeneità dei sistemi di gestione locali: la gestione locale delle risorse può essere effettuata attraverso i più diversi sistemi. Di qui la necessità di consentire l'interazione con i più diffusi sistemi di gestione locale delle risorse.

- Estensibilità delle politiche: le applicazioni Grid sono progettate da un'ampia varietà di individui e/o organizzazioni e ognuna ha i propri requisiti. Una soluzione alla gestione delle risorse deve consentire lo sviluppo periodico di nuove strutture di gestione, specifiche per il dominio, senza richiedere cambiamenti al codice delle applicazioni.

- Co-allocazione: molte applicazione richiedono l'utilizzo contemporaneo di più risorse dislocate in diversi siti. La temporanea indisponibilità di alcune di esse suggerisce la necessità di meccanismi per l'allo-

cazione contemporanea di un insieme di risorse, che diano inizio alla computazione su quelle risorse, per il monitoraggio e la gestione delle computazioni.

- Controllo interattivo: le applicazioni possono prevedere un meccanismo di negoziazione che consente di utilizzare nuove risorse che si rendono disponibili nel corso dell'esecuzione. In particolare, l'impiego di nuove risorse può essere richiesto per cambiamenti nei requisiti dell'applicazione o nelle caratteristiche delle risorse durante l'esecuzione.

Nel tentativo di realizzare una soluzione alla gestione delle risorse, che fosse in grado di risolvere tutte le problematiche succitate, nell'ambito del progetto Globus, è stata progettata l'architettura schematizzata in figura A.3 che supporta, come meccanismo base, l'interfaccia *Grid Resource Allocation and Management* (GRAM).

In particolare si farà riferimento alla prima implementazione di questo servizio (nel Globus Toolkit 2.x) attualmente noto con il nome di Pre-WebService GRAM [17].

Più dettagliatamente l'architettura descritta risolve le problematiche legate all'autonomia del sito e all'eterogeneità dei sistemi di gestione locale delle risorse, introducendo delle entità chiamate *resource manager*. Quest'ultime forniscono un'interfaccia ben definita ai diversi strumenti per la gestione locale delle risorse, politiche di scheduling e meccanismi di sicurezza. In pratica permettono l'interazione con i sistemi succitati e controllano l'esecuzione dei processi sulle singole risorse.

Per le problematiche relative al controllo interattivo e l'estensibilità delle politiche è stato definito il *Resource Specification Language* (RSL), utilizzato dalle differenti componenti dell'architettura per comunicarsi i requisiti per le risorse.

I *resource broker* gestiscono il mapping tra le richieste di un'applicazione ad alto livello e le richieste agli scheduler locali. In pratica il ruolo dei resource broker è quello di analizzare le specifiche RSL di alto livello forni-

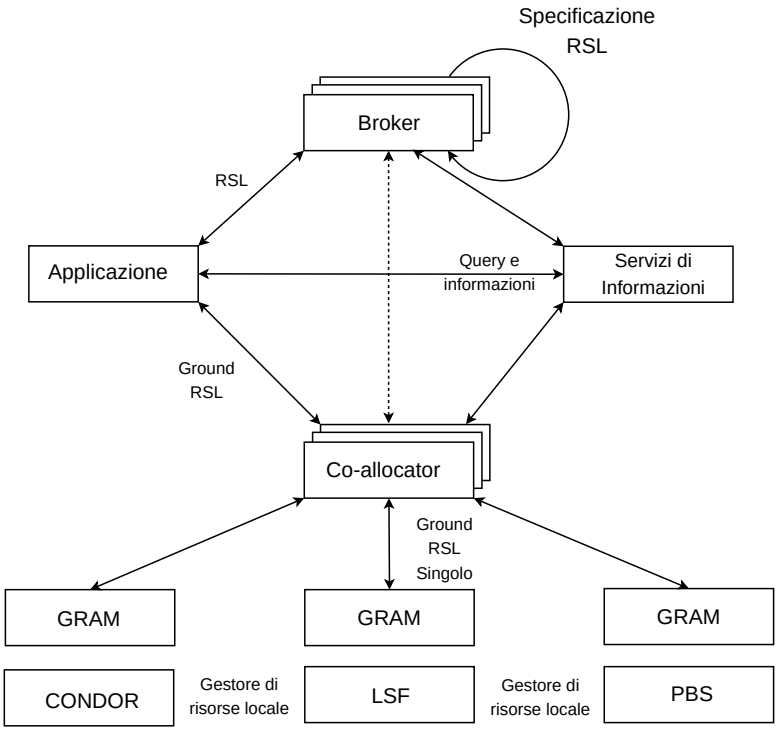

Figura A.3: Esempio di architettura di un sistema di Execution Management.

te dalle applicazioni e convertirle in *ground request*: specifiche più concrete riguardanti le risorse da utilizzare.

Infine, la problematica della co-allocazione, è gestita attraverso i *resource co-allocator*, che coordinano l'allocazione contemporanea di più risorse e l'esecuzione dei processi su tali risorse. Tali co-allocator suddividono le richieste multiple, che prevedono l'utilizzo di più risorse, in richieste singole per i vari resource manager che a loro volta interagiscono con i sistemi di gestione locali.

Nei successivi paragrafi verranno descritti:

- il linguaggio RSL;
- la libreria di co-allocazione DUROC;

- il Grid Resource Allocation and Management (GRAM).

Resource Specification Language (RSL)

La sintassi di una specifica RSL, descritta nella figura A.4, è basata sulla sintassi dei filtri di ricerca LDAP.

specification	:=	request
request	:=	multirequest
multirequest	:=	+ request-list
conjunction	:=	& request-list
disjunction	:=	| request-list
request-list	:=	(request) request-list | (request)
parameter	:=	parameter-name op value
op	:=	= | > | < | >= | <= | !=
value	:=	([a..Z][0..9][ ])+

Figura A.4: La sintassi RSL.

Una richiesta di job consiste di una specifica semplice, un componente singolo (*subjob*) o diversi subjob.

Una specifica semplice è rappresentata da una linea

parameter-name op value.

Il *parameter-name* è una stringa di caratteri (con o senza le virgolette), *op* è uno degli operatori logici come <, >, o =, *value* è una stringa di caratteri alfanumerici (con o senza le virgolette).

Un subjob consiste di diverse specifiche semplici o di diversi ulteriori subjob, connessi mediante gli operatori & per la congiunzione, | per la disgiunzione e + per l'unione.

L'insieme dei simboli *parameter-name* è estensibile: resource broker, co-allocator e resource manager possono definire un proprio insieme di nomi di parametri.

I resource manager, i componenti responsabili dell'interazione con i sistemi di scheduling locali, accettano due tipologie di nomi di parametri:

- I nomi degli attributi delle entry del Servizio di Informazioni: in questo caso il nome del parametro si riferisce al campo definito nella entry del sistema di informazioni. Vengono utilizzati per definire le specifiche sulle risorse che dovranno essere allocate (per esempio $memory >= 64$, $network = atm$, ...).

- I parametri di esecuzione per lo scheduler: utilizzati per comunicare le informazioni riguardanti il job allo scheduler, come il numero di processi richiesti, il tempo massimo di esecuzione richiesto, l'eseguibile, gli argomenti, la directory di esecuzione e le variabili di ambiente.

La *multirequest* +, permette di specificare risorse multiple (co-allocazione). L'RSL

```
+ (& (count=5)(memory>=64)
    (executable=sim.1))
  (&(network=atm) (executable=sim.2))
```

esegue cinque istanze di sim.1 su un calcolatore con almeno 64M di RAM e, contemporaneamente, esegue sim.2 su un calcolatore avente una connessione ATM.

I comandi **globusrun** e **globus-job-*** avviano la co-allocazione di una multi-richiesta usando la compomente Globus chiamata *Dynamically Updated Request Online Co-allocator* (DUROC).

L'uso combinato di resourse broker, Servizi di Informazione, e RSL rendono possibile il controllo interattivo. Questi servizi, utilizzati insieme, consentono di effettuare richieste di risorse in modo dinamico, sulla base dello stato del sistema e della negoziazione tra l'applicazione e le risorse stesse.

La libreria di co-allocazione DUROC

Quando si esegue un'applicazione su Grid può accadere che un singolo cluster non sia sufficiente da solo a fornire tutte le risorse di cui l'applicazione necessita. Perciò è necessario suddividere il job associato all'applicazione

in componenti, subjob, che vanno allocati su cluster differenti ed eseguiti contemporaneamente.

La libreria di co-allocazione DUROC fornisce un insieme di funzioni (API) che consentono di implementare il meccanismo di co-allocazione specifico dell'infrastruttura Grid di Globus. Il co-allocator è un programma di più alto livello che fa uso di queste API.

Il meccanismo di co-allocazione consiste nelle procedure tecniche che consentono la co-allocazione, e quindi la suddivisione dei job, la distribuzione dei subjob, il monitoraggio e la gestione della loro esecuzione.

Il meccanismo di co-allocazione consiste in tre fasi principali [18]:

1. fase di *allocazione* in cui vengono acquisite le risorse computazionali richieste;

2. fase di *configurazione* in cui l'applicazione viene inizializzata;

3. fase di *controllo/monitoraggio* in cui l'applicazione viene eseguita.

In figura A.5, sono rappresentate le tre fasi principali del meccanismo di co-allocazione.

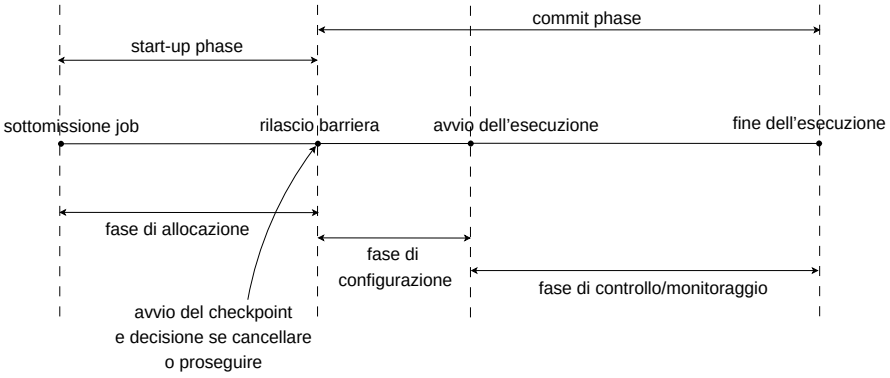

Figura A.5: Le tre fasi principali del meccanismo di co-allocazione e il meccanismo di barriera di DUROC

Fase di Allocazione Nella fase di allocazione, il co-allocator per prima cosa decompone la richiesta di job in componenti (subjob) attraverso un meccanismo di parsing. Estrae da ogni subjob gli attributi specifici di DUROC, cioè *ResourceManagerContact*, *label*, *subjobCommsType* e *subjobStartType*, aggiungendo diverse variabili d'ambiente, utili per l'esecuzione di ogni subjob sulla corrispondente risorsa.

Il co-allocator manda una richiesta di esecuzione ai vari resource manager delle risorse coinvolte e garantisce l'atomicità dell'avvio dell'esecuzione del job: se non è stato possibile allocare la risorsa richiesta per tutti i subjob, l'intero job viene cancellato. DUROC, allo scopo di garantire l'atomicità dell'avvio dell'esecuzione del job, utilizza un meccanismo di barriera che si sviluppa attraverso due fasi.

Prima fase (*start-up phase*): il co-allocator, dopo aver fatto richiesta di esecuzione, attende di ricevere da ogni resource manager un messaggio di conferma che ogni subjob è entrato nella sua barriera. Se un subjob è entrato nella sua barriera vuol dire che per esso il resource manager locale ha allocato la risorsa richiesta.

Tutti i subjob vengono rilasciati dalle loro barriere solo dopo che il co-allocator ha ricevuto messaggi da tutte le risorse richieste. L'interò job può così passare alla fase successiva.

Nel caso in cui qualche subjob non ha raggiunto la sua barriera per qualche problema (risorse insufficienti, guasti hardware, problemi di rete, ...), il co-allocator riceve per quel subjob un messaggio di fallimento, e agirà nella fase successiva seguendo una determinata politica di transazione.

Seconda fase (*commit phase*): se tutti i subjob sono stati rilasciati dalla loro barriera, allora vuol dire che l'intero job ha superato la fase di start-up con successo e può procedere alla fasi di configurazione e controllo/monitoraggio.

Se il co-allocator individua un fallimento nello start-up di un particolare subjob, agisce secondo una delle seguenti politiche di transazione:

- *Atomic transaction*: il co-allocator cancella l'intero job;
- *Interactive transaction*: il co-allocator non cancella subito il job ma

esamina il tipo di risorse richieste dal subjob in errore. Tali risorse sono classificate come *necessaria*, *interattiva* o *opzionale*. Nel primo caso viene cancellato l'intero job; nel secondo caso, le risorse non disponibili vengono rimpiazzate da altre risorse o rimosse dallo script RSL e viene processata la richiesta del job così modificata; nel terzo caso, il co-allocator ignora l'errore e la richiesta di job continua ad essere processata, pena un'eventuale diminuzione delle prestazioni.

In figura A.5, sono mostrate le due fasi succitate.

Il tipo di risorsa, *necessaria*, *interattiva* o *opzionale*, può essere specificato attraverso l'attributo RSL *subjobStartType* assegnandogli rispettivamente il valore *barrier*, *loose-barrier*, o *no-barrier*.

Il meccanismo di barriera di DUROC richiede che ogni applicazione Grid invochi la funzione barriera, cioè *globus_duroc_runtime_barrier()*. Ciò induce ogni subjob generato dall'applicazione a entrare nella barriera.

Le applicazioni che utilizzano la libreria MPICH-G2 non devono invocare tale funzione esplicitamente poichè la chiamata a tale funzione è già inclusa nella funzione *MPI_Init()*.

Per rilasciare la barriera, una volta che tutti i subjob sono entrati nelle rispettive barriere, DUROC richiede che il co-allocator invochi la funzione di rilascio barriera chiamata *globus duroc barrier release()*.

Fase di Configurazione Una volta completata con successo la richiesta di co-allocazione i resource manager locali creano, sulle risorse allocate, un insieme di processi per ogni subjob. Ogni processo è in esecuzione su un processore (CPU).

La configurazione o inizializzazione di questi processi spesso richiede che essi si rivelino e comunichino l'un l'altro. Allo scopo di conciliare una vasta gamma di possibili configurazioni è stato identificato un insieme base di operazioni che includono:

- determinazione del numero di subjob;

- determinazione della dimensione (cioè il numero di processori) di uno specifico subjob;
- capacità che almeno un processore in ogni subjob comunichi con ogni altro processore nel subjob stesso;
- capacità che almeno un processore in ogni subjob comunichi con almeno un processore di ogni altro subjob.

Fase di Controllo/Monitoraggio Dopo la fase di configurazione tutti i processi dei subjob cominciano l'esecuzione sulle risorse allocate. L'intero job non sarà completo finche tutti i processi non completeranno la loro esecuzione. Durante l'esecuzione è auspicabile poter monitorare e controllare l'esecuzione dei processi come se si trattasse di un'unica unità collettiva.

Quindi, le operazioni di controllo consentono di agire su tutti i processi dell'applicazione, come ad esempio nel caso di richiesta di interruzione dell'esecuzione dell'applicazione da parte dell'utente.

Le operazioni di monitoraggio consentono di conoscere lo stato globale dell'intero job e quello dei singoli processi e di ricevere notifiche sulle loro transizioni di stato.

Ad esempio, DUROC fornisce le API *globus_duroc_control_subjob_states()*, per monitorare i cambiamenti di stato di ogni subjob, e *globus_duroc_job_cancel()* per cancellare l'intero job.

Grid Resource Allocation and Management (GRAM)

Nel livello più basso dell'architettura di gestione delle risorse troviamo i resource manager locali la cui implementazione è chiamata Globus Resource Allocation and Management (GRAM). Il GRAM è responsabile per:

- l'elaborazione delle specifiche RSL che rappresentano le richieste di risorse che si traduce in un rifiuto della richiesta o nella creazione di uno o più processi (job) che soddisfano la richiesta;

- il monitoraggio e la gestione remota dei job creati;
- l'aggiornamento periodico del Servizio di Informazioni con informazioni riguardanti la disponibilità e le proprietà delle risorse che vengono gestite.

Inoltre risponde anche ai seguenti obiettivi:

- eseguire lo staging dei file di input/output;
- notificare il cambiamento di stato di un job;
- consentire il recupero dell'output prodotto dall'applicazione.

Il GRAM è stato progettato per fornire una interfaccia tra un vasto ambiente di metacomputing e una entità autonoma responsabile della creazione dei processi sulla singola risorsa. Questo significa che un resource manager non ha bisogno di coincidere con un singolo host, ma piuttosto con un servizio che agisce per conto di una o più risorse computazionali.

Il GRAM attraverso la specifica di una richiesta di risorse può identificare le risorse locali che la soddisfano senza ulteriori interazioni con l'entità che ha generato la richiesta. A tal scopo mappa la specifica di richiesta di risorse in una richiesta per alcuni sistemi di allocazione locale delle risorse. In tal modo l'utente, attraverso GRAM, può interagire con diversi sistemi di gestione locale senza necessariamente conoscerne il funzionamento.

I sistemi supportati sono Condor, NQE, CODINE, EASY, LSF, PBS, e LoadLeveler, ma la realizzazione di interfacce a nuovi sistemi di gestione è un compito semplice e ben documentato [46].

Le API del GRAM forniscono funzionalità per la sottomissione e la cancellazione di richieste di job, e per venire a conoscenza del tempo di attesa previsto prima dell'esecuzione di un job (già sottomesso o da sottomettere). Una volta che è stato sottomesso un job, ad esso viene associato un *job handle* unico e globale che può essere utilizzato per monitorare e controllare lo stato del job.

La struttura dell'implementazione considerata del GRAM è costituita da due componenti principali: il *GateKeeper* e il *JobManager*. Il GateKeeper, in esecuzione sulla risorsa come utente root, riceve una richiesta di esecuzione di un job e svolge tre operazioni:

- esegue mutua autenticazione tra l'utente e la risorsa;
- identifica l'utente remoto con un utente locale;
- avvia l'esecuzione come utente locale di un JobManager che si occupa di gestire la richiesta sulla risorsa.

Le prime due operazioni vengono eseguite attraverso chiamate alla libreria GSI.

Il JobManager si occupa di avviare l'esecuzione dei processi richiesti dall'utente. Per consentire l'esecuzione dei processi, il JobManager si occupa di reperire l'eseguibile, i file di input ed i file di dati specificati dall'utente. Successivamente sottomette la richiesta di esecuzione allo scheduler locale.

Una volta che i processi sono stati creati, il JobManager si occupa di monitorarne lo stato, e di comunicarne su richiesta la transizione degli stati.

Il JobManager implementa anche operazioni di controllo, come la terminazione dei processi, e si occupa dello staging dei file di output, consentendo il recupero dell'output prodotto dall'applicazione.

Riassumendo, l'esecuzione del job passa attraverso i seguenti stati:

- Unsubmitted: il job non è stato ancora sottomesso allo scheduler.
- StageIn: il JobManager sta reperendo il file eseguibile, il file di input o eventuali file di dati necessari per l'esecuzione del job.
- Pending: il job è stato sottomesso allo scheduler, ma questi non ha ancora allocato risorse per la sua esecuzione.
- Active: il job è in esecuzione.

- Suspended: lo scheduler ha temporaneamente sospeso l'esecuzione del job.
- StageOut il JobManager sta restituendo i file di output all'utente.
- CleanUp il JobManager fa richiesta di cancellazione dei file/directory e del file proxy utente.
- Done: il job è stato completato.
- Failed: il job è terminato prima di essere stato completato.

A.2 MPICH-G2

MPICH-G2 è un'implementazione *grid-enabled* dello standard MPI-1 che consente agli utenti di eseguire programmi MPI su diversi calcolatori, non necessariamente localizzati presso lo stesso sito, come se stessero utilizzando lo stesso calcolatore parallelo.

MPICH-G2 estende l'implementazione MPICH [11] di MPI, sviluppata in Argonne National Laboratory [1] in collaborazione con Mississippi State University [3], in modo da utilizzare i servizi forniti dal Globus Toolkit per l'autenticazione degli utenti, l'allocazione delle risorse, lo staging dell'eseguibile, l'accesso remoto a file, la gestione dell'I/O e la creazione, monitoraggio e controllo dei processi.

MPICH-G2, attraverso i servizi del Globus Toolkit, rende trasparenti all'utente le problematiche legate all'eterogeneità delle risorse e alla loro dislocazione in domini differenti [38].

La libreria consente di sfruttare i costrutti MPI per gestire le prestazioni della propria applicazione MPI.

Ad esempio il costrutto **MPI_Communicator** può essere utilizzato per organizzare i processi in una struttura multilivello che rifletta la loro posizione rispetto alla topologia di rete dell'ambiente Grid. Adattando i processi alla topologia di rete multilivello dell'ambiente Grid, si consente all'applicazione

di scegliere di volta in volta il canale di comunicazione più efficiente tra quelli disponibili.

Nei successivi paragrafi verrà descritto come MPICH-G2:

- estende l'implementazione MPICH utilizzando il Globus Toolkit;
- consente l'esecuzione di programmi MPI;
- utilizza i servizi del Globus Toolkit, nelle fasi di startup e management dei processi, nascondendo le problematiche legate all'eterogeneità e distribuzione geografica delle risorse;
- organizza e rende accessibile agli utenti le informazioni sulla topologia di rete dell'ambiente Grid;
- gestisce le comunicazioni tra i processi.

A.2.1 MPICH-G2 estensione dell'implementazione MPICH

MPI è lo standard di fatto più diffuso tra le librerie di message passing per il calcolo scientifico e l'HPC.

La sua implementazione più popolare è MPICH che deve il suo successo ad un'ampia portabilità e alla distribuzione gratuita.

L'ampia portabilità deriva dalle interfacce e dall'architettura a livelli (figura A.6).

All'estremità superiore di tale architettura c'è l'interfaccia MPI così come definita dallo standard MPI.

Immediatamente sotto c'è il livello MPICH che implementa l'interfaccia MPI. La maggior parte del codice in un'implementazione MPI è indipendente dal dispositivo di rete o dal sistema di gestione dei processi. Questo codice che include il controllo degli errori e la manipolazione dei vari costrutti come **MPI_Communicator**, **MPI_Type**, ecc. è implementato direttamente nel livello MPICH.

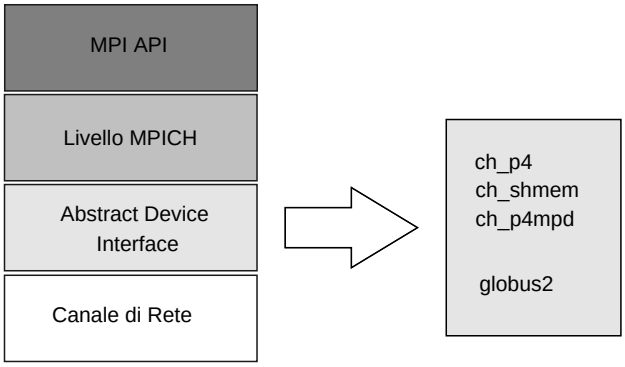

Figura A.6: Architettura a livelli di MPICH.

Il resto delle funzionalità è delegato ai livelli più bassi.

L'*Abstract Device Interface* (ADI) è un'interfaccia più semplice rispetto a quella MPI situata nell'architettura a livelli tra il livello MPICH e il sottosistema di rete. Su tale interfaccia sono costruite tutte le funzioni al livello MPICH indipendentemente dal sistema di rete sottostante. Tutte le interfacce ADI forniscono all'esterno le stesse funzionalità ma per implementarle utilizzano un particolare hardware o canale di comunicazione.

Quindi per ottenere un'implementazione completa di MPI su una particolare piattaforma è sufficiente definire le routine nell'ADI.

Ad esempio il dispositivo *ch_p4* è l'implementazione dell'ADI per cluster di PC omogenei, LAN di workstation, workstation singole. Il dispositivo *ch_shmem* è quella per i processori simmetrici a memoria condivisa.

MPICH-G2, quindi, non è altro che una particolare implementazione dell'ADI, chiamata *globus2*, della libreria MPICH, che utilizza il Globus Toolkit come canale di comunicazione in ambiente Grid.

A.2.2 Usare MPICH-G2

Una delle principali funzionalità di MPICH-G2 è quella di consentire l'esecuzione di programmi MPI in ambiente Grid senza richiedere alcuna modi-

fica del codice attraverso comandi simili (ed in alcuni casi identici) a quelli utilizzati per un singolo calcolatore parallelo.

MPICH-G2 fornisce gli strumenti per la compilazione e l'esecuzione propri di MPICH, tra i quali il comando **mpirun** per l'esecuzione dei programmi ed i wrapper per la compilazione **mpicc**, **mpif77**, **mpif90**, ecc.

Una volta compilato il programma, esso può essere eseguito utilizzando il comando **mpirun** specificando il numero di processi, il nome del *machinefile*, il nome dell'eseguibile e gli argomenti. Il *machinefile* deve contenere un elenco degli host che si vogliono utilizzare per l'esecuzione del programma con il numero di nodi che si intende utilizzare per ogni calcolatore.

La sintassi del comando **mpirun** è la seguente:

```
% /usr/local/mpich-g2/bin/mpirun -np <numero di processi> \
  -machinefile <nome del machinefile> <nome del programma> \
  [argomenti]
```

L'utilizzo del comando **mpirun** non richiede nessuna conoscenza di comandi e protocolli del Globus Toolkit: tutto quello che è necessario per avviare l'esecuzione in ambiente Grid avviene in maniera totalmente trasparente all'utente.

Il comando **mpirun** eseguito con il dispositivo *globus2* genera uno script in Resource Specification Language (RSL) (vedi paragrafo A.1.4) dove sono indicate le risorse Grid utilizzate per l'esecuzione e per ognuna di esse viene descritto un *subjob* che contiene alcuni parametri: il numero di processi, la directory di lavoro, locazioni remote e nome degli eseguibili, variabili di ambiente, argomenti di linea di comando.

Esistono due modalità per gestire lo script RSL a livello utente: utilizzarlo in maniera implicita attraverso il comando **mpirun**; generare lo script senza proseguire con l'esecuzione del programma apportando delle modifiche nelle parti in cui è necessario per fornire ulteriori dettagli. La sintassi da utilizzare in questo caso è:

```
% /usr/local/mpich-g2/bin/mpirun -dumprsl -np <numero di processi> \
```

```
-machinefile <nome del machinefile> <nome del programma> \
[argomenti] > nomefile.rsl
```

Modificando lo script RSL è possibile sfruttare alcune funzionalità particolari di MPICH-G2. È possibile:

- Specificare un'interfaccia di rete o un insieme di interfacce con la variabile di ambiente

  MPICH_GLOBUS2_USE_NETWORK_INTERFACE.

- Specificare il range di porte TCP/IP da utilizzare per le comunicazioni con la variabile di ambiente

  GLOBUS_TCP_RANGE.

  Questa funzionalità risulta particolarmente utile in presenza di firewall quando solo un certo range di porte è abilitato ad accettare connessioni dall'esterno.

- Regolare la dimensione del buffer TCP attraverso la variabile di ambiente

  MPICH_GLOBUS2_TCP_BUFFER_SIZE.

- Specificare che alcuni subjob vengono eseguiti su calcolatori appartenenti alla stessa LAN utilizzando la variabile di ambiente

  GLOBUS_LAN_ID.

- Indicare il nome dell'host attraverso la variabile di ambiente

  GLOBUS_HOSTNAME.

Una volta creato lo script RSL è possibile lanciare l'esecuzione con:

```
% /usr/local/mpich-g2/bin/mpirun -globusrsl nomefile.rsl
```

oppure

```
% globusrun -w -f nomefile.rsl
```

A.2.3 Startup e management dei processi

In questo paragrafo verrà descritto come MPICH-G2 nelle fasi di startup e management dei processi utilizza i servizi del Globus Toolkit per nascondere le problematiche legate all'eterogeneità delle risorse e alla loro dislocazione in siti differenti.

Come illustrato in figura A.7, MPICH-G2 utilizza vari servizi del Globus Toolkit per eseguire le complesse operazioni necessarie all'esecuzione di programmi MPI in ambiente Grid.

Operazioni come l'autenticazione sui diversi siti, l'interfacciamento con sistemi di scheduling differenti con caratteristiche differenti, la redirezione dell'output, l'avvio coordinato dei processi, il controllo dei processi, le comunicazioni eterogenee tra i processi, sono completamente trasparenti all'utente.

Per prima cosa è necessario che l'utente ottenga un certificato proxy che gli consentirà di essere autenticato su ogni calcolatore di ogni sito. A tale scopo l'utente utilizza il servizio Grid Security Infrastructure (GSI) attraverso il comando **grid-proxy-init**. Questo passo fornisce capacità di *single sign-on* (vedi paragrafo A.1.1).

L'utente può anche avvalersi del Monitoring and Discovery Service (MDS) per la selezione delle risorse da utilizzare (vedi paragrafo A.1.3).

Una volta autenticato, l'utente può lanciare l'esecuzione del programma MPI utilizzando il comando **mpirun**. L'implementazione in MPICH-G2 di questo comando utilizza il Resource Specification Language (RSL) per descrivere il job da allocare. In sostanza l'utente utilizza uno script RSL che identifica le risorse e specifica i requisiti (numero di CPU, memoria, tempo di esecuzione,...) e i parametri (locazione degli eseguibili, variabili di ambiente, argomenti di linea di comando,...) del programma MPI.

Lo script RSL viene utilizzato come argomento del comando **globusrun** attraverso il quale viene invocata la libreria di co-allocazione Dinamically-Updated Request Online Coallocator (DUROC) (vedi paragrafo A.1.4) per la sincronizzazione della fase di startup sui diversi calcolatori specificati dall'utente. La libreria DUROC a sua volta utilizza le API e il protocollo Grid

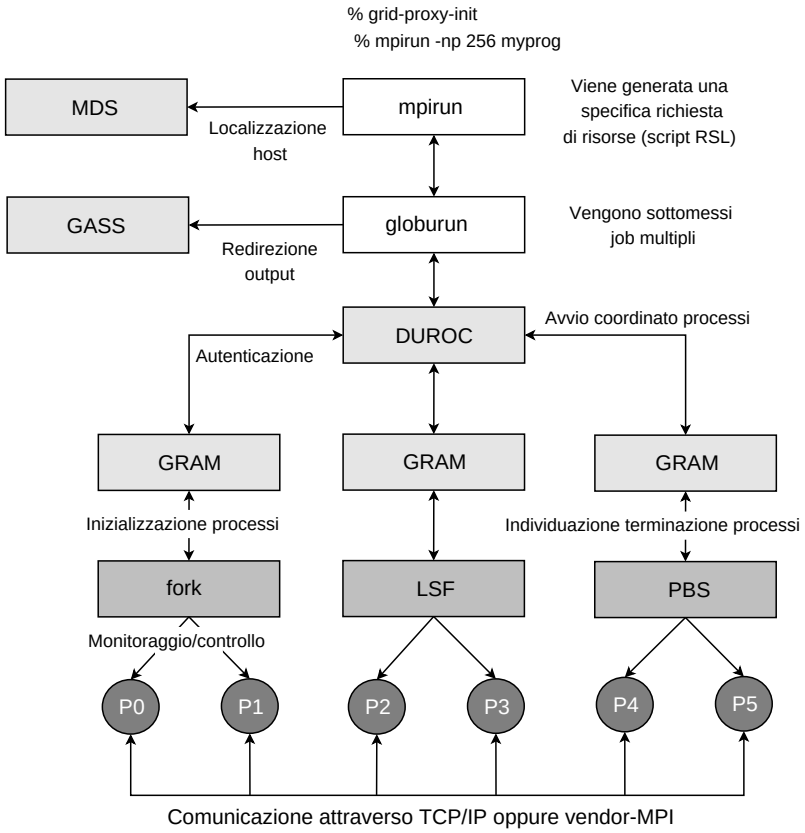

Figura A.7: MPICH-G2: utilizzo dei componenti del Globus Toolkit, per rendere trasparenti all'utente le problematiche connesse all'eterogeneità delle risorse nelle fasi di startup e gestione dei processi. Fork, LSF e PBS sono differenti scheduler locali.

Resource Allocation and Management (GRAM) (vedi paragrafo A.1.4) per l'esecuzione dei processi sui diversi calcolatori tramite interazione con i sistemi di scheduling locali. In particolare GRAM inizializza e gestisce un insieme di sotto-computazioni, una per ogni calcolatore. Per ognuna delle sotto-computazioni, DUROC genera delle richieste per GRAM che, una volta autenticato l'utente, crea per esso delle credenziali locali temporanee e interagisce con lo scheduler locale per avviare la computazione.

Se specificato nello script RSL, GRAM utilizzerà il Global Access to Secondary Storage (GASS) (vedi paragrafo A.1.2) per rendere disponibili ai processi gli eseguibili dalle locazioni remote in cui essi si trovano. Tali locazioni sono indicate nello script da indirizzi URL. GASS si fa carico, inoltre, di reindirizzare gli stream di standard output e standard error verso il terminale dell'utente.

Durante l'esecuzione MPICH-G2 seleziona il canale di comunicazione più efficiente tra ogni coppia di processi in base alla topologia: il TCP utilizzando i servizi del Globus Toolkit; oppure tramite un'eventuale implementazione proprietaria di MPI fornita con il calcolatore (*vendor-MPI*) (vMPI).

DUROC e GRAM interagiscono per controllare e gestire l'esecuzione dei processi sui vari calcolatori.

Ogni server GRAM prima di avviare la computazione attende un segnale di sblocco da DUROC che si occupa di sincronizzare l'avvio: ogni processo viene bloccato con una barriera DUROC attraverso l'esecuzione della funzione *MPI_Init()* (funzione di inizializzazione dell'ambiente MPI invocata da ogni processo) e verrà rilasciato solo quando tutti gli altri processi avranno raggiunto la stessa barriera.

I server GRAM, in esecuzione sulle varie risorse computazionali, comunicano le varie transizioni di stato dei processi sotto il proprio controllo mentre la libreria DUROC gestisce le interruzioni della computazione richieste dall'utente.

A.2.4 Topologia

MPICH-G2 utilizza le informazioni dello script RSL per organizzare i processi in una struttura multilivello che riflette la topologia di rete sottostante dell'ambiente Grid: ad ogni livello corrisponde un canale di comunicazione.

Il livello 0 è il TCP su WAN, il livello 1 è il TCP su LAN, il livello 2 è il TCP sulla rete interna dei calcolatori, il livello 3 è la libreria MPI proprietaria.

Ad ogni processo in **MPI_COMM_WORLD** è assegnato un valore di

profondità della topologia che corrisponde al numero di canali di comunicazione su cui il calcolatore su cui è in esecuzione può comunicare. Ad esempio, se un calcolatore è dotato di un'implementazione MPI proprietaria i processi in esecuzione su di essi avranno profondità 4.

MPICH-G2 utilizza le profondità per raggruppare i processi ad un determinato livello assegnando loro un *colore* (un intero che ha sempre valore maggiore o uguale a zero): due processi hanno lo stesso colore ad un determinato livello se possono comunicare tra di loro attraverso il canale di comunicazione corrispondente.

MPICH-G2 assegna i colori a partire dalle informazioni specificate nello script RSL utilizzando le seguenti regole: due processi qualsiasi possono sempre comunicare tra loro al livello 0 e quindi tutti i processi hanno lo stesso colore a quel livello, i processi possono comunicare (hanno lo stesso colore) al livello 1 se e solo se appartengono alla stessa LAN (hanno lo stesso **GLOBUS_LAN_ID** nello script RSL), i processi hanno lo stesso colore al livello 2 se e solo se sono nello stesso subjob nello script RSL, i processi hanno lo stesso colore al livello 3 se e solo se sono nello stesso subjob nello script RSL e per quel subjob è specificato ($jobtype = mpi$).

L'assegnazione delle profondità e dei colori non tiene però conto dell'eventuale presenza di calcolatori nei quali non tutti i nodi hanno accesso agli stessi canali di comunicazione. È questo il caso dei cluster a rete privata nei quali c'è generalmente un solo nodo che ha accesso alle reti WAN e LAN, il *front-end*, mentre gli altri nodi hanno accesso alla sola rete privata del cluster. Solo i front-end pertanto hanno accesso a tutti e tre i canali TCP (WAN, LAN, rete interna) e non anche i nodi interni così come risulterebbe invece dalle informazioni topologiche ricostruite da MPICH-G2. I nodi interni dei cluster a rete privata hanno accesso al solo canale TCP sulla rete interna e pertanto non dovrebbero avere profondità 3.

MPICH-G2 rende accessibile agli utenti le informazioni sulla topologia attraverso gli attributi

MPICHX_TOPOLOGY_DEPTHS

ed

MPICHX_TOPOLOGY_COLORS

associati ad ogni comunicatore.

L'attributo **MPICHX_TOPOLOGY_DEPTHS** è un vettore di interi in cui l'i-mo elemento è la profondità del processo che ha rango i nel comunicatore. L'attributo **MPICHX_TOPOLOGY_COLORS** è un vettore di puntatori ad interi nel quale l'i-mo elemento è a sua volta un puntatore ad un vettore di interi (di lunghezza pari alla profondità del processo di rango i) e i cui elementi sono i colori del processo di rango i ad ogni livello.

Tramite questo meccanismo di "scoperta" della topologia, l'utente ha la possibilità di ricostruire la struttura multilivello del sistema di rete dell'ambiente Grid e organizzare le comunicazioni in modo da sfruttare maggiormente i canali di comunicazione più efficienti.

A.2.5 Gestione delle Comunicazioni

MPICH-G2 utilizza due metodi per le comunicazioni: il protocollo TCP e la libreria MPI (quando esiste) presente sui calcolatori.

Se due processi sono in esecuzione sullo stesso calcolatore, dotato di una propria libreria MPI, MPICH-G2 utilizzerà tale libreria per le comunicazioni tra i due processi, in caso contrario verrà utilizzato il TCP. Nel primo caso, MPICH-G2 mappa le funzioni di comunicazione (*MPI_Send()*, *MPI_Recv()*,...) su quelle della libreria vMPI. Questo consente, quando si eseguono programmi in locale sui singoli calcolatori, di ottenere prestazioni praticamente identiche a quelle di un'implementazione non grid-enabled (vMPI o MPICH configurato con il dispositivo TCP ch_p4), lasciando comunque aperta la possibilità di accedere a risorse remote tramite l'infrastruttura Grid.

La verifica da parte di un processo della presenza di messaggi TCP in ingresso è un'operazione relativamente costosa. MPICH-G2 minimizza il numero di queste operazioni utilizzando le informazioni relative al contesto nel

quale l'applicazione invoca le funzioni di comunicazioni (sorgente del messaggio e stato del processo che riceve). Tali informazioni influenzano il modo in cui MPICH-G2 implementa le funzioni stesse.

Specificatamente, per processi in esecuzione sullo stesso calcolatore parallelo dotato di una propria libreria MPI, si distinguono tre modalità differenti per l'implementazione della funzione *MPI_Recv()*.

- **Specificato**: il processo che invia i dati è in esecuzione sullo stesso calcolatore parallelo del processo che riceve (i due processi appartengono allo stesso job mpi) e concorrentemente non c'è nessuna comunicazione non bloccante pendente (nessuna chiamata ad *MPI_Irecv()* o *MPI_Isend()* ancora non completata). In questo caso viene invocata direttamente la corrispondente routine vMPI.

- **Specificato con pendenze**: il processo che invia i dati è in esecuzione sullo stesso calcolatore parallelo ma ci sono anche una o più comunicazioni non bloccanti pendenti da o verso processi in esecuzione sullo stesso calcolatore. In questo caso viene invocato periodicamente *MPI_Iprobe()* per la verifica dei messaggi in ingresso sul canale vMPI.

- **Metodo multiplo**: il processo che invia i dati non è specificato (viene utilizzato **MPI_ANY_SOURCE**) o ci sono una o più comunicazioni non bloccanti pendenti da o verso processi in esecuzione su altri calcolatori. In questo caso MPICH-G2 deve verificare periodicamente la presenza di messaggi in ingresso sia sul canale TCP che su quello vMPI.

L'implementazione delle funzioni di comunicazioni non contempla comunque il caso in cui i processi coinvolti siano in esecuzione su nodi di calcolatori che non hanno accesso alla rete pubblica.

L'assenza di un meccanismo di smistamento di messaggi tra le reti private, limita di fatto il numero di applicazione che possono essere portate da un calcolatore parallelo convenzionale ad un sistema Grid costituito da più calcolatori paralleli, senza modificare il codice sorgente.

In MPICH le funzioni di comunicazione collettive sono implementate sulla base dell'assunzione che i processi siano "equidistanti" l'uno dall'altro. Questa assunzione non è più ovviamente valida in ambiente Grid.

MPICH-G2 è in grado di ricavare le informazioni sulla topologia del sistema Grid dagli script RSL e di utilizzare tali informazioni per implementare le funzioni di comunicazione collettive allo scopo di minimizzare le comunicazioni sui canali più lenti.

Oltre al colore, MPICH-G2 utilizza un'altra informazione per descrivere la topologia: l'*identificativo di raggruppamento*. Per raggruppamento si intende l'insieme dei processi che possono comunicare tra loro ad ogni livello attraverso il canale di comunicazione corrispondente. All'interno di ogni raggruppamento, ad ogni livello viene individuato un processo *master* (il "rappresentante" del raggruppamento per quel livello).

Si consideri la funzione di broadcast, nella quale un processo (radice) manda un messaggio a tutti i processi contenuti in un comunicatore.

L'implementazione di MPICH-G2 di tale funzione è strutturata in tre fasi: nella prima fase la radice invia il messaggio di broadcast ai rappresentati dei vari siti; nella seconda fase, presso ogni sito, il master invia il messaggio di broadcast a tutti i rappresentati dei calcolatori dello stesso sito; nella terza ed ultima fase, i master dei vari calcolatori inviano il messaggio a tutti i nodi dei rispettivi calcolatori.

Il vantaggio di questa strategia sta nella riduzione del numero di comunicazioni sui canali più lenti e nella possibilità di utilizzare diversi algoritmi di broadcast sui vari canali.

A.3 Condor e Condor-G

Per molti scienziati la qualità della ricerca dipende fortemente dalla potenza di calcolo loro offerta. Una chiave fondamentale per ottenere tale potenza è l'utilizzo delle risorse disponibili.

A tale scopo il sistema Condor implementa due importanti funzionalità

come l'utilizzo di workstation inattive e l'organizzazione distribuita dei job su diverse risorse dedicate, come cluster o insiemi di cluster.

Il software Condor è composto da due parti. La prima parte si occupa della gestione dei job: consente la sottomissione di nuovi job, la richiesta di informazioni sullo stato dei job, la gestione dei file di input e output. La seconda parte si occupa invece della gestione delle risorse: alloca le risorse disponibili in funzione delle richieste degli utenti.

Condor-G contiene la prima parte di software per la gestione dei job di Condor e utilizza il Globus Toolkit per la gestione delle risorse e per avviare l'esecuzione dei job sulle risorse remote.

Nei successivi paragrafi verranno descritti il sistema Condor e la sua estensione Condor-G per la gestione dell'elaborazione di Condor alle Griglie multi-istituzionali.

A.3.1 Condor

Condor è un particolareggiato sistema di gestione del carico di lavoro per applicazioni computing-intensive in ambienti di calcolo distribuito. Esso è il frutto di più di dieci anni di ricerca e sviluppo condotti nell'ambito del *Condor Research Project* [14] al *Computer Sciences Department* [10] dell'Università del Wisconsin-Madison.

Il sistema Condor, al pari degli altri sistemi *batch* completi, offre meccanismi efficienti per la gestione delle code e politiche di scheduling, gestione delle priorità, monitoraggio e gestione delle risorse.

Gli utenti sottomettono i propri job batch (paralleli o sequenziali) al sistema Condor il quale, li posiziona nella coda, si occupa di individuare e allocare le risorse su cui vengono eseguiti, ne monitora l'andamento ed infine notifica l'utente sull'esito della loro esecuzione.

Alcuni principi fondamentali alla base dello sviluppo del sistema Condor sono:

- L'esecuzione del sistema batch Condor non ha nessuna forte influenza sulla disponibilità e la qualità del servizio delle risorse nei confronti

dei proprietari delle risorse stesse: quest'ultimi hanno priorità assoluta sui propri calcolatori ottenendone l'uso esclusivo quando occorre. Condor gestisce questa priorità automaticamente ed in modo del tutto trasparente sia agli utenti Condor, sia ai proprietari delle risorse.

- Non è richiesta nessuna particolare tecnica di programmazione per l'utilizzo di Condor, il quale preserva l'ambiente operativo della risorsa sulla quale è stato sottomesso un job.

- Se una risorsa su cui è in esecuzione un job di un utente Condor non è più disponibile (ad esempio il proprietario della workstation su cui il job è in esecuzione ne reclama l'uso dedicato), Condor, in maniera trasparente, salva lo stato corrente del job (*checkpoint*) e ne ripristina l'esecuzione, appena possibile, su un'altra risorsa disponibile.

- Condor non richiede la presenza di un *filesystem* condiviso: se quest'ultimo non è presente, Condor può trasferire i file dati del job per conto dell'utente, e/o può redirigere in maniera trasparente tutte le richieste di I/O del job da/verso il calcolatore da cui il job è stato sottomesso.

Condor permette sia di gestire cluster di computer dedicati (come cluster Beowulf), sia di sfruttare la potenza computazionale di workstation inattive. Come risultato, Condor permette di mettere insieme tutta la potenza computazionale di un'organizzazione in un'unica risorsa.

Il sistema Condor per espletare le proprie funzionalità si avvale dei seguenti processi:

- *Master daemon*: demone principale che sovraintende tutti i demoni in esecuzione sui calcolatori del Condor *pool* (insieme di risorse/host usate dal sistema Condor);

- *Schedd (scheduler) daemon*: in esecuzione su ogni calcolatore del Condor pool, gestisce la coda permanente dei job; notifica il sistema delle richieste di risorse dei job sottomessi; si occupa di contattare le risorse disponibili e di spedire loro i job.

- *Startd (starter) daemon*: in esecuzione su ogni calcolatore del Condor pool, si prende carico dell'esecuzione, gestione e monitoraggio dei job assegnati dal sistema a quel calcolatore e notifica periodicamente il sistema sulle caratteristiche e disponibilità del calcolatore stesso.

- *Negotiator daemon*: demone responsabile di tutte le negoziazioni fra la richiesta di risorse dei job e l'offerta del pool, alla ricerca dei possibili matching (compatibilità tra richieste e offerte), consentendo la corretta allocazione dei job.

- *Collector daemon*: demone responsabile della raccolta di tutte le informazioni sulle caratteristiche e lo stato dei calcolatori del Condor pool.

Ci si riferisce al *Central Manager* come l'unico processo logico che racchiude il Negotiator e il Collector.

Il meccanismo denominato *ClassAd* (*Classified Advertisements*) [53] offre uno strumento estremamente flessibile e significativo per rappresentare le caratteristiche delle risorse disponibili e dei job. Tale meccanismo consente a Condor di fare il *matchmaking* [48], cioè di effettuare i matching fra le richieste di risorse (requisiti dei job Condor specificati dall'utente) e le risorse offerte (attributi delle risorse del Condor pool, come memoria disponibile, tipo di CPU e velocità, carico medio corrente).

Attraverso le ClassAd è possibile specificare requisiti e preferenze sia per i job da sottomettere, sia per i calcolatori con riferimento ai job che possono eseguire. In generale le ClassAd non sono vincolate ad uno schema specifico. Gli utenti possono estenderle come vogliono per rappresentare le risorse e i job. Come risultato, le informazioni ricavabili dalle ClassAd possono essere utilizzate per il brokering e il matchmaking dei job sottomessi alla Griglia.

Per sottomettere un job a Condor, si possono individuare i seguenti quattro passi:

- Scelta dell'universo: un *universo* in Condor definisce un particolare

ambiente operativo nel quale saranno eseguiti i job. Gli universi di Condor sono:

  - Standard: consente il checkpointing e consente chiamate a sistema remote; per preparare un programma che possa essere eseguito in questo universo, si dovrà provvedere al *re-link* attraverso il comando **condor_compile** (non è quindi necessario modificare il codice sorgente).
  - Vanilla: viene di solito utilizzato per programmi che non possono essere *re-linkati* o per script di shell; in questo universo non è possibile effettuare chiamate a sistema remote e non è consentito il checkpointing.
  - PVM: fornisce il supporto per l'esecuzione di programmi PVM (Parallel Virtual Machine);
  - MPI: fornisce il supporto per l'esecuzione di programmi MPI (in particolare MPICH);
  - Globus: fornisce un'interfaccia standard per eseguire job in un sistema Globus attraverso Condor (Condor-G);
  - Java: fornisce il supporto per i programmi scritti per la Java Virtual Machine (JVM).
  - Scheduler: consente al job di essere eseguito immediatamente sul calcolatore sulla quale è stato sottomesso; il job non aspetta di essere matchato ad una risorsa.

- Preparazione del job batch: un job sottomesso a Condor deve essere in grado di girare in background senza input/output interattivo. Condor può redirigere l'output della console (stdout e stderr) o l'input da tastiera (stdin) verso file specificati. È quindi necessario preparare i file dati che rappresentano l'input del programma e che verranno inviati all'host che eseguirà il job dopo che questi è stato sottomesso al sistema Condor.

- Creazione del *submit description file*: creazione di un file di testo nel quale è possibile specificare tutte le informazioni necessarie per la sottomissione del job quali l'universo, il nome dell'eseguibile, i file associati all'input/output, il file di log, i requisiti e le preferenze. In figura A.8 è mostrato un esempio di submit description file.

```
universe = MPI
executable = myjob.mpi
output = myjob.out.$(Node)
error = myjob.err.$(Node)
log = myjob.log
machine_count = 12
requirements = (HasMPI == ‘‘TRUE’’) && (Arch == ‘‘INTEL’’) &&
               (OpSys == ‘‘LINUX’’)
rank = 10*Mips + 2*KFlops + 100*Memory
queue
```

Figura A.8: Esempio di submit description file.

- Sottomissione del job: il job viene sottomesso utilizzando il comando **condor_submit** e come suo argomento il submit description file appena creato:

  ```
  % condor_submit my_job.desc
  ```

  Tale comando analizza il submit description file, verifica che non contenga errori e crea una ClassAd che descrive il job.

A.3.2 Condor-G

Condor-G offre un potente strumento per interfacciare il sistema Condor con risorse Grid, consentendo quindi agli utenti di integrare risorse appartenenti a più domini come se appartenessero ad un unico dominio personale.

Condor-G combina i protocolli (di security, comunicazioni, resource discovery, accesso a risorse in ambienti multi-dominio) implementati dal Glo-

bus Toolkit e la gestione dell'elaborazione e l'organizzazione distribuita delle risorse in un singolo dominio amministrativo, forniti da Condor.

Tutte le attività di sottomissione, gestione del job, gestione dell'I/O ed argomenti vengono svolte attraverso i comandi standard di Condor quali ad esempio **condor_submit**.

In figura A.9, viene mostrato come Condor-G interagisce con i protocolli Globus.

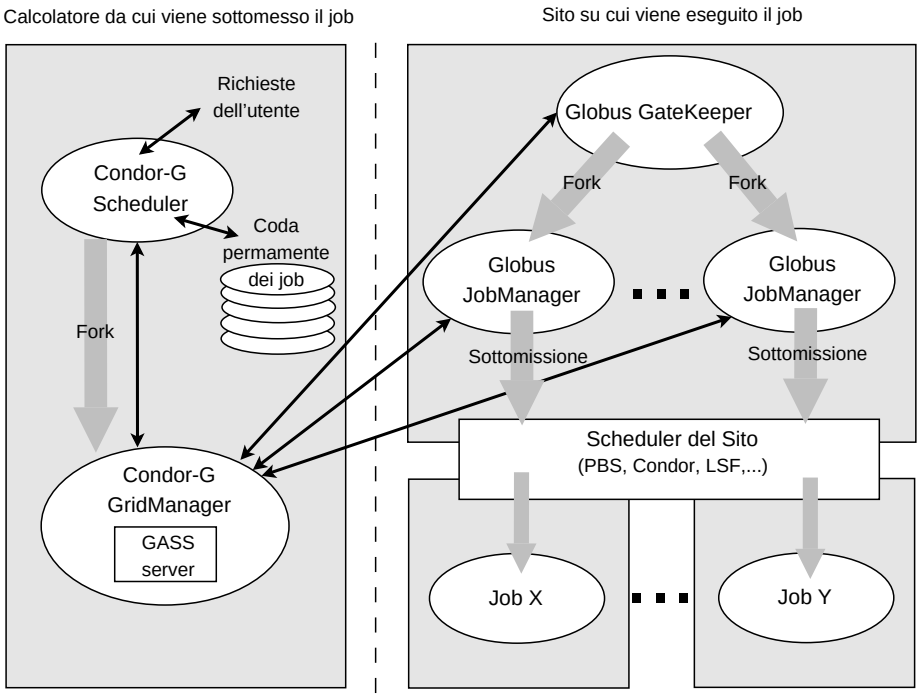

Figura A.9: Esecuzione remota su risorse gestite da Globus attraverso Condor-G.

Per l'accesso alle risorse remote vengono utilizzati i protocolli definiti dal Globus Toolkit:

- GSI (vedi paragrafo A.1.1) per l'autenticazione/autorizzazione di tutte le richieste sui siti remoti;

- GASS (vedi paragrafo A.1.2) utilizzato per trasferire l'eseguibile, lo standard input, lo standard output e lo standard error da/verso il sito di esecuzione remoto;

- GRAM utilizzato per contattare il GateKeeper remoto (vedi paragrafo A.1.4), richiedendo la sottomissione d un nuovo job, e conseguentemente per monitorarne e controllarne l'esecuzione.

Di seguito vengono schematizzati i punti fondamentali dell'implementazione di Condor-G:

- Lo *Scheduler* risponde alla richiesta di sottomissione di un job di un utente.

- Crea un nuovo demone *GridManager* per eseguire e gestire il job. Ogni processo GridManager gestisce tutti i job di un singolo utente e termina quando tutti i job di quell'utente sono completati.

- Ogni richiesta di esecuzione di job al GridManager crea un demone Globus JobManager (vedi paragrafo A.1.4).

- I JobManager comunicano con il GridManager per trasferire gli eseguibili dei job e per i dati di I/O.

- Il JobManager sottomette i job per l'esecuzione allo scheduler locale del sito.

- Il JobManager invia gli aggiornamenti sullo stato dei job al GridManager e quindi allo Scheduler Condor-G.

Condor-G è in grado di rilevare e gestire alcuni fallimenti come il crash del Globus JobManager, del calcolatore che gestisce la risorsa remota (per esempio il GateKeeper e/o il JobManager), del calcolatore su cui è in esecuzione il GridManager (o crash del GridManager), e i fallimenti nei collegamenti di rete tra i calcolatori coinvolti.

I guasti vengono rilevati dal GridManager che periodicamente controlla tutti i JobManager. Se un JobManager non risponde, controlla il GateKeeper in esecuzione sullo stesso sito: se il GateKeeper risponde, vorrà dire che il JobManager è crashato, altrimenti, vorrà dire che il sito ha avuto un crash oppure sono sopraggiunti dei problemi di rete.

Se il JobManager è crashato, il GridManager tenta di eseguire un nuovo JobManager. Se non c'è contatto con il sito remoto, il GridManager attende fino a quando non riesce a stabilire il contatto. Una volta stabilito il contatto, tenta di connettersi con il JobManager. In caso di connessione riuscita, il JobManager controlla i job in esecuzione, comunicandone lo stato al GridManager. Se il GridManager non riesce a connettersi al JobManager, crea un nuovo JobManager, che controlla i job in esecuzione.

Al fine di gestire i casi di crash locale, lo stato dei job è memorizzato in modo persistente nella coda dei job dello Scheduler Condor-G. Dopo un crash locale, il GridManager riparte, cercando di riconnettersi a qualsiasi JobManager in esecuzione al momento del crash.

Nel paragrafo successivo verrà descritto come sottomettere un job a Globus attraverso Condor-G.

Esecuzione dei job nell'universo Globus

Per sottomettere job a Globus attraverso Condor è necessario disporre delle giuste credenziali: attraverso un certificato X.509 viene creato un proxy che fornirà l'autorizzazione all'uso delle risorse Globus (vedi paragrafo A.1.1).

Per sottomettere il job a Condor utilizzando l'universo Globus si utilizza il comando **condor_submit** con argomento un opportuno submit description file.

Un esempio di tale submit description file è illustrato in figura A.10.

In questo esempio, l'eseguibile del programma (specificato dal comando **executable** sarà trasferito dal calcolatore locale al sito remoto; l'eseguibile dovrà quindi essere compilato per l'architettura di destinazione.

Specificando l'universo Globus il job sarà inviato al JobManager remo-

```
executable = myjob
globusscheduler = beocomp.dma.unina.it/jobmanager
universe = globus
output = myjob.out
error = myjob.err
log = myjob.log
```

Figura A.10: Esempio di submit description file per l'universo Globus.

to specificato dal comando **globusscheduler**. Condor trasferirà quindi il risultato prodotto dall'esecuzione del job sul sito remoto, al file myjob.out (specificato dal comando **output**) memorizzato sul calcolatore locale, registrando tutte le relative operazioni nel file myjob.log (specificato dal comando **log**), anch'esso sul calcolatore locale.

Altri comandi utili per l'esecuzione di job nell'universo Globus sono:

```
Transfer_Executable = <true|false>
```

per indicare se l'eseguibile andrà o meno trasferito sul sistema remoto;

```
environment = <par1=val1>; .. ; <parN=valN>
```

lista delle variabili d'ambiente da impostare sul sistema remoto prima dell'esecuzione del job;

```
globusrsl = (name1=value1) .. (nameN=valueN)
```

per impostare ulteriori attributi per il job secondo il Resource Specification Language (vedi paragrafo A.1.4).

Appendice B

Risorse hardware e configurazioni software di base del sistema Grid di supporto

In questa appendice viene descritto il sistema Grid di supporto utilizzato per i test di esecuzione dell'implementazione multi-sito del Gradiente Coniugato a Blocchi e i test per la sottomissione attraverso l'MPI jobs management system.

Il sistema è composto da 3 cluster di classe Beowulf [54] e da una workstation (figura B.1).

I cluster di classe Beowulf sono denominati:

- **Vega** [vega.na.icar.cnr.it]
- **Beocomp** [beocomp.dma.unina.it]
- **Altair** [altair.dma.unina.it]

La workstation è denominata:

- **Lamu2-9** [lamu2-9.na.icar.cnr.it]

Di seguito sono descritte le caratteristiche hardware e software di base delle varie risorse della Griglia di supporto.

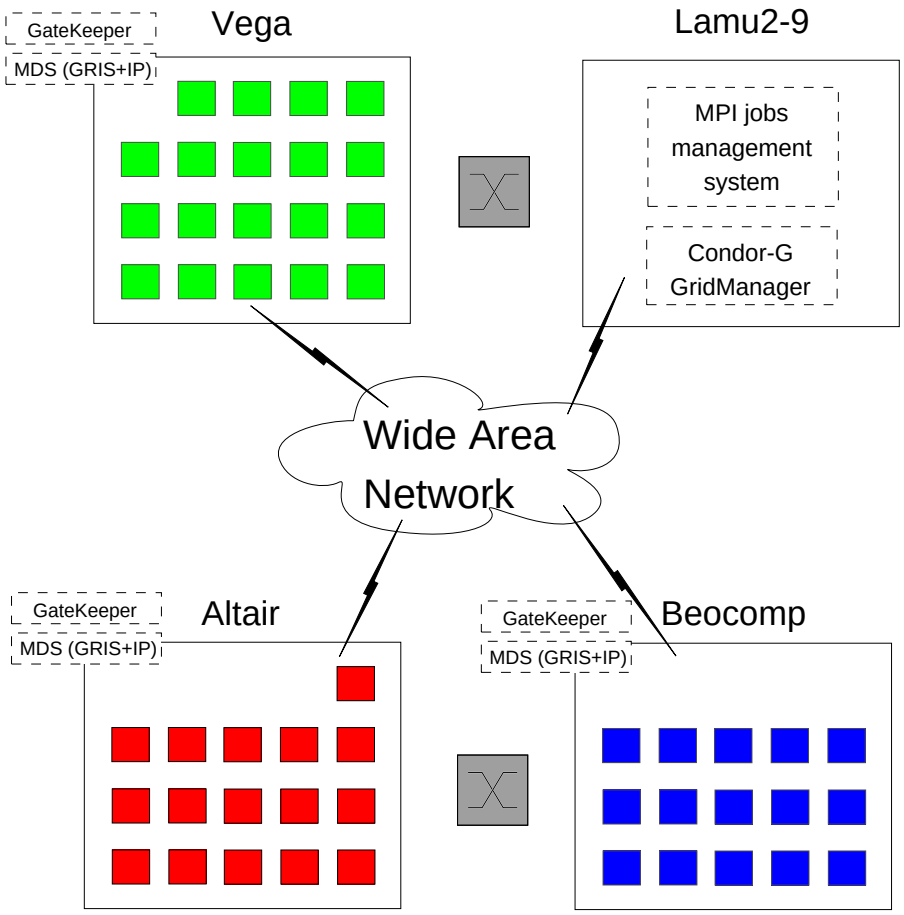

Figura B.1: Sistema Grid di supporto.

Vega Vega è un cluster Beowulf di 19 nodi. La configurazione hardware dei nodi è la seguente:

- **CPU** Intel® Pentium® 4 (Socket 423)
- **Velocità CPU** 1500MHz
- **Frontside Bus** 400 MHz
- **Chipset** Intel 850

- **Memoria** 512MB non-ECC PC800 RAMBUS
- **Hard disc** Quantum 40GB FireBall Plus AS Ultra ATA 100, 7200RPM, cache 2MB
- **Scheda di rete** 3Com 3C905

La rete di interconnessione è Fast Ethernet realizzata mediante uno switch 3com® SuperStack® II Switch 3900.

Il software di base installato su Vega è:

- **Sistema operativo** Red Hat Linux 7.2
- **Kernel Linux** 2.4.20
- **Librerie C** GNU libc 2.2.4

Il software di sviluppo installato su Vega è:

- **Compilatore C/C++** gcc 2.96
- **Ambiente di sviluppo Java** Sun Java Development Kit 1.4.2
- **Compilatore Fortran** Intel ifort 8.0
- **Librerie matematiche di base** Intel Math Kernel Library 5.2

La versione locale di MPI è mpich 1.2.5.2.

Beocomp Beocomp è un cluster Beowulf di 15 nodi. La configurazione hardware dei nodi è la seguente:

- **CPU** Intel® Pentium® II (Slot 1)
- **Velocità CPU** 450Mhz
- **Frontside Bus** 100 MHz
- **Chipset** Intel 440 BX

- **Memoria** 256MB SDRAM
- **Hard disc** Fujitsu mpc3102at 10GB Ultra ATA
- **Scheda di rete** Realtek RTL-8139

La rete di interconnessione è Fast Ethernet realizzata mediante uno switch IBM 8271-F24 a 24 porte.

Il software di base installato su Beocomp è:

- **Sistema operativo** Red Hat Linux 7.1
- **Kernel Linux** 2.2.16
- **Librerie C** GNU libc 2.2.4

Il software di sviluppo installato su Beocomp è:

- **Compilatore C/C++** gcc 2.96
- **Ambiente di sviluppo Java** Sun Java Development Kit 1.4.2
- **Compilatore Fortran** Intel ifc 8.0
- **Librerie matematiche di base** Intel Math Kernel Library 5.2

La versione locale di MPI è mpich 1.2.5.2.

Altair Altair è un cluster Beowulf di 16 nodi. La configurazione hardware dei nodi è la seguente:

- **CPU** Intel® Pentium® Pro
- **Velocità CPU** 200Mhz
- **Frontside Bus** 66 MHz
- **Chipset** Intel 440 FX

- **Memoria** 128MB DRAM
- **Hard Disc** Fujitsu mpa3026atu 2.6GB ATA
- **Scheda di rete** 3com 3c905
- **Scheda di rete** 3com 3c905B

La rete di interconnessione è una doppia rete Fast Ethernet realizzata mediante due switch Bay Networks BayStack 350T.

Il software di base installato su Altair è:

- **Sistema operativo** Red Hat Linux 7.2
- **Kernel Linux** 2.4.20
- **Librerie C** GNU libc 2.2.4

Il software di sviluppo installato su Altair è:

- **Compilatore C/C++** gcc 2.96
- **Ambiente di sviluppo Java** Sun Java Development Kit 1.4.2
- **Librerie matematiche di base** Intel Math Kernel Library 5.2

Lamu2-9 Lamu2-9 è una workstation la cui configurazione hardware è identica a a quella di un nodo del cluster Beocomp.

Il software di base installato su Lamu2-9 è:

- **Sistema operativo** Red Hat Linux 7.2
- **Kernel Linux** 2.4.20
- **Librerie C** GNU libc 2.2.4

Il software di sviluppo installato è:

- **Compilatore C/C++** gcc 2.96

B.1 Middleware per l'infrastruttura di Grid

Per la realizzazione dell'infrastruttura Grid è stato utilizzato come middleware il Globus Toolkit nella versione 2.4.3 e Condor/Condor-G nella versione 6.6.9.

Il software distribuito con il Globus Toolkit si suddivide in tre componenti:

- **Execution Management** che comprende gli strumenti per la gestione dell'esecuzione delle applicazioni sulle risorse Grid.
- **Information Services** che comprende gli strumenti per la gestione delle informazioni sulle risorse Grid.
- **Data Management** che comprende gli strumenti per la gestione e l'accesso ai dati sulla Griglia.

Il software relativo a ogni componente è a sua volta suddiviso in tre categorie:

- **Client** che comprende gli strumenti client relativi alle singole componenti.
- **Server** che comprende le applicazioni server relative alle singole componenti.
- **SDK** che comprende le librerie e gli header file necessari alla realizzazione di applicazioni basate sulle singole componenti.

Ogni calcolatore è dotato delle categorie Client, Server, SDK delle tre componenti del Globus Toolkit.

Condor/Condor-G è installato sulla workstation Lamu2-9. In particolare su tale workstation sono in esecuzione i demoni: Master, Schedd, Startd, Negotiator, Collector.

Bibliografia

[1] Argonne National Laboratory. *http://www.anl.gov.*

[2] EPCC, University of Edinburgh. *http://epcc.ed.ac.uk/.*

[3] Mississippi State University. *http://www.msstate.edu.*

[4] National Center for Supercomputing Applications (NCSA). *http://www.ncsa.uiuc.edu/.*

[5] Northern Illinois University, High Performance Computing Laboratory. *http://www.hpclab.niu.edu/.*

[6] Royal Institute of Technology, Sweden. *http://www.kth.se/.*

[7] Swedish Center for Parallel Computers. *http://www.pdc.kth.se/.*

[8] Univa Corporation. *http://www.univa.com/.*

[9] University of Southern California Information Sciences Institute. *http://www.isi.edu/.*

[10] UW-Madison Computer Sciences Department Home Page. *http://www.cs.wisc.edu/.*

[11] William Gropp, Ewing Lusk, Nathan Doss, Anthony Skjellum - *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Mathematics and Computer Science Division, Argonne National Laboratory, implementation note.

[12] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc home page. *http://www.mcs.anl.gov/petsc*.

[13] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. Internet x.509 public key infrastructure certificate policy and certification practices framework. Internet-draft, IETF, November 2003.

[14] Condor Project Homepage. *http://www.cs.wisc.edu/condor/*.

[15] J. Cullum and W. E. Donath. A block Lanczos algorithm for computing the Q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse real symmetric matrices. In *Proceedings of the 1974 IEEE Conference on Decision and Control, Phoenix, AZ*, pages 505–509, 1974.

[16] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *To appear in Proc. 10 th IEEE Symp. On High Performance Distributed Computing*, 2001.

[17] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. *Lecture Notes in Computer Science*, 1459:62–??, 1998.

[18] Karl Czajkowski, Ian T. Foster, and Carl Kesselman. Resource Co-Allocation in Computational Grids. In *HPDC*, 1999.

[19] DataTAG. *http://datatag.web.cern.ch/datatag/*.

[20] T. Dierks and C. Allen. The TLS protocol version 1, January 1999.

[21] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems*, 12:53–65, 1996.

[22] I. Foster. What is the grid? a three point checklist. *Grid Today*, VOL. 1(NO. 6), July 22, 2002.

[23] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[24] Ian Foster and Carl Kesselman. Computational grids. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 15–51. Morgan Kaufmann, 1998.

[25] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal Supercomputer Applications*, 15(3), 2001.

[26] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[27] L. Giraud and J. Langou. When modified Gram-Schmidt generates a well-conditioned set of vectors. Technical Report TR/PA/01/17, CERFACS, Toulouse, France, 2001. Preliminary version of the paper published in the IMA Journal of Numerical Analysis, vol. 22, nber 4, pp 521-528,2002.

[28] Globus Toolkit Homepage. *http://www.globus.org/toolkit/*.

[29] The GLUE Schema Activity. *http://www.cnaf.infn.it/ sergio/datatag/glue/*.

[30] G. H. Golub and C. F. Van Loan. *Matrix Computation*. John Hopkins University Press, 1989. Second Edition.

[31] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. In J. Rice, editor, *Mathematical Software III*, pages 364–377. New York, NY, USA, 1977.

[32] F. Gregoretti, G. Laccetti, A. Murli, and G. Oliva. MGF: MPI Globus Forwarder. Un'implementazione di MPI per Grid Computing. Technical Report TR-ICAR-NA-04-20, ICAR-CNR Sezione di Napoli, 2004.

[33] F. Gregoretti, G. Laccetti, A. Murli, and G. Oliva. MGF: MPI Globus Forwarder. Un'implementazione di MPI per Grid Computing. Working Note n.50, WP9 "Grid-enabled Scientific Libraries", Progetto FIRB (*Grid.it*), 2004.

[34] F. Gregoretti, G. Laccetti, A. Murli, G. Oliva, and U. Scafuri. MGF: a grid-enabled MPI library with a caching mechanism to improve collective operations. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2005), Lecture Notes in Computer Science, Springer*, 2005.

[35] Tim Howes and Mark Smith. *LDAP: programming directory-enabled applications with lightweight directory access protocol.* Macmillan Publishing Co., Inc., 1997.

[36] IETF Home Page. *http://www.ietf.org/.*

[37] iVDGL. *http://www.ivdgl.org/.*

[38] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[39] B. Kónya. *The NorduGrid Information System.* Tech. manual, NorduGrid, http://www.nordugrid.org/documents/ng-infosys.pdf.

[40] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[41] Globus Toolkit 2.2 MDS Technology Brief, January 30, 2003.

[42] GT Information Services: Monitoring & Discovery System (MDS). *http://www.globus.org/toolkit/mds*.

[43] Almerico Murli. *Lezioni di calcolo parallelo*. Liguori, 2006.

[44] Dianne P. O'Leary. Parallel implementation of the block conjugate gradient algorithm. *Parallel Computing*, 5(1-2):127–139, 1987.

[45] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. 12:617–629, 1975.

[46] The Globus Project. Writing a scheduler interface. *http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/developer/scheduler.html*.

[47] Lili Qiu, Yin Zhang, and Srinivasan Keshav. On individual and aggregate tcp performance. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, page 203. IEEE Computer Society, 1999.

[48] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[49] Rajesh Raman, Miron Livny, and Marvin Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 80, Washington, DC, USA, 2003. IEEE Computer Society.

[50] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[51] CORPORATE Computer Science and Telecommunications Board. *Realizing the information future: the Internet and beyond.* National Academy Press, 1994.

[52] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator, 1996.

[53] Marvin Solomon. *The ClassAd Language Reference Manual - Version 2.4.* Computer Sciences Department, University of Wisconsin–Madison, May, 2004.

[54] Thomas L. Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *ICPP (1)*, pages 11–14, 1995.

[55] The Globus Security Team. Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective, Version 2 updated December 8, 2004.

[56] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide: Version 2.1.* Massively Parallel Computing Research Laboratory, Sandia National Laboratories, December, 2004.

[57] UML. *http://www.uml.org/.*

[58] R. Underwood. An iterative block Lanczos method for the solution of large sparse symmetric eigenproblems. Technical Report STAN-CS-75-496, Computer Science, Stanford University, Stanford, CA, USA, 1975.

[59] Gregor von Laszewski, S. Fitzgerald, I. Foster, C. Kesselman, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance

Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, Portland, OR, 5-8 August 1997.