Università degli Studi di Perugia
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Tesi di Laurea

Implementazione di un ambiente Grid per il calcolo ad alte prestazioni

Laureando:
Carlo Manuali

Relatore:
Prof. O. Gervasi

Corelatori:
Prof. A. Laganà

Prof. S. Tasso

Anno Accademico 2001/2002

Ringraziamenti

Desidero ringraziare il Prof. *Antonio Laganà* per avermi dato l'opportunità di cimentarmi in un lavoro per me così interessante e innovativo.

Desidero ringraziare, inoltre, il Prof. *Osvaldo Gervasi* per la continua ed assidua collaborazione durante tutto lo svolgimento del progetto, il Prof. *Sergio Tasso* per l'immancabile supporto nei momenti delicati che sempre scandiscono lo sviluppo di una tesi, il Prof. *Gianfranco Galmacci* (Direttore del *Centro d'Ateneo per i Servizi Informatici* dell' Università degli Studi di Perugia, ente per il quale lavoro) per avermi concesso la disponibilità delle strutture necessarie, il Dott. *Loriano Storchi* ed il *Gruppo di Chimica Inorganica Teorica*, del Dipartimento di Chimica di Perugia, per i contributi forniti.

Ringrazio inoltre, come sempre, tutte le persone che mi sono state vicino, ed in particolare modo *Chiara*.

Infine ringrazio *Francesco Sportolari*, amico e collega che mai ha fatto mancare la propria presenza.

a Papà.

Indice

Introduzione **1**

1 HPC - High Performance Computing **4**

1.1 La teoria dei Cluster 5

1.1.1 Che cos'è un cluster 6

1.1.2 Tipologie di cluster 9

1.2 Grid 16

1.2.1 Definizione 17

1.2.2 Le motivazioni 19

1.2.3 Le applicazioni 21

1.2.4 La comunità 24

1.2.5 I Requisiti tecnici 26

1.2.6 L'architettura 27

2 Condor e Globus Toolkit 2 **30**

2.1 Condor 31
2.1.1 Che cos'è Condor 31
2.1.2 Perchè utilizzare Condor 34
2.1.3 L'ambiente operativo 36
2.1.4 Applicazioni Parallele in Condor 40
2.1.5 Condor-G 45
2.2 Globus 49
2.2.1 L'architettura Globus 50
2.2.2 GRAM - Globus Resource Allocation Manager 52
2.2.3 MDS - Monitoring and Discovery Service 62
2.2.4 Data Management 65
2.2.5 GSI - Grid Security Infrastructure 68

3 Programmazione Parallela in ambiente Grid: MPICH-G2 77

3.1 Le metodologie ed i paradigmi odierni 77
3.1.1 Gli indici di valutazione 80
3.2 MPICH-G2 82
3.2.1 Le caratteristiche principali di MPICH-G2 83
3.2.2 Come lavora MPICH-G2 88
3.2.3 Come utilizzare MPICH-G2 90

4 Sperimentazione dell'ambiente Grid 96

4.1 Installazione dei Cluster 96
4.1.1 Beowulf 97
4.1.2 Mosix 104
4.2 Installazione di Globus Toolkit 113
4.2.1 GRAM e MPICH-G2 su un'architettura *Beowulf* . . . 118
4.2.2 Le *Certification Authority* e gli utenti Globus 128
4.2.3 MDS su un'architettura *Beowulf* 133
4.3 MPICH-G2: Esempi 140
4.4 Testbed 150
4.4.1 mpptest 151
4.4.2 goptest 166
4.4.3 Valutazioni finali 170

Conclusioni **172**

Bibliografia **174**

Appendice A: Le caratteristiche tecniche dei cluster **179**

Appendice B: Il codice del programma glob_env **181**

B-1 File *glob_env.c* 181

Appendice C: L'output del comando grid-info-search **182**

Appendice D: MPICH-G2: Il codice dei programmi d'esempio 190

D-1 File *ring.c* 190

D-2 File *report_colors.c* 193

Appendice E: GRAM error codes list 195

Elenco delle figure

2.1 ***Daemon*** **principali in Condor.** 33

2.2 ***Submit description file*** **per l'esecuzione di un job MPI in ambiente Condor.** 42

2.3 ***Submit description file*** **per l'esecuzione di un job PVM in ambiente Condor.** 44

2.4 **Esecuzione remota di Condor-G sopra risorse Globus.** 46

2.5 ***Submit description file*** **per l'esecuzione di un job nell'ambiente Grid implementato da Globus mediante Condor-G.** 48

2.6 **Rappresentazione di** ***Globus Toolkit*** **in forma di 3 pilastri.** 51

2.7 **Esempio di file RSL per la sottomissione di un job in Grid.** 55

2.8 **Esempio di script RSL per l'esecuzione di un job nella griglia.** 57

2.9 **Schema di utilizzo del protocollo GASS e del sistema di *caching*.** 61

2.10 **Lo schema di funzionamento di *Monitoring and Discovery Service* (MDS).** 65

2.11 **La *Globus Security Infrastructure* (GSI).** 74

2.12 **Esempio di richiesta di un certificato alla *Certification Authority* (CA).** 75

2.13 **Inizializzazione dell'ambiente di lavoro Globus (generazione del *proxy*).** 75

3.1 **Script RSL generata dal comando mpirun di MPICH-G2 per la griglia.** 94

4.1 **Monitoraggio dei nodi del cluster GIZA tramite *bWatch*.** 104

4.2 **Topologia del cluster GRID.** 107

4.3 **mosix.map, file di configurazione principale di MOSIX (cluster GRID).** 109

4.4 **globus-gatekeeper, file di configurazione del *Globus Gatekeeper* per xinetd.** 115

4.5 globus-ftp, file di configurazione di ***GridFTP*** per xinetd. 116

4.6 L'architettura realizzata. 117

4.7 Creazione del ***Gatekeeper*** e del jobmanager (default) di Globus. 120

4.8 Codice sorgente di Globus che determina l'attuale mancato supporto di Condor come ***scheduler*** MPI. 122

4.9 Errore riportato dalla compilazione standard del pacchetto ***sdk*** di GRAM, con il ***flavor mpi***. 123

4.10 Ricompilazione manuale del componente ***globus_core*** con ***flavor mpi*** e LAM/MPI locale. 124

4.11 Tipi di dato Fortran necessari per l'esecuzione di programmi MPI (Fortran) a fronte di trasferimenti tra macchine eterogenee. 125

4.12 mpigrun, script per lo ***'spawn'*** dei job MPICH-G2 sulle risorse locali (cluster) attraverso LAM/MPI. . . . 126

4.13 hosts, file di configurazione dell'ambiente LAM/MPI locale (cluster GRID). 126

4.14 ***Makefile*** d'esempio per la compilazione di programmi MPI con MPICH-G2. 128

4.15 Interrogazione via Web delle risorse Grid. 138

4.16 ***Makefile*** **per la compilazione dell'applicazione d'esempio** ***ring*****.** 141

4.17 **ring.rsl, script RSL per l'esecuzione dell'applicazione d'esempio** ***ring*****.** 142

4.18 ***Output*** **dell'applicazione d'esempio** ***ring*** **sulla griglia computazionale (opzione** ***trip*** **= 2).** 143

4.19 ***Output*** **dell'applicazione d'esempio** ***ring*** **su 4 processori indipendentemente dalla loro locazione fisica (opzione** ***trip*** **= 2).** 145

4.20 **Rappresentazione dell'applicazione** ***ring*** **attraverso una macchina a stati finiti.** 146

4.21 ***Makefile*** **per la compilazione dell'applicazione d'esempio** ***report_colors*****.** 147

4.22 **report_colors.rsl, script RSL per l'esecuzione dell'applicazione d'esempio** ***report_colors*****.** 148

4.23 ***Output*** **dell'applicazione d'esempio** ***report_colors*** **su una griglia formata da 16 processori (cluster GRID e GIZA).** 148

4.24 **Prestazioni del cluster GRID con messaggi corti e protocollo sincrono.** 153

4.25 **Prestazioni del cluster GRID con messaggi lunghi e protocollo sincrono.** 154

4.26 **Prestazioni del cluster GRID con messaggi lunghi e protocollo asincrono.** 154

4.27 **Prestazioni del cluster GIZA con messaggi corti e protocollo sincrono.** 156

4.28 **Prestazioni del cluster GIZA con messaggi lunghi e protocollo sincrono.** 157

4.29 **Prestazioni del cluster GIZA con messaggi lunghi e protocollo asincrono.** 157

4.30 **Prestazioni del cluster HPC con messaggi corti e protocollo sincrono.** 159

4.31 **Prestazioni del cluster HPC con messaggi lunghi e protocollo sincrono.** 160

4.32 **Prestazioni del cluster HPC con messaggi lunghi e protocollo asincrono.** 160

4.33 **Comunicazioni *point–to–point* tra HPC e GRID con messaggi corti e protocollo sincrono.** 161

4.34 **Comunicazioni *point–to–point* tra HPC e GIZA con messaggi corti e protocollo sincrono.** 162

4.35 **Comunicazioni *point–to–point* tra GRID e GIZA con messaggi corti e protocollo sincrono.** 162
4.36 ***Performance* dell'intera griglia computazionale con messaggi *short* e protocollo sincrono.** 164
4.37 ***Performance* dell'intera griglia computazionale con messaggi *long* e protocollo sincrono.** 164
4.38 **Broadcast MPI sull'intera griglia a partire dal cluster GRID.** 166
4.39 **Broadcast MPI sull'intera griglia a partire dal cluster HPC.** 167
4.40 **Broadcast MPI fra 24 processori a partire dal cluster GRID.** 168
4.41 **Broadcast MPI fra 24 processori a partire dal cluster HPC.** 169

Introduzione

Lo scopo del presente lavoro di Tesi è stato quello di studiare ed implementare, con tecnologia **Grid**, un ambiente di calcolo ad alte prestazioni basato su tre **Cluster** di computer interconnessi mediante una rete WAN (*Wide Area Network*).

Si può intendere un *Cluster* come un insieme, omogeneo od eterogeneo, di computer che condividono informazioni e risorse comuni, di solito basandosi su comunicazioni ad alta velocità in un ambito spaziale estremamente ristretto.

Grid, letteralmente "griglia", è invece un tipo di sistema che permette la condivisione, la selezione e l'aggregazione di risorse distribuite attraverso differenti domini amministrativi, secondo la loro disponibilità e prestazioni, in grado di soddisfare le diverse richieste qualitative di servizio da parte degli utenti.

Nell'ambiente Grid, ogni nodo ha un proprio gestore delle risorse ed una

particolare politica di allocazione; in un cluster tali entità sono di solito assegnate ad un unico nodo, spesso identificato con il termine di *control workstation*.

In parole povere, due o più cluster amministrati all'interno di una singola Università possono identificare un'infrastruttura Grid (a volte denominata semplicemente "Grid"), come anche singole macchine o gruppi di computer sparsi su un continente.

Ha senso perciò intendere singoli PC o cluster di computer come unità primitive di Grid, come è nel nostro caso, in cui la griglia è costituita da tre cluster differenti installati all'interno dell'Ateneo di Perugia.

È facile intuire le potenzialità e le motivazioni che hanno favorito l'emergere di approcci Grid su larga scala.

Infatti, la possibilità di utilizzare risorse gestite da diverse organizzazioni indipendentemente dalla loro locazione fisica apre nuove prospettive per l'implementazione di ambienti atti ad ottenere alte prestazioni e calcoli massivi.

In particolare, nel primo capitolo di questo lavoro verrà analizzato l'ambiente Grid, la teoria dei cluster e gli strumenti applicativi a disposizione.

Nel secondo capitolo, sarà analizzata in dettaglio la struttura ed il funzionamento del software che permette di realizzare tali ambienti, fornendo dei primi esempi esplicativi.

Nel terzo capitolo, si illustrerà come adattare tecniche di programmazione già conosciute ed affermate, come ad esempio MPI, all'ambiente Grid; si parlerà quindi di MPICH-G2 e si forniranno diversi esempi di funzionamento.

Infine, nel quarto ed ultimo capitolo, verrà presentata la realizzazione dell'intero sistema, alcuni esempi rappresentativi, ma soprattutto verranno mostrati i risultati ottenuti nell'ambiente Grid realizzato per la valutazione della soluzione adottata circa le sue prestazioni.

Capitolo 1

HPC - High Performance Computing

La domanda sempre crescente di risorse di calcolo ha portato, in generale, alla crescita esponenziale della capacità elaborativa dei computer.

Nonostante l'impressionante incremento della capacità computazionale, le richieste che determinate applicazioni, specie in campi come la chimica [1], [2] o la fisica, effettuano in termini di potenza di calcolo, di memoria e di spazio disco, sono ormai divenute troppo elevate per ottenere risposte soddisfacenti da un singolo computer.

Per ovviare a questo problema, negli ultimi anni si è assistito al rapido diffondersi di strumenti di calcolo basati sull'utilizzo di diversi computer connessi in rete [3], [4].

Intuitivamente, si potrebbe pensare che l'enorme disponibilità di calcolo generata da cluster o macchine parallele specializzate con centinaia di nodi di elaborazione sia sufficiente, invece non è così.

Di fronte al crescente numero di applicazioni che necessitano grandi quantità di risorse computazionali, diventa sempre più impellente recuperare cicli macchina potenzialmente disponibili per l'elaborazione.

Questa tendenza ha portato alla diffusione di sistemi che consentono di allocare i programmi di calcolo su grandi insiemi di computer, soprattutto quando questi risultano essere scarsamente utilizzati.

La maturazione di questo processo ha portato allo sviluppo del **Grid Computing**, che consente in modo diverso l'esecuzione di applicazioni di vario tipo su una griglia di computer eterogenei, la quale può comprendere elaboratori paralleli altamente performanti, *workstation* ed anche singoli PC.

1.1 La teoria dei Cluster

A partire dai primi anni '90 le tecnologie di rete si sono ampiamente diffuse, soprattutto in ambito accademico.

Questo ha favorito all'interno di istituzioni accademiche, ditte ed istituti di ricerca, una graduale sostituzione degli ambienti computazionali basati sui grandi *mainframe*, che avevano caratterizzato gli anni '70, con stazioni di

lavoro interconnesse ad alta velocità.

Con l'avvento di questa distribuzione delle risorse computazionali è stato coniato il termine **Cluster** [5].

Il cluster risponde all'esigenza di utenti ed amministratori di sistema di assemblare insieme più macchine per garantire prestazioni, capacità, disponibilità, scalabilità ad un buon rapporto prezzo/prestazioni ed allo stesso tempo ridurre il più possibile il costo di gestione di tale sistema distribuito.

Di cluster si parla sin dalla 'preistoria' dell'Informatica (sistemi VAX della Digital -1983-, sistemi IBM JES3 per mainframe 360/370) ma spesso se ne è parlato male confondendo le problematiche hardware con quelle software.

Infatti per cluster si può intendere un modello architetturale all'interno del quale si possono ritrovare tutti i paradigmi riconosciuti, ovvero una sorta di contenitore generale da non mettere in contrapposizione con i vari tipi e modelli in esso contenuti.

1.1.1 Che cos'è un cluster

Con il termine "Cluster" si intende quindi una particolare configurazione hardware e software avente per obiettivo quello di fornire all'utente un insieme di risorse computazionali estremamente potenti ed affidabili.

Con l'avvento dei servizi di rete Internet e del web in particolare, i cluster

sono cresciuti sempre più in popolarità, proponendosi come sistemi altamente scalabili ed efficienti.

Quando siti web o server che offrono servizi vengono acceduti da molti utenti, si può prevedere l'utilizzo di un cluster al fine di distribuire il carico complessivo sui diversi computer (i "nodi" del cluster) o per garantire la continuità del servizio a fronte dell'indisponibilità di uno dei nodi.

In generale questo tipo di organizzazione è assolutamente trasparente all'utente finale; ad esempio vengono adottate soluzioni per cui all'indirizzo web del sito in questione corrisponderanno fisicamente i diversi nodi del cluster.

Ovviamente, affinchè ciò funzioni come descritto macroscopicamente in precedenza, occorre adottare una serie di accorgimenti software ed hardware particolari.

Tra questi, va citato il servizio di *Network Information System* (NIS) noto anche con il nome *Yellow Pages* (YP), che consente di distribuire le principali mappe di sistema ad un insieme di macchine, in modo che l'utente sia identificato univocamente sull'intero *pool* dei nodi.

Attraverso il servizio di *Network File System* (NFS), è invece possibile consentire all'utente (ed al sistema) di condividere lo stesso spazio disco sopra i diversi computer, mediante un meccanismo client-server che consente di montare dischi del server sui client sia in modo statico che dinamico, garantendo un'assoluta trasparenza delle operazioni, coerenza e sincronizzazione

dei dati.

In particolari ambienti in cui l'esigenza di risorse computazionali è molto sentita, rivestono un ruolo molto importante i servizi di *Batch Queueing*, i quali consistono, in sostanza, di un gestore della distribuzione del carico di lavoro su un *pool* macchine, anche eterogenee.

Il sistema rende quindi disponibili all'utente delle code, entità in grado di accettare applicazioni in esecuzione, in modo da consentire il bilanciamento del carico e l'esecuzione di job in base alle specifiche dell'utente ed alla disponibilità dei diversi sistemi.

Il sistema decide inoltre quando si verificano le condizioni per eseguire il programma, ed infine, al completamento dello stesso, si occupa della restituzione dei risultati sul richiesto flusso di *output*.

Esistono dei prodotti evoluti che permettono il bilanciamento del carico all'interno del *pool* di macchine anche di applicazioni interattive, permettendo quindi il *checkpoint* dello stato di avanzamento della computazione e la migrazione da un nodo ad un altro se per qualche ragione il nodo su cui viene lanciato il job dovesse essere spento.

La nuova frontiera va ben oltre questo approccio, fornendo dei meccanismi di migrazione automatica di processi e/o dati per garantire una maggiore omogeneità di distribuzione dei carichi di lavoro.

Si può pertanto parlare propriamente di cluster quando un insieme di

computer completi ed interconnessi ha le seguenti proprietà:

- I vari computer appaiono all'utente come una singola risorsa computazionale;
- Le varie componenti sono risorse dedicate al funzionamento dell'insieme.

1.1.2 Tipologie di cluster

Diverse motivazioni hanno contribuito alla crescente affermazione dei sistemi distribuiti:

- un ambiente distribuito è scalabile, e può essere facilmente incrementato per venire incontro a nuove esigenze computazionali;
- la richiesta di risorse computazionali può essere distribuita fra un insieme di macchine invece di essere confinata ad un singolo sistema, eliminando i cosiddetti "colli di bottiglia" e fornendo, quindi, migliori prestazioni;
- la disponibilità delle risorse e dei servizi viene fortemente incrementata.

Di contro, è possibile individuare alcune problematiche che scaturiscono proprio dalla scelta di adottare un ambiente distribuito:

- l'amministrazione di decine o centinaia di sistemi è molto più onerosa di quella del singolo sistema;
- la gestione della sicurezza cresce esponenzialmente all'aumentare degli host presenti nella rete;
- la gestione di risorse condivise via rete, è più complessa rispetto a quella locale.

Al fine di limitare l'effetto delle problematiche legate all'adozione di sistemi distribuiti e favorire la diffusione di sistemi di calcolo ad alte prestazioni implementati attraverso limitate risorse economiche è stato introdotto, dapprima in ambito accademico e poi anche in ambito aziendale, il concetto di *cluster di personal computer*.

Questo ha fatto anche si che il sistema operativo **Linux** giocasse un ruolo chiave per la definizione e l'implementazione di modelli di clustering, basati su tale concetto, atti ad affrontare e risolvere le diverse esigenze specifiche ed i vari macro–problemi.

Il modello generale

Un cluster può, in generale, essere visto come un insieme di nodi ognuno dei quali è un fornitore di servizi a tutto l'insieme.

All'interno del cluster avremo quindi diverse tipologie di nodi come *File Server*, *Batch Server*, *Tape Server* o nodi che offrono servizi specializzati come NIS, DNS, Web, Mail, FTP e così via.

Per passare da un insieme di sistemi interconnessi ad un cluster sono necessari almeno tre elementi:

- un meccanismo di condivisione delle informazioni amministrative;
- un'architettura con forte centralizzazione delle immagini di sistema (auspicabile ma non essenziale);
- un meccanismo di gestione della schedulazione di richieste delle risorse.

Quest'ultimo punto (insieme alla distribuzione sui vari nodi di tali richieste) è il primo valore aggiunto che deve essere previsto dopo lo *startup* iniziale.

Ovviamente la topologia del cluster e soprattutto la sua architettura vanno selezionate in base alle particolari esigenze degli utilizzatori ed al particolare contesto in cui si viene ad operare.

Mentre un responsabile IT potrebbe essere interessato al mantenimento della massima affidabilità dei propri server o alla diminuzione del tempo generale di esecuzione delle applicazioni e dei servizi di punta dell'organizzazione, un fisico delle alte energie potrebbe essere interessato alla simulazione su larga scala o alla disponibilità di enormi quantità di dati sperimentali fi-

sicamente distribuiti su WAN.

Si possono quindi individuare almeno quattro categorie distinte [6]:

- cluster per l'affidabilità dei servizi;
- cluster per l'alta disponibilità dei servizi (*High Availability*, HA);
- cluster per il bilanciamento del carico (*Load Balancing*, LB);
- cluster per il calcolo parallelo.

Cluster per l'affidabilità dei servizi

Tale soluzione prevede la predisposizione di un insieme di sistemi che devono mantenere copie separate dei servizi e che rispondano comunque ad un unico nome di host virtuale con il quale i client interrogano il singolo servizio.

Essenzialmente, avendo a disposizione un cluster con forte centralizzazione delle immagini di sistema ed applicative, diverse tipologie di servizi di rete si prestano per far sì che ogni nodo del cluster possa, se necessario, diventare un nodo-servizio.

Questo permette di avere una scalabilità dei servizi a costi irrisori rispetto ad un eventuale cambio architetturale (ad esempio da *low* a *high level server*).

Il punto cruciale di questo modello è il servizio di virtualizzazione dei nomi a livello DNS (fino ad arrivare a criteri avanzati di *load-balancing*) e la

gestione del carico di rete (attuabile con l'introduzione di *router* in grado di effettuarne il bilanciamento).

Cluster per l'alta disponibilità dei servizi

Se il modello di affidabilità risponde alla domanda di scalabilità, quello di alta disponibilità risponde alla domanda di continuità di servizio.

Un cluster per l'alta disponibilità (HA cluster) è costituito essenzialmente da due (o più) server gemelli dove meccanismi hardware e software permettono di far sì che se il nodo principale del cluster HA si blocca, il nodo secondario, fino a quel momento in attesa, si riconfiguri per apparire come il nodo principale del cluster, con un'immagine congrua al nodo originale immediatamente prima della sua indisponibilità (ovvero senza perdite di dati sensibili).

All'utente finale, attraverso complessi algoritmi per la determinazione delle variazioni di stato dei sistemi, l'intero processo di riconfigurazione apparirà come una breve indisponibilità temporanea del servizio.

Tale modello è particolarmente indicato per tipologie di servizi come *File Server* o *Database Server*.

Cluster per il bilanciamento del carico

Un *load-balancing* cluster può essere un incomparabile strumento di produttività.

L'idea (che estende il modello di affidabilità) è quella di introdurre un sotto-sistema di schedulazione delle richieste applicative, ad esempio un sistema di code che sia in grado di reindirizzare la richiesta sul nodo più scarico, in grado di far fronte alla richiesta utente.

I sotto-sistemi di questo tipo solitamente sono dotati di vari strumenti amministrativi che permettono un controllo molto fine sull'utilizzo delle risorse e che consentono un monitoraggio avanzato dello stato del cluster.

L'asintoto attuale di questo modello è rappresentato dalla possibilità di includere a livello *kernel* i meccanismi di schedulazione, la gestione dello spazio globale delle risorse (*file system* e processi) e la gestione delle politiche di bilanciamento del carico e della migrazione dei processi.

Cluster per il calcolo parallelo

La ragione principale per cui molti sforzi si sono concentrati verso lo sviluppo di architetture cluster per il calcolo parallelo basate su PC sono i costi.

In questo modo, invece di rincorrere a colpi di svariati milioni di dollari l'ultimo supercomputer in grado di far fronte alle sempre maggiori richieste di

TeraFlops (trilioni di operazioni in virgola mobile per secondo), si è sempre più affermata l'idea di assemblare grandi quantità di processori classe PC con una struttura di comunicazione a banda alta e bassa latenza, appositamente progettata.

Per mantenere tali caratteristiche a volte viene utilizzato un protocollo di rete diverso dal **TCP/IP** (come **Myrinet**), che contiene troppo *overhead* rispetto alle limitate esigenze di indirizzamento, *routing* e controllo nell'ambito di una rete in cui i nodi siano ben noti a priori.

In alcuni casi viene utilizzato un meccanismo di *Direct Memory Access* (DMA) fra i nodi, fornendo una sorta di *distribuited shared memory* che può essere acceduta direttamente da ogni processore su ogni nodo.

Inoltre, è previsto un livello di comunicazione a scambio di messaggi (*message passing*) per la sincronizzazione dei nodi, le cui implementazioni più diffuse sono rappresentate da **MPI** (*Message Passing Interface*), **OpenMP** (*Open Message Passing*) e **PVM** (*Parallel Virtual Machine*).

MPI, come vedremo, è una API (*Application Program Interface*) per gli sviluppatori di programmi paralleli che garantisce una piena astrazione dall'hardware correntemente utilizzato senza alcuna necessità di inserire nel codice del programma alcuna direttiva di effettiva distribuzione dei segmenti di codice fra i nodi del cluster garantendo, fra l'altro, una buona portabilità del codice prodotto.

PVM permette ad un insieme di computer di essere visto come una singola macchina parallela; la *Parallel Virtual Machine* è composta da due entità principali: il processo *PVM daemon* su ogni processore e l'insieme delle *routine* che ne forniscono il controllo.

1.2 Grid

In quest'ultimo biennio il mondo scientifico e quello commerciale sono stati interessati dall'esplosione del fenomeno del *Grid Computing*, il quale viene da più parti dichiarato come la piattaforma di sviluppo e di evoluzione di Internet nei prossimi anni.

Una delle prime definizioni proposte da *Ian Foster* e *Carl Kesselman*, nel 1998, relativamente alle griglie computazionali, recita:

> *"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities".*

In seguito, tale concetto si è arricchito a fronte di nuove tematiche politico–sociali; le griglie computazionali divengono quindi la piattaforma di riferimento per la condivisione coordinata delle risorse e per la soluzione di problemi in organizzazioni virtuali, dinamiche e multi-istituzionali:

*"The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a **virtual organization**".*

1.2.1 Definizione

Oggi, si possono riassumere e coordinare le varie successive definizioni in tre punti chiave, assumendo quindi che *Grid* è un sistema che [7]:

- Coordina risorse che non sono soggette ad un controllo centralizzato. (integra e coordina risorse ed utenti che non sono definite in un unico dominio di gestione, per esempio differenti unità amministrative di una stessa compagnia o differenti compagnie, dove le varie politiche di sicurezza, appartenenza, autenticazione ed esecuzione divengono trasparenti);

- Utilizza interfacce e protocolli standard, aperti e *general-purpose*.

  (un sistema *Grid* si basa su tali protocolli ed interfacce per la gestione delle autorizzazioni, dell'autenticazione e per l'accesso alle risorse disponibili);

- Produce un'accettabile qualità di servizio.

  (un sistema *Grid* permette di gestire le proprie risorse per ottenere diverse qualità di servizio come tempi di risposta, disponibilità, sicurezza e *performance*, per soddisfare completamente la richiesta utente; in questo contesto le griglie computazionali così ottenute rappresentano un sistema combinato più efficiente rispetto alla somma delle sue singole parti).

Particolare attenzione è rivolta al fatto che i diversi protocolli di gestione debbano essere *standard*: risulta infatti quasi scontato che per ottenere la condivisione di risorse fra ambienti completamente diversi ed eterogenei sia necessaria la presenza di interfacce di comunicazione comuni.

Un esempio di tali requisiti è rappresentato da **Globus Toolkit**, che, come vedremo più avanti, offre appunto un insieme di protocolli standard ed aperti, *general-purpose*, per gestire e negoziare le condivisioni ed indirizzare le diverse dimensioni di qualità di servizio.

1.2.2 Le motivazioni

I computer sono utilizzati per modellare e simulare problemi scientifici ed ingegneristici complessi, diagnosticare analisi mediche, controllare equipaggiamenti industriali, previsioni del tempo, operazioni bancarie, biotecnologie e molto altro.

L'osservazione principale è che la media degli ambienti computazionali esistenti risulta inadeguata ad assolvere tali funzioni al fine di ottenere obiettivi così sofisticati e complessi.

Il termine *"computational grid"* si avvicina fortemente al concetto di *"electric power grid"*, modello delle reti di distribuzione dell'energia elettrica, al quale idealmente si ispira.

Una rete elettrica rende oggigiorno ampiamente disponibili e collega fonti di energia eterogenee (vi sono centrali elettriche di innumerevoli tipi diversi, che sfruttano le più svariate tecnologie), distanti, gestite da soggetti ed enti diversi: basti pensare al fatto che intere nazioni acquistano e vendono quotidianamente energia elettrica.

Allo stesso modo si può pensare a dei prototipi di Grid quando si considerano le reti di comunicazione stradali, ferroviarie, marittime e aeree o le reti telefoniche, postali e di trasporto, che sono tutte entità complesse ed interdipendenti.

Allo stesso modo le griglie computazionali dipendono da altre infrastrutture, come ad esempio cluster di varia natura, estendendone le capacità e rappresentandone, quindi, la naturale evoluzione.

Per continuare con l'analogia della rete elettrica, è possibile notare che lo stato attuale delle risorse di calcolo è tutto sommato quasi analogo a quello dell'elettricità nei primi anni del '900.

Infatti in quegli anni era possibile generare energia elettrica, venivano inoltre costruiti macchinari che dipendevano da questa per funzionare, ma la necessità, per ogni utilizzatore, di dover disporre oltre al macchinario, anche di un generatore, ne limitava fortemente la diffusione; la vera rivoluzione, in questo settore, si ebbe con lo sviluppo delle tecnologie per il trasporto e la distribuzione di energia elettrica su larga scala: questo processo evolutivo ha portato oggi ad una disponibilità capillare di energia elettrica affidabile ed a basso costo.

Allo stesso modo oggi esistono, localizzate in tutto il globo, risorse di calcolo ed applicazioni che lavorano su di esse; tuttavia esse sono separate fra loro, disomogenee e spesso confinate alla soluzione di problemi specifici: potremmo infatti parlare di risorse computazionali *"embedded"*.

Una tecnologia atta alla condivisione di tali risorse, permetterebbe quindi di renderle disponibili (magari a pagamento), a chi necessita (ad esempio intere organizzazioni), o semplicemente a coloro che occasionalmente richie-

dano certe capacità (ad esempio dei laboratori o utenti singoli), senza farsi carico della notevole spesa che comporterebbe una loro installazione in loco.

1.2.3 Le applicazioni

Gli ambienti operativi che presumibilmente utilizzeranno appieno il *Grid Computing* possono essere suddivisi in cinque classi.

Supercalcolo distribuito

Questo tipo di applicazioni possono utilizzare Grid per unificare le risorse di calcolo di alcuni, o molti, supercomputer al fine di risolvere problemi che, altrimenti, sarebbe oggi proibitivo risolvere su di un unico calcolatore, per quanto potente.

Un tipico esempio di applicazione che trarrebbe grande vantaggio dall'uso di Grid è la *Simulazione Interattiva Distribuita* (DIS), tecnica molto usata nell'ambiente militare per la simulazione di scenari di guerra realistici, che coinvolgono centinaia di migliaia di elementi dai comportamenti eterogenei e complessi; ad oggi, anche i più potenti elaboratori possono trattare al più poche decine di migliaia di questi elementi.

Un ambito in cui il supercalcolo distribuito può portare notevoli innovazioni è quello scientifico; una simulazione accurata di un fenomeno spesso richiede risoluzioni spaziali e temporali molto elevate; in queste situazioni,

utilizzare insieme diversi supercomputer o veri e propri insiemi di cluster può portare a superare le limitazioni esistenti.

Campi applicativi in cui si è già iniziato a sperimentare tali soluzioni sono la chimica computazionale [8], [9], i modelli climatici e la cosmologia.

High Throughput Computing

In questo tipo di applicazioni, Grid può essere utilizzato per organizzare il lavoro di un grande numero di programmi, che siano tra loro poco o per nulla collegati; lo scopo di questi sistemi è, in genere, quello di sfruttare il tempo macchina inutilizzato, ad esempio durante le ore notturne.

Un esempio di software adatto ad implementare architetture di questo tipo è appunto **Condor**, sviluppato dall'*Università del Wisconsin*, utilizzato per gestire *pool* di centinaia di *workstation* in università ed enti di ricerca di tutto il mondo [10].

On Demand Computing

Le applicazioni *on demand* (a richiesta) utilizzano Grid per rendere disponibili risorse di calcolo che, per il loro saltuario utilizzo, non sarebbe conveniente avere sempre a disposizione.

Queste risorse possono essere processori, software, archivi di dati, strumenti molto specializzati e così via.

Al contrario del supercalcolo distribuito, queste applicazioni sono spesso motivate dal rapporto costo-prestazioni, piuttosto che dalla necessità di *performance*.

Applicazioni di questo tipo, come programmi di analisi numerica o acquisizione in tempo reale di immagini, rappresentano notevoli sfide per Grid, in quanto sono richiesti livelli di flessibilità e complessità che obbligano a trattare con molta cura le problematiche relative all'allocazione di risorse, gestione delle code, affidabilità e sicurezza.

Data Intensive Computing

Nelle applicazioni di questo tipo, il cui scopo è la gestione di enormi quantità di dati distribuiti geograficamente (diversi *Terabyte* di dati generati quotidianamente), si riscontrano spesso problemi legati all'alto carico computazionale ed alle modalità di trasferimento di tali moli di dati.

Le sfide principali di queste applicazioni sono legate alle problematiche di *scheduling* ed alla gestione di flussi di dati numerosi e complessi.

Calcolo collaborativo

Questo tipo di applicazioni mirano soprattutto a favorire le comunicazioni e le collaborazioni tra le persone fisiche, pertanto sono spesso pensate in termini di spazi virtuali.

Molte di queste applicazioni devono rendere disponibili risorse di calcolo condivise, ad esempio archivi di dati, e pertanto condividono tutti gli aspetti delle applicazioni già analizzate; in questo contesto, ciò che assume particolare rilievo è la necessità di fornire queste funzionalità in tempo reale.

1.2.4 La comunità

Dall'analisi svolta, si è evidenziato che, almeno in un primo momento, le categorie di persone che maggiormente saranno interessate all'utilizzazione, e quindi allo sviluppo di Grid, saranno scienziati, tecnici, ingegneri e, comunque, tutte quelle categorie di persone direttamente interessate alle ricadute tecnologiche della scienza.

Tali soggetti in genere necessitano di visualizzare i risultati delle proprie applicazioni in tempo reale, e spesso necessitano della possibilità di modificare parametri elaborativi in corso di esecuzione.

Oggi giorno è spesso necessario, per certe applicazioni, aspettare giornate intere per poter ottenere il risultato di una simulazione ed utilizzare dispositivi di memorizzazione come nastri per archiviare o trasportare i risultati ottenuti; di fatto passano giorni, o settimane, prima di poter valutare il risultato finale.

Risulterebbe molto più efficiente poter controllare in tempo reale i risul-

tati della simulazione e, magari, poterne modificare i parametri, evitando così situazioni come quelle appena descritte.

All'esterno del mondo scientifico Grid si prospetta di notevole interesse anche per grosse multinazionali, che potranno beneficiare della conseguente capacità di aggregazione, piuttosto che scuole e università che potranno condividere sia archivi che risorse multimediali, o accedere in modo proficuo alle diverse banche dati.

Bisogna infine ricordare tra i beneficiari anche gli organi governativi dei vari stati, che potranno in futuro usufruire di Grid per l'integrazione di strumenti atti a gestire l'intera "cosa pubblica", come ad esempio la manutenzione di archivi catastali elettronici a larga diffusione, registri automobilistici o dati fiscali distribuiti lungo il territorio nazionale.

Ovviamente, oltre gli utenti finali, diversi personaggi si rendono necessari nella comunità per lo sviluppo di tali tecnologie; dagli sviluppatori dei sistemi di gestione delle griglie agli sviluppatori di applicazioni *"Grid–Oriented"*, dagli amministratori di sistema dei singoli centri elaborativi ai *"Grid Administrator"*.

1.2.5 I Requisiti tecnici

Affinchè Grid si affermi decisamente sul panorama internazionale, e quindi si riveli utilizzabile nelle applicazioni descritte in precedenza, è necessario che vengano soddisfatti alcuni requisiti.

Prima di tutto, va evidenziato che una griglia computazionale deve essere un'*infrastruttura*, in quanto pensata come una struttura su larga scala che metta a disposizione risorse di calcolo, di memorizzazione e di trasporto dati in modo trasparente all'utente finale oltre che assolvere le funzioni di monitoraggio e controllo.

Un'altro requisito indispensabile è la *continuità del servizio*, in quanto gli utenti devono poter contare su un sistema che garantisca loro prestazioni di alto livello ma soprattutto senza interruzioni: è difatti impensabile lavorare su una piattaforma che non garantisca prestazioni più o meno omogenee o addirittura che non risulti più disponibile con il rischio di perdere il proprio lavoro.

Un altro fattore decisivo è costituito dalla *diffusione*: una tecnologia infatti si rivela tanto più utile quanto più utilizzata, quindi disponibile.

Vi è poi la *consistenza del servizio*: affinchè tali sistemi presentino un'elevata diffusione risulta necessario, come già accennato, che essi stessi siano basati su tecnologie (protocolli e varie interfacce di comunicazione) *standard*,

sulla speranza che restino tali il più a lungo possibile.

Ovviamente, vista l'eterogeneità dei destinatari (a volte perfetti sconosciuti), la criticità delle appplicazioni, l'importanza dei dati e l'obbligatoria riservatezza, è scontato sottointendere come tali sistemi debbano essere *fault-tolerant* ma soprattutto definiti su strutture architetturali che garantiscano un alto livello di *sicurezza*, requisito obbligatorio di tutti i sistemi produttivi, tanto più se a pagamento.

Infine, il costo deve risultare ragionevolmente sbilanciato verso il basso a fronte dei ricavi ottenuti: è per questo (ma anche per altri fattori come la diffusione, il bacino di adesione e le migliori funzionalità rispetto ad altri sistemi proprietari) che spesso, come nel nostro caso, vengono utilizzati sistemi operativi *Open Source* quale **Linux**.

1.2.6 L'architettura

In relazione alla complessità, si possono individuare quattro livelli dimensionali sui quali si può operare:

- *End System*: il modello a "singolo computer" che per decenni ha costituito l'unica risorsa computazionale;
- *Cluster*: insiemi di computer, di solito omogenei, che hanno introdotto nuove tematiche di parallelismo e gestione distribuita;

- *Intranet*: modello che sopravanza il precedente allargando le prospettive verso una più ampia distribuzione geografica ed eterogeneità dei sistemi;

- *Internet*: modello che prevede la generalizzazione delle procedure e l'estensione geografica su scala mondiale, con pesanti implicazioni in termini di sicurezza ed affidabilità delle diverse infrastrutture.

A prescindere dal particolare modello, anche se, come si è ampiamente evidenziato, con Grid si intendono infrastrutture che dovranno soddisfare esigenze profondamente eterogenee, è tuttavia possibile identificare alcuni servizi di base che la maggior parte delle griglie computazionali debbono necessariamente fornire.

Il primo servizio indispensabile, in un sistema di calcolo che coinvolge delle risorse distribuite, è una procedura di autenticazione che stabilisca i diritti di un utente in merito all'esecuzione di applicazioni, dove ognuno di essi genera uno o più *thread* di controllo; i vari processi, eseguiti in uno spazio di memoria distribuito tra i vari host, possono poi comunicare tra loro attraverso scambi di messaggi.

Un processo quindi agisce per conto dell'utente (che sovrintende l'intera attività elaborativa attraverso l'uso di segnali scambiati sempre tramite messaggi) che lo ha creato con lo scopo di acquisire risorse: verranno quin-

di eseguite le istruzioni, le letture e scritture su disco e l'invio/ricezione dei messaggi.

La capacità di acquisire risorse è pertanto limitata dal meccanismo di autorizzazione sottostante che implementa la relativa *policy* per l'allocazione delle risorse stesse; tale *policy* tiene normalmente conto dell'identità dell'utente, dei suoi privilegi, delle richieste concorrenti da parte di altri utenti e così via.

In *background*, un sistema di *scheduling* provvederà a gestire le richieste multiple e concorrenti ed un sistema di *accounting* terrà traccia delle risorse utilizzate da ogni singolo utente sia per quello che riguarda la memoria principale utilizzata che per i vari dispositivi di memorizzazione secondari come la memoria virtuale, *file system* o *database*.

Capitolo 2

Condor e Globus Toolkit 2

Gli ultimi due anni hanno visto il progressivo affermarsi di due tecnologie per la condivisione di risorse nei sistemi distribuiti: il *Grid Computing* ed i *Web Services*.

Il termine *Grid Computing*, che nasce e si sviluppa nel mondo dei grandi gruppi di ricerca scientifica internazionale, indica, come si è visto, un insieme di tecniche che consentono la condivisione di risorse informative al fine di rendere disponibile la potenza elaborativa necessaria per affrontare le grandi sfide computazionali della fisica, della chimica [11], dell'astronomia e così via.

I *Web Services* nascono invece dal mondo dell'*e-business* e mirano a rendere semplice, efficiente e basato su standard l'interazione fra programmi all'interno di ambienti complessi e eterogenei quali una *supply chain* o un *market place*.

Al recente *Global Grid Forum* (GGF4), **Globus Project** e **IBM** hanno annunciato l'*Open Grid Services Architecture* (OGSA) che rappresenta il matrimonio e l'evoluzione di queste due tecnologie.

OGSA introduce il concetto fondamentale di *Grid Service*, che altro non è che un *web service* che dispone di interfacce che lo rendono manipolabile per mezzo di protocolli Grid.

In questo capitolo, analizzeremo quindi le caratteristiche fondamentali di **Condor** e **Globus**, due fra gli applicativi più utilizzati per la costituzione e gestione dei servizi Grid basati su griglie computazionali; in particolare quest'ultimo sarà approfondito più nel dettaglio e costituirà l'elemento portante di questo lavoro, in quanto tale architettura risulta in maniera determinante essere la migliore e la più utilizzata nel panorama mondiale, rispetto ad altre soluzioni, per una serie di aspetti tecnici quali ad esempio la gestione dei varie problematiche relative alla sicurezza.

2.1 Condor

2.1.1 Che cos'è Condor

Condor è un particolare sistema per la gestione del carico di lavoro di applicazioni che richiedono grandi risorse computazionali.

Esso nasce nell'ambito *Condor Research Project*, ideato da *Miron Livny* dell'Università del *Wisconsin - Madison* (UW-Madison) nel 1988.

Come altri sistemi *batch* completi, Condor offre meccanismi specifici per la gestione delle code, politiche di *scheduling* dei job, gestione delle priorità, monitoraggio delle risorse e gestione delle stesse; per assolvere a tutte queste funzionalità, esso si avvale di diversi processi (fig. **2.1**) che vengono illustrati in dettaglio:

- **master daemon**: demone principale che sovraintende tutti i demoni in ogni macchina del *pool* (insieme di host) Condor;
- **schedd daemon**: demone che identifica realmente una risorsa, deve essere presente in ogni macchina abilitata all'esecuzione di job;
- **startd daemon**: demone che schedula le richieste dei job pervenute tramite l'utilizzo di code di esecuzione;
- **negotiator daemon**: demone responsabile di tutte le negoziazioni fra la richiesta di risorse per i job e l'offerta del *pool*, alla ricerca dei possibili *matching*, cioè della corretta allocazione dei programmi nei nodi appropriati;
- **collector daemon**: demone responsabile della raccolta di tutte le informazioni di stato di un *pool* Condor.

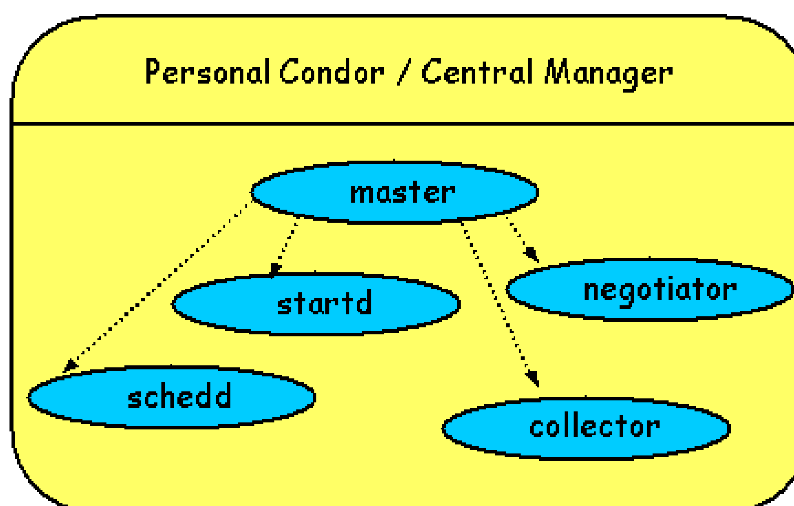

Figura 2.1: ***Daemon* principali in Condor.**

Gli utenti possono sottomettere job seriali o paralleli a Condor, il quale li inserirà in una coda, sceglierà quando e dove eseguirli in base a *policy* specifiche, terrà sotto controllo il loro evolversi ed infine notificherà l'utente dopo il loro completamento.

Condor, a differenza di altri sistemi di *scheduling*, permette sia di gestire cluster di computer dedicati (come *Beowulf*) o utilizzare risorse arbitrarie di workstation inattive (ad esempio quando non viene rilevata attività di mouse o tastiera per più di un certo intervallo di tempo).

Come risultato, Condor permette di combinare insieme tutta la potenza computazionale di una organizzazione in un'unica risorsa.

Il meccanismo denominato *ClassAd* offre uno strumento flessibile per gestire le risorse richieste (dagli utenti) e le risorse offerte (dalle macchine); esso

consiste in una serie di espressioni regolari definite secondo un paradigma semplice ed intuitivo.

Condor può inoltre essere utilizzato per costituire vere e proprie griglie computazionali oltre i limiti canonici imposti tra le organizzazioni; tale tecnologia, chiamata *flocking*, permette infatti a diversi *pool* Condor di lavorare insieme come un'unica risorsa globale.

2.1.2 Perchè utilizzare Condor

Per molti scienziati o ricercatori, la qualità della ricerca dipende fortemente dalla potenza di calcolo loro offerta.

Una chiave fondamentale per ottenere tale potenza, è l'efficiente utilizzo delle risorse disponibili; allo scopo, Condor implementa due importanti funzioni come l'utilizzo di macchine "utente" altrimenti inoperative e l'organizzazione distribuita dei job sopra diverse risorse dedicate come singoli cluster o insiemi di essi.

Oltrechè usufruire dei vantaggi espressi fino ad ora, Condor permette agli utilizzatori di inviare diversi e numerosi job insieme; in questo modo, un'enorme quantità di computazione può essere eseguita con un intervento degli utenti veramente ridotto.

Condor infatti offre altre importanti funzionalità come ad esempio il non

dover modificare nulla del proprio programma appena si decida di farlo girare in un ambiente di questo tipo: automaticamente tali job possono produrre *checkpoint* ed effettuare chiamate di sistema remote.

Un *checkpoint* è un insieme completo di informazioni che comprendono lo stato del programma; esso viene principalmente utilizzato nell'esecuzione di applicazioni molto onerose, in modo da consentirne la ripresa dopo un'eventuale interruzione.

Per lunghe computazioni, la possibilità di utilizzare questi meccanismi può permettere di risparmiare giorni o intere settimane di calcoli già effettuati; se ad esempio la macchina andasse in *crash* oppure dovesse essere riavviata per *task* amministrativi, un *checkpoint* preserva lo stato del calcolo già effettuato.

Questo meccanismo è inoltre indispensabile in quei contesti dove l'autorizzazione ad utilizzare le capacità computazionali di un host è subordinata al fatto che il computer sia inattivo; in questo modo, l'applicazione viene interrotta (ed il lavoro svolto viene quindi salvato) quando l'host torna ad essere operativo.

Tale sistema consente di utilizzare tutta la potenza computazionale degli host di un'organizzazione, senza penalizzare coloro che utilizzano le stesse risorse per i loro fini primari.

Condor può inoltre effettuare *process migration*: può cioè far proseguire i

job interrotti da una macchina ad un'altra o sopra un differente *pool* Condor (della stessa piattaforma).

Quando un utente sottomette un job a Condor, esso viene eseguito su una macchina remota che il *pool* Condor mette a disposizione in quel momento: quando un job effettua una chiamata di sistema, per esempio per attività di input/output, i dati sono mantenuti sulla macchina alla quale il job era stato inviato originariamente.

In ogni caso, le chiamate di sistema vengono effettuate da Condor stesso e non dal sistema operativo della macchina remota: Condor infatti invia la chiamata di sistema dalla macchina remota alla macchina dove era stato sottomesso il job; essa viene poi eseguita e Condor ritorna indietro i risultati alla macchina remota di partenza.

Questa implementazione permette ad un utente che vuole inviare i propri job a Condor di non dover necessariamente avere un *account* definito sulla macchina remota.

2.1.3 L'ambiente operativo

Tanto più diverse macchine partecipano ad un *pool* condor, tanto più la quantità di risorse computazionali aumenta; Condor infatti lavora molto bene con *pool* formati da centinaia di macchine.

Mediante il meccanismo di *ClassAd* l'utente può richiedere tutto ciò di cui necessita al sistema Condor, compresa la specifica piattaforma di esecuzione per la particolare applicazione.

Le varie macchine disponibili hanno una serie di caratteristiche intrinseche come appunto il tipo di sistema operativo disponibile, la quantità di memoria libera o il numero di processori; Condor cerca quindi di effettuare un *matching* fra le caratteristiche, presenti al momento, delle macchine e le richieste di un job.

Oltre alle risorse convenzionali, questo meccanismo permette di specificare risorse particolari come le migliori funzionalità per le gestione di operazioni in virgola mobile di una piattaforma rispetto ad un altra oppure scegliere insiemi omogenei di macchine che condividono una stessa architettura.

L'utente che vuole specificare tali opzioni produce un file di configurazione che specifica tutte le preferenze ed i parametri per quel job; in questo modo l'utente, padrone del job, ha un controllo flessibile e completo sopra le proprie macchine e può esso stesso farle partecipare ad un *pool* Condor, attraverso un meccanismo chiamato *joining*.

Per inviare job a Condor, si possono individuare quattro *step* fondamentali:

- **Preparazione del codice**: un job su condor deve essere in grado di

girare in *backgroud* senza input/output interattivo; Condor infatti può riassegnare l'output della console (*stdout* e *stderr*) o l'input da tastiera (*stdin*) ad un file specificato, tramite comandi Condor: si dovranno quindi preparare degli appositi file contenenti l'input del programma, i quali verranno inviati all'host che eseguirà il job al momento della sua sottomissione all'ambiente Condor;

- **Scelta dell'universo**: a seconda delle situazioni, andrà scelto un ambiente operativo di Condor sotto il quale far girare i diversi job, come vedremo successivamente;
- **Creazione del file di comandi Condor**: una volta stabiliti tutti i parametri per il nostro job, andrà creato un file di testo (chiamato *submit description file*) dove si potranno specificare tutte le informazioni necessarie quali ad esempio il nome dell'eseguibile, i file associati all'input/output, il file di *log*, le caratteristiche richieste, l'universo da utilizzare piuttosto che l'E-Mail dove spedire la notifica dell'avvenuto completamento (o eventuali errori);
- **Invio del job**: il job verrà inviato attraverso il comando `condor_submit` fornendo come argomento il *submit description file* appena creato.

L'universo di Condor definisce diversi ambienti in cui un job può essere eseguito; esistono i seguenti universi:

- **Standard**: in questo ambiente, Condor fornisce *checkpointing* e *remote system calls*; per preparare un programma per essere eseguito in questo universo, si dovrà provvedere al *re-link* attraverso il comando `condor_compile` (non è quindi necessario modificare il codice sorgente). Tale universo comunque presenta alcune restrizioni come l'impossibilità di utilizzare *shared memory*, *memory mapped files* (per esempio la chiamata `mmap()`), *interprocess communication* (per esempio semafori o pipe), *multi-process job* (per esempio le chiamate di sistema `fork()`, `exec()` e `system()`) ed altro;

- **Vanilla**: tale universo è di solito utilizzato per programmi che non possono essere "re-linkati" o per *shell script*; purtroppo in questo universo non è possibile utilizzare chiamate di sistema remote o *checkpoint*. In UNIX, Condor si basa su meccanismi di condivisione di file come NFS o AFS per assicurare che l'accesso ai dati possa avvenire da qualsiasi macchina che potenzialmente eseguirà il job in maniera coerente rispetto alle informazioni amministrative.

- **PVM**: questo universo permette di far girare sopra Condor programmi scritti per l'interfaccia PVM (*Parallel Virtual Machine*);

- **MPI**: in questo caso è possibile eseguire programmi scritti secondo il paradigma di *Message Passing Interface* (MPICH);

- **Globus**: tale ambiente fornisce un'interfaccia standard per avviare job in un sistema *Globus* attraverso Condor (Condor-G);
- **Java**: un programma inviato a quest'ultimo universo può essere eseguito su Condor attraverso una *Java Virtual Machine* (JVM) standard o ottimizzata.

2.1.4 Applicazioni Parallele in Condor

Condor supporta due fra gli ambienti di programmazione parallela più diffusi: MPI (*Message Passing Interface*) [12] e PVM [13] (*Parallel Virtual Machine*).

MPI offre un ambiente per la comunicazione e la sincronizzazione di programmi paralleli; eseguire programmi basati su tale paradigma in Condor semplifica molto il lavoro ai programmatori: Condor infatti dedica le risorse per l'esecuzione di questi programmi attraverso le stesse modalità utilizzate per tutti i job non MPI.

Attualmente Condor supporta MPICH, versione *ch_p4*, l'implementazione MPI dell'*Argonne National Labs* [14]: i programmi ivi prodotti devono quindi essere compilati con il comando `mpicc` per essere sottomessi per l'esecuzione a Condor, senza ulteriori operazioni di *linking* o successive ricompilazioni.

PVM offre una serie di primitive per il *message passing* da utilizzare nei programmi scritti in C, C++ o Fortran; tali primitive, insieme con l'ambien-

te PVM, permettono la parallelizzazione a livello di programma: processi multipli possono essere eseguiti su diverse macchine e possono comunicare con le altre per scambiare dati ed informazioni.

Condor-PVM offre un *framework* per eseguire applicazioni PVM in un ambiente Condor: mentre PVM necessita di macchine dedicate per eseguire applicazioni PVM, Condor può essere utilizzato per costruire dinamicamente macchine virtuali PVM a partire da un *pool* condor di macchine.

Supporto MPI

Per eseguire programmi MPI con Condor, è necessario configurare le macchine atte allo scopo secondo uno *scheduling* dedicato (normalmente Condor utilizza uno *scheduling* opportunistico); le risorse devono quindi garantire che un programma non sia mai sospeso (ad esempio per mandare in esecuzione un altro programma) nè interrotto.

Per semplificare lo *scheduling* dedicato delle risorse, di solito una singola macchina viene eletta come *scheduler* di job MPI; tale configurazione obbliga quindi l'utente a sottomettere i propri programmi MPI attraverso tale macchina.

Tipicamente, una volta che il programma è stato scritto e compilato, e le risorse Condor correttamente configurate, si utilizzerà un *submit description file* simile a quello mostrato in figura **2.2**.

```
#########################################
## MPI example submit description file ##
#########################################
universe = MPI
executable = mpi_program
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
log = logfile
machine_count = 4
queue
```

Figura 2.2: ***Submit description file* per l'esecuzione di un job MPI in ambiente Condor.**

Dopo aver ovviamente scelto come *universe* MPI, verranno specificati i vari parametri tipici che comprendono il nome del file eseguibile, gli eventuali file di *input*, *output*, *error* e *log* ed il numero di macchine richieste per il job; infine tramite il comando `queue` il programma verrà effettivamente spedito alla coda dello *scheduler* Condor.

Condor-PVM

Attraverso *Condor-PVM* (un modulo opzionale di Condor), vengono supportate le funzioni di *resource manager* per il demone PVM: quando un programma PVM richiede dei nodi, tale richiesta viene reindirizzata a Condor; esso individua quindi una macchina del *pool* attraverso i meccanismi consueti e la aggiunge alla *virtual machine* PVM (quando una macchina dovrà lasciare suddetto *pool*, il programma PVM verrà informato tramite i meccanismi

PVM standard).

Uno dei più comuni paradigmi di programmazione parallela è conosciuto con il nome di *master-worker*.

In questo modello, un nodo attua la funzione di controllore per le applicazioni parallele ed invia parti di lavoro agli altri nodi (*worker*) che eseguono la loro parte computazionale e ritornano indietro i risultati al *master* che provvederà a riorganizzarli insieme.

Condor-PVM utilizza anch'esso tale paradigma utilizzando come *master* il nodo al quale è stato sottomesso il job e come *worker* i nodi disponibili del *pool* Condor (al contrario di MPI, in questo caso è possibile utilizzare un ambiente operativo opportunistico).

I job PVM sono ovviamente sottomessi a Condor attraverso il *PVM universe*; in figura **2.3** si riporta un tipico *submit description file* per un job PVM che ha come *master* il programma `master.exe`.

Il comando `executable = master.exe` comunica quindi a Condor che il programma *master* è `master.exe`; esso verrà avviato sulla macchina alla quale il job viene sottomesso.

I comandi di `input`, `output` ed `error` indicano rispettivamente i file da utilizzare come *standard input*, *output* ed *error* (normalmente questi file non comprendono l'*output* generato dai *worker* a meno che il *master* non invochi la primitiva `pvm_catchout()`).

```
#########################################
## PVM example submit description file ##
#########################################
universe = PVM
executable = master.exe
input = "in.dat"
output = "out.dat"
error = "err.dat"
##  Machine class 0  ##
Requirements = (Arch == "INTEL") && (OpSys == "LINUX")
machine_count = 2..4
queue
##  Machine class 1  ##
Requirements = (Arch == "SUN4x") && (OpSys == "SOLARIS26")
machine_count = 1..3
queue
##  Machine class 2  ##
Requirements = (Arch == "INTEL") && (OpSys == "SOLARIS26")
machine_count = 0..3
queue
```

Figura 2.3: ***Submit description file* per l'esecuzione di un job PVM in ambiente Condor.**

Questo file di comandi specifica anche che la macchina virtuale consisterà di tre diverse classi di macchine; la classe 0 sarà formata da macchine con processori INTEL e sistema operativo Linux, la classe 1 sarà formata da macchine con processori SUN4x (SPARC) e la classe 2 da macchine con processori INTEL, quest'ultime due entrambe con sistema operativo Solaris.

Attraverso la direttiva `machine_count = <min>..<max>` si desidera invece che, prima di avviare il *PVM master*, si debbano avere disponibili almeno `<min>` macchine per ogni classe specificata seppur ne vengano richieste un numero pari a `<max>`.

Infine, il comando `queue` deve essere inserito al termine di ogni classe

della macchina virtuale.

2.1.5 Condor-G

Condor-G offre un potente strumento per interfacciare *Condor* con risorse *Grid*, permettendo agli utenti di inviare e gestire job in una griglia computazionale.

Condor-G combina i protocolli di gestione delle risorse implementati dal *Globus Toolkit* ed i metodi di gestione dei job in Condor.

Condor-G permette agli utenti di considerare una griglia computazionale come una risorsa locale; per cui 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`, `condor_q` e `condor_status`.

In figura **2.4**, viene mostrato come *Condor-G* interagisce con i protocolli *Globus* (che analizzaremo più avanti nel dettaglio); *Condor-G* ha un proprio *GASS server* utilizzato per trasferire l'eseguibile, lo *standard input*, *output* ed *error* verso e da il sito di esecuzione remoto, utilizza il protocollo GRAM per contattare il *Globus Gatekeeper* remoto (e per monitorarne il progresso) e richiede che un nuovo job venga avviato.

Condor-G inoltre individua e gestisce localmente tutte le eccezioni, come ad esempio il *crash* remoto delle risorse e dei job.

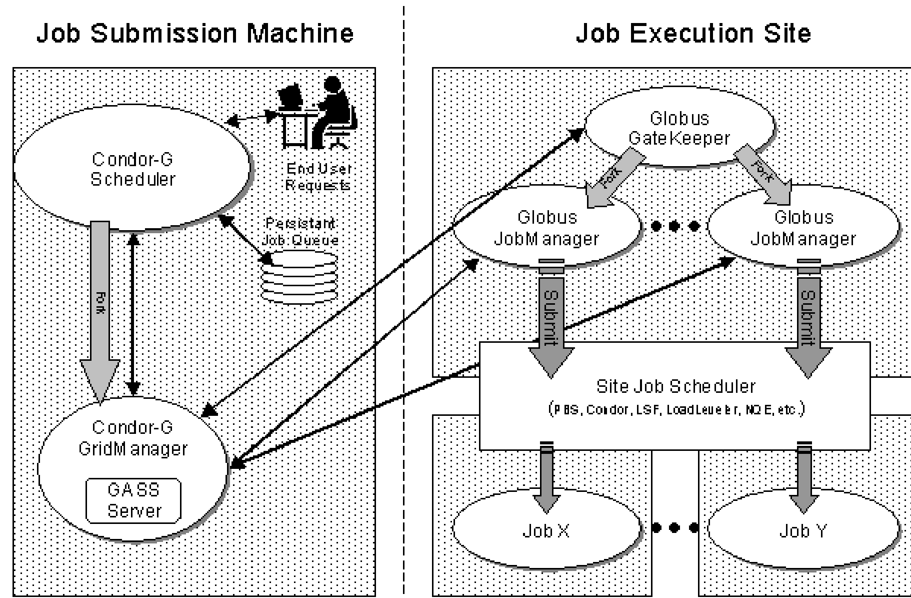

Figura 2.4: **Esecuzione remota di Condor-G sopra risorse Globus.**

Installazione

Esistono due modalità di installazione per lavorare con Condor-G; la prima sfrutta un'installazione completa di Condor alla quale si applica un ulteriore modulo per sottomettere job all'universo Globus mentre la seconda prevede l'installazione, all'interno di un ambiente Globus, di una versione di Condor-G appositamente cablata secondo la *Globus Packaging Technology* (GPT).

In quest'ultimo caso non si avranno però le funzionalità complete di un normale *pool* Condor; tale modalità infatti è pensata per ambienti dove la gestione dei job avviene attraverso Globus stesso.

Per ottenere ciò è quindi necessario configurare Globus affinchè i job in entrata attraverso il *Gatekeeper* siano effettivamente gestiti dall'ambiente

Condor sottostante; ciò avviene definendo un nuovo *jobmanager* in Globus, il quale sarà invocato da Condor-G al momento della sottomissione del job.

I principali *step* quindi prevedono:

- Installazione di un nuovo *jobmanager*:

```
# $GLOBUS_LOCATION/setup/globus/setup-globus-gram-job-mana
ger -type=condor;
```

- Modifica del nuovo *jobmanager* per inserire i parametri relativi al tipo di sistema operativo ed all'architettura (in questo contesto si presuppone che il *pool* Condor sia omogeneo):

```
# cat $GLOBUS_LOCATION/etc/grid-services/jobmanager-condor
...
-condor-arch INTEL -condor-os LINUX;
```

- Attivazione del nuovo *jobmanager*:

```
# $GLOBUS_LOCATION/setup/globus/setup-globus-gram-reporter
-type=condor.
```

Esecuzione dei job nel *Globus Universe*

Per sottomettere job a Globus attraverso Condor è necessario avere a disposizione le giuste credenziali; in particolare si avrà bisogno di un certificato X.509 autenticato ed allocato sulle risorse Globus.

Un tipico *submit description file* è illustrato in figura **2.5**.

```
executable = myjob
globusscheduler = gw.grid.unipg.it/jobmanager-condor
universe = globus
output = myjob.out
log = myjob.log
queue
```

Figura 2.5: ***Submit description file* per l'esecuzione di un job nell'ambiente Grid implementato da Globus mediante Condor-G.**

In questo caso, l'eseguibile del programma sarà trasferito dalla macchina locale al sistema remoto (default); è quindi necessario ricordarsi che il file dovrà essere compilato per la piattaforma di destinazione.

Il job sarà quindi inviato allo *scheduler* Condor di Globus, ovviamente specificando il relativo universo `globus` (direttiva obbligatoria).

Condor trasferirà quindi il risultato prodotto dalla macchina remota al file `myjob.out` della macchina locale, registrando tutte le relative operazioni nel file `myjob.log`, anch'esso sulla macchina locale.

Altri parametri molto utilizzati, che possono essere inseriti nel *submit description file* sono:

- `Transfer_Executable = true|false` (per indicare se l'eseguibile andrà o meno trasferito sul sistema remoto);

- `environment = <par1=val1>; .. ; <parN=valN>` (per il passaggio di parametri al programma);

- `globusrsl = (name1=value1) .. (nameN=valueN)` (per impostare ulteriori attributi al job secondo il *Resource Specification Language* - RSL, cfr. cap. 2.2).

Limitazioni

Al momento esistono ancora una serie di limitazioni per la sottomissione di job in un ambiente Globus attraverso Condor-G; principalmente:

- Non esiste la possibilità di utilizzare *checkpoint*;
- Le operazioni di trasferimento file sono limitate, si possono infatti trasferire solamente l'eseguibile del programma ed i file relativi allo *standard input*, *output* ed *error*;
- Non è possibile utilizzare *ClassAd* per operazioni di *matchmaking*;
- I job non rendono disponibile un codice d'uscita;
- Le piattaforme supportate sono solamente Linux, Solaris, Digital UNIX, e IRIX.

2.2 Globus

L'infrastruttura Grid si occupa del coordinamento delle risorse condivise di Organizzazioni Virtuali (VO), cioè organizzazioni dinamiche multi-istituzionali

che operano in un ambiente sicuro.

La condivisione diventa così accesso diretto a computer, dati, software ed altre risorse e non più un semplice scambio di file.

La sicurezza integrata deve garantire l'autenticazione e l'autorizzazione dei partecipanti, il *monitoring* e l'*auditing* delle risorse e relative *policy* tra organizzazioni differenti.

Un'architettura Grid lavora in assenza di locazione e controllo centralizzati, priva dell'esistenza di relazioni di fiducia pre-esistenti o di configurazioni omogenee.

Tale architettura differisce da quella convenzionale distribuita in quanto centra l'attenzione sulla condivisione di risorse su larga scala e su applicazioni innovative orientate verso alte prestazioni.

L'ambizione di **Globus Project** [15] è sostanzialmente quello di aver introdotto nel calcolo computazionale una rivoluzione simile a quella che ha rappresentato il web per Internet; esso infatti offre strumenti per la definizione di griglie computazionali e per lo sviluppo di applicazioni che operano su di esse.

2.2.1 L'architettura Globus

La distribuzione corrente di *Globus* consiste sostanzialmente di due *toolkit*, entrambi necessari:

- *Grid Packaging Toolkit*, GPT;
- Il *Globus Toolkit* versione 2.

Il primo è un insieme di strumenti che sono utilizzati per gestire l'organizzazione dei sorgenti e dei binari del software Globus quindi per decompirimere, compilare, linkare ed installare tutti gli applicativi; la sua funzione ricalca quella di altri *tool* di gestione dei pacchetti come Linux RPM o Solaris pgk*.

Il secondo può essere immaginato con quelli che vengono definiti *The Three Pillars* (fig. **2.6**) dove ognuno di essi rappresenta una componente primaria di Globus e condivide un'unica infrastruttura di sicurezza denominata *Grid Security Infrastructure* (GSI) che tratteremo nel paragrafo conclusivo di questo capitolo.

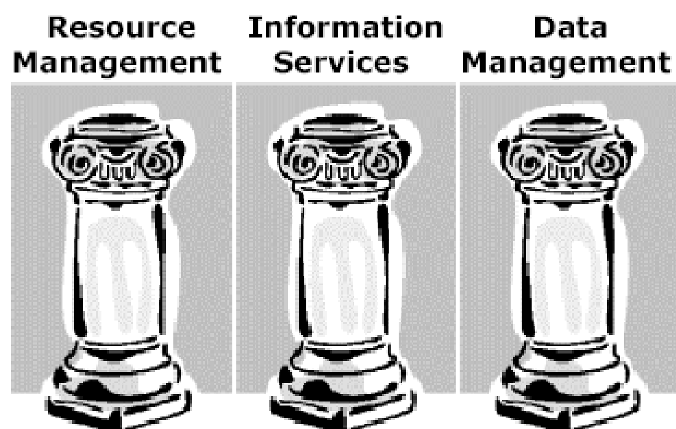

Figura 2.6: **Rappresentazione di *Globus Toolkit* in forma di 3 pilastri.**

Rispettivamente, il *Resource Management Pillar* implementa la gestione ed allocazione delle risorse (GRAM), l'*Information Services Pillar* implementa le funzionalità di monitoraggio e *discovery* delle risorse (MDS) ed il *Data Management Pillar* implementa i servizi di trasferimento dati (GridFTP, Globus Replica Management e Globus Replica Catalog).

Un *Pillar* consiste di un gruppo di *bundle* che a loro volta comprendono diversi pacchetti software (ad esempio il *Resource Management Client bundle* contiene i pacchetti che offrono le funzionalità client per GRAM).

Ognuno di questi pacchetti necessita di essere compilato, linkato ed installato.

GPT provvede alla gestione di queste funzionalità, attraverso altri *tool* che permettono di manipolare ogni *bundle*.

2.2.2 GRAM - Globus Resource Allocation Manager

Il *Globus Resource Allocation Manager* è il livello più basso dell'architettura preposta alla gestione delle risorse; esso rende disponibile la possibilità di eseguire job da remoto, fornendo un'API specifica per la sottomissione, il monitoraggio e la conclusione dei job.

Quando un job viene sottomesso, la richiesta viene inoltrata al *Gatekeeper*, situato sul computer remoto.

Il *Gatekeeper* è una parte di GRAM che risiede sulla risorsa di calcolo

(tecnicamente un server), interpreta la richiesta ricevuta dal client remoto ed inizializza il relativo *jobmanager*.

Esso esegue e tiene sotto controllo il programma remoto comunicandone i cambiamenti di stato all'utente che ha sottomesso il job; quando l'applicazione remota termina, sia correttamente che generando un errore, il *jobmanager* segue la stessa sorte. GRAM si occupa inoltre di:

- Interpretare le richieste in linguaggio RSL, che contengono la descrizione delle risorse necessarie all'esecuzione di un job; tale compito è svolto creando uno o più processi atti a soddisfare la richiesta, oppure rifiutando il permesso di eseguire il job;

- Abilitare la possibilità di controllare e monitorare da remoto lo stato dei job appena creati;

- Aggiornare il *Monitoring and Discovery Service* (MDS) con le informazioni relative alla disponibilità delle risorse che amministra;

- Fornire un'interfaccia comune per l'interazione con i sistemi di gestione delle risorse (nodi) locali, sia che essi siano Condor, PBS, LoadLeveler o LSF.

GRAM tecnicamente è un sistema di tipo *client/server*.

In generale le risorse Grid hanno un server GRAM, il *Gatekeeper*, che rimane in ascolto delle richieste su una porta TCP/IP (normalmente la 2119).

Tutte le macchine, che in qualche modo debbono poter effettuare delle richieste di sottomissione, dispongono invece del client, che si occupa di comunicare le richieste al server utilizzando il linguaggio RSL.

Una delle caratteristiche principali di questo lavoro è che mostreremo come ottenere una griglia computazionale formata da tre cluster, su modello architetturale *Beowulf*, dove, per ogni cluster, verrà eseguita un'unica installazione di Globus (avremo quindi un unico *Gatekeeper* per ogni cluster); in particolare, si mostrerà come effettuare calcolo parallelo attraverso il protocollo MPI sopra l'intera griglia.

RSL - Resource Specification Language

Il *Resource Specification Language* è il linguaggio strutturato utilizzato per descrivere al *manager* delle risorse Grid (GRAM) le caratteristiche del job da eseguire; esso è pertanto il linguaggio con cui le interfacce utente dovranno comunicare con i meccanismi di più basso livello.

Esso offre quindi un'interfaccia comune per la descrizione delle risorse; i vari componenti di GRAM manipolano le stringhe RSL per effettuare le loro funzioni di gestione insieme agli altri applicativi Globus.

Ogni attributo in una descrizione delle risorse si comporta come un pa-

rametro per controllare e definire il comportamento specifico dei singoli job.

L'elemento chiave di RSL è la relazione che associa ad un attributo un valore (per esempio `executable = a.out`) andando a definire strutture complesse che descrivono dettagliatamente le richieste.

A seconda del particolare ambiente in cui ci troviamo, RSL permette di specificare richieste semplici con attributi nome-valore, sequenze di valori, variabili stringa dereferenziate o richieste multiple.

Queste ultime sono usate da DUROC (che verrà analizzato nel paragrafo successivo) per specificare risorse multiple in un ambiente parallelo e distribuito.

Un tipico esempio di stringa RSL utilizzata per il *submit* di un job può essere quello mostrato in figura **2.7**.

```
(* this is a comment *)
& (executable = a.out)
  (directory  = /home/carlo)
  (arguments  = "arg1" "arg 2")
  (count = 3)
```

Figura 2.7: **Esempio di file RSL per la sottomissione di un job in Grid.**

In questo caso il programma che risiede nella *home directory* dell'utente viene eseguito da Globus, attraverso il *Gatekeeper*, per due volte con gli argomenti `arg1` ed `arg2`.

DUROC - Dynamically Updated Request Online Coallocator

DUROC, più semplicemente un ”co-allocatore” di richieste, consiste di un’API (*Application Program Interface*) per coordinare le richieste di risorse distribuite attraverso diversi *Resource Manager*.

Esso quindi coordina l’esecuzione di job attraverso *pool* di gestione multipli; sostanzialmente un tipico *co–allocation agent* si compone di due parti: la prima processa la richiesta vera e propria cercando di determinare come il job dovrà essere distribuito fra i vari *Resource Manager* mentre la seconda cercherà di allocare tali distribuzioni a quello di pertinenza.

Il valore aggiunto che DUROC fornisce a GRAM è quello di implementare una sorta di **atomicità** della richiesta laddove essa si presenti distribuita sopra diversi *Resource Manager*; tale meccanismo è detto *job-start barrier*, il quale provvede appunto a garantire la cosiddetta *inter-manager atomicity.* Un tipico esempio, utilizzato in questo lavoro, è dato dalla script RSL riportata in figura **2.8**.

```
+
( &(resourceManagerContact="fe.giza.unipg.it")
   (count=8)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/mpi_test")
)
( &(resourceManagerContact="gw.grid.unipg.it")
   (count=18)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/mpi_test")
)
( &(resourceManagerContact="bwcw1.hpc.thch.unipg.it")
   (count=16)
   (jobtype=mpi)
   (label="subjob 2")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/mpi_test")
)
```

Figura 2.8: **Esempio di script RSL per l'esecuzione di un job nella griglia.**

In questo caso, viene eseguito il programma parallelo `mpi_test` sopra una griglia formata da tre cluster (si noti un indice DUROC diverso per ogni GRAM) per una complessiva capacità elaborativa che prevede la disponibilità di 42 processori distribuiti fra i 25 nodi a disposizione.

GASS - Global Access to Secondary Storage

Il *Globus Access to Secondary Storage* è il sistema principale utilizzato da Globus per fornire, ad utenti e programmi, gli strumenti necessari per accedere ai dispositivi di memorizzazione di massa delle risorse [16].

Esso fornisce essenzialmente due funzionalità: la prima consiste nel mettere a disposizione dell'utente un protocollo, denominato *x-gass*, tramite il quale accedere ai file sulla macchina remota.

Questo può avvenire tramite l'utilizzo di appositi programmi forniti nel *Globus Toolkit*, oppure, all'interno di un software creato dall'utente, tramite apposite chiamate di libreria che andranno a sostituire le chiamate classiche di lettura e scrittura su file.

La seconda funzionalità fornisce invece al sistema la possibilità di creare e gestire in modo trasparente, sia per l'utente che per l'applicazione, un sistema di cache remota dei dati.

In questo modo, un'applicazione lanciata dall'utente registrerà temporaneamente sia i dati in *input* che il proprio *output* sul disco della macchina che esegue l'elaborazione (snellendo l'*overhead* generato dal trasferimento dati da una macchina all'altra).

Al momento della sottomissione del job, l'utente può comunicare (tramite RSL) al sistema le proprie preferenze in merito alla gestione della cache,

specificando se l'*output* debba essere restituito durante l'esecuzione del programma, nel qual caso si parla di modalità interattiva, oppure se debba essere conservato nella cache fino ad una successiva richiesta da parte dell'utente (modalità *batch*).

Questo sistema di cache fornisce i seguenti vantaggi: velocizza le operazioni di I/O, rende trasparenti le operazioni di gestione dei file remoti, permette l'implementazione di un efficace sistema di trasporto degli eseguibili, uniforma la gestione dei processi batch o interattivi e permette di controllare gli eventi disastrosi.

Per quanto riguarda il primo punto, esso dipende sostanzialmente dal fatto che risulta ampiamente riconosciuta la maggior velocità di esecuzione complessiva di un processo che debba accedere, in lettura o in scrittura, a dati registrati in un posto "vicino", rispetto ad un analogo processo che acceda ai dati sulla macchina di destinazione situata generalmente "lontano" dal luogo di elaborazione.

Il secondo punto riguarda il fatto che non sempre il modo in cui gli utenti e le applicazioni gestiscono i file è il più efficente o anche solo il più adatto ad un sistema di elaborazione come quello qui implementato.

Questo sistema di cache permette invece di creare un filtro che traduce l'organizzazione dei file usata dagli utenti in un sistema più efficiente per una macchina, ma totalmente incomprensibile per un uomo; tecnicamen-

te, questo sistema permette poi di trasportare non solo i dati necessari a un programma per funzionare, ma l'eseguibile stesso del programma, che verrà mantenuto nella *cache* il tempo necessario per eseguire l'elaborazione ed essere poi cancellato.

Per quanto riguarda i processi *batch* o interattivi, essi diventano essenzialmente la stessa cosa; l'unica differenza consisterà nel fatto che l'*output* verrà restituito immediatamente oppure in un secondo momento.

L'ultimo punto riguarda il caso in cui un processo termini precocemente, quale che sia il motivo; in questo caso il sistema, grazie al particolare modo in cui sono stati denominati i descrittori dei file, è in grado di sapere con precisione quali di essi appartenevano al job deceduto e può quindi cancellarli automaticamente, evitando così di intasare la cache con "dati sporchi".

C'è da dire comunque che tale sistema aggiunge un *overhead* nelle operazioni di I/O e comporta la necessità di fornire sufficiente spazio disco sulle macchine che eseguono le elaborazioni.

Questo perchè, sostanzialmente, i file vengono scritti un numero maggiore di volte rispetto a quelle strettamente indispensabili in un sistema senza cache; ciò comunque non rappresenta un controsenso rispetto a quanto detto in precedenza perchè, se è vero che un sistema di questo tipo rende più lenta l'esecuzione di un programma, rispetto a quello che si avrebbe in un'esecuzione locale, è anche vero che i benefici che si ottengono tramite l'e-

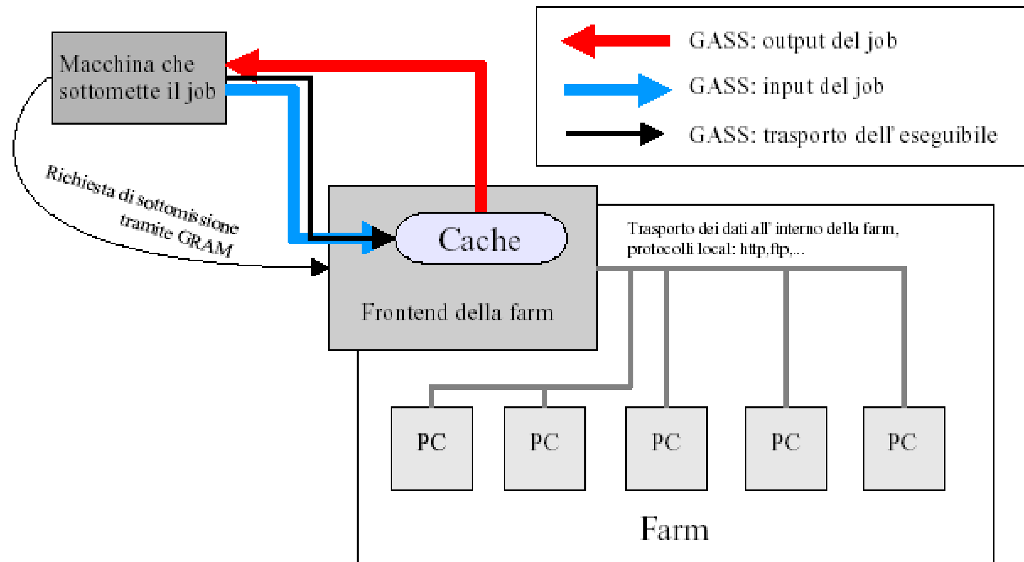

Figura 2.9: **Schema di utilizzo del protocollo GASS e del sistema di *caching*.**

levata distribuzione del calcolo superano di gran lunga l'entità dell'*overhead* accumulato.

Infine è opportuno notare che GASS, così come gli altri protocolli Grid, operano trasferimenti di dati su media e larga scala; infatti il trasporto e la gestione dei file nelle *farm* [17] locali (livelli 3/4 stack ISO-OSI) avvengono utilizzando i protocolli standard comuni, dipendenti dalle architetture locali e fuori controllo degli utenti remoti.

In figura **2.9** è riportato un esempio dell'utilizzo del protocollo GASS all'interno della griglia computazionale.

E' opportuno notare come la macchina che sottomette il job può essere la macchina dell'utente così come un *resource broker* che fa da intermediario;

in figura si nota come GASS non si occupa affatto del trasporto dei file all'interno della *farm*, ma si limita a creare i file nella *cache* del *Front-End*.

2.2.3 MDS - Monitoring and Discovery Service

Globus Monitoring and Discovery Service offre gli strumenti necessari per costruire un'infrastruttura informativa basata su LDAP (*Lightweight Directory Application Protocol*) conforme allo standard ISO/OSI X.500; MDS utilizza infatti il protocollo LDAP per gestire in maniera uniforme le varie risorse (spesso organizzate ad albero, secondo le specifiche LDAP), a livello informativo, di un definito *namespace* che può coinvolgere diverse organizzazioni (di solito strutturate in modo gerarchico).

MDS si costituisce di due componenti principali:

- Il *Grid Resource Information Service* (GRIS), che provvede a raccogliere tutte le caratteristiche (dinamiche o permanenti) di una griglia computazionale;
- Il *Grid Index Information Service* (GIIS) che provvede a mettere insieme i diversi GRIS offrendo un'unica immagine coerente agli utenti ed alle applicazioni Grid.

In un cluster, una modalità particolare di utilizzo (come verrà ampiamente descritto nel quarto capitolo) consiste nel considerare la griglia computazio-

nale il singolo nodo che ha nel *Front-End* l'unico GIIS, il quale permetterà quindi una visione coerente delle risorse globali dell'intero cluster (configurazione dei nodi, memoria disponibile, spazio disco, stato, tipo e numero di processori e così via).

Il servizio MDS ha un carattere prettamente gerarchico; in esso infatti, i dati collezionati dalle singole risorse risalgono una catena piramidale di server LDAP fino a giungere al database di livello più alto (in strutture complesse, di solito non esiste un unico GIIS che riceve le informazioni da tutti i GRIS esistenti; per di più è sempre possibile agganciare un GRIS esistente ad uno di livello più alto).

Il funzionamento di MDS (schematizzato in fig. **2.10**), che può anche essere installato separatamente dal pacchetto Globus, si basa su diversi file di configurazione:

- **grid-info.conf**

  Definisce i parametri di configurazione per il servizio MDS locale;

- **grid-info-resource-ldif.conf**

  Specifica quali caratteristiche verranno monitorate dal GRIS locale e quindi inviate al GISS;

- **grid-info-resource-register.conf**

Indica quale debba essere il server GIIS al quale il servizio GRIS locale si dovrà registrare;

- **grid-info-site-giis.conf**

  Permette ad un GIIS di inizializzare le informazioni senza dover attendere la registrazione dei vari GRIS;

- **grid-info-site-policy.conf**

  Definisce la politica di autorizzazione di un GISS; specifica quali macchine (o domini) possono registrarvisi e la porta sulla quale dovrà girare il servizio (di solito la 2135);

- **grid-info-slapd.conf**

  Specifica ad LDAP i vari parametri di GRIS e GIIS e ne stabilisce gli schemi informativi; specifica i *back-end* supportati dal server `slapd` e setta il *binding* anonimo.

Una volta configurato l'ambiente MDS, è necessario avviare il demone con il comando:

```
# {GLOBUS_LOCATION}./sbin/SXXgris start
```

Questo programma chiama la *script* `grid-info-slapd`, la quale definisce le variabili d'ambiente necessarie ed invoca il *server* `slapd`, che a sua volta legge il file di configurazione `grid-info-slapd.conf` che determina tutti i

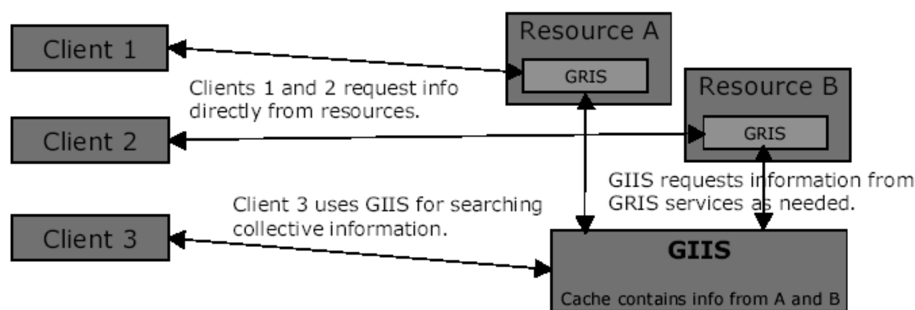

Figura 2.10: **Lo schema di funzionamento di *Monitoring and Discovery Service* (MDS).**

file di configurazione ulteriori da consultare.

Per effettuare una *query* ad un nodo locale si può utilizzare il comando:

```
# grid-info-search -x -b "Mds-Vo-name=local, o=grid"
```

In questo caso, si otterranno tutti gli oggetti e le risorse definite, esportate dal GRIS, sulla macchina locale.

2.2.4 Data Management

Le applicazioni scientifiche richiedono spesso l'accesso ad una grande mole di dati (Terabyte o Petabyte); tali applicazioni inoltre richiedono l'accesso distribuito ai dati stessi (per esempio, accesso in diversi posti da diverse persone, ambienti di collaborazione virtuali, etc.):

> *"Access to distributed data is typically as important as access to distributed computational resources."* [18]

Globus tenta di identificare e valutare le tecnologie chiave richieste alle griglie di dati per la computazione; l'intenzione è quella di offrire un'infrastruttura generica di griglia di dati sotto forma di servizi di trasferimento e librerie di gestione apposite.

Allo scopo, il *core* applicativo messo a disposizione è il seguente:

- **GridFTP**:

  Un protocollo sicuro ad alta affidabilità ottimizzato per traferimenti di dati su reti geografiche ad alte prestazioni;

- **Globus Replica Catalog**:

  Un meccanismo per mantenere un catalogo replicato di insiemi di dati al fine di ottenere un accesso alle strutture ed ai dati stessi molto più veloce;

- **Globus Replica Management**:

  Un software di gestione per manipolare e far cooperare insieme *GridFTP* e *Replica Catalog*.

GridFTP in particolare è un protocollo basato sul popolare FTP al quale aggiunge alcune funzionalità quali ad esempio:

- Sicurezza sui canali di controllo e trasmissione dati;
- Canali di trasferimento multipli per trasferimenti paralleli;

- Trasferimenti parziali di file;
- Autenticazione dei canali di trasmissione dei dati;
- Riusabilità dei canali trasmissivi;
- *Pipeling.*

La sua implementazione essenzialmente si basa su due nuove API e relative librerie: `globus_ftp_control` e `globus_ftp_client`: la prima offre un servizio a basso livello per l'implementazione del protocollo FTP sia lato client che lato server; la seconda estende tali funzionalità verso un ambiente parallelo introducendo estensioni specifiche per la sicurezza ed il miglioramento delle prestazioni [19].

In aggiunta a tali librerie, si ha un tool chiamato `globus-url-copy` (basato anch'esso su una libreria specifica, la `globus_gass_copy`) che integra GridFTP, HTTP e l'I/O locale in maniera sicura, permettendo quindi di sfruttare qualsiasi loro combinazione.

Infine è anche possibile utilizzare un'adattamento del popolare FTP server `wu-ftpd` (*Washington University*); tale versione infatti implementa le principali funzionalità del protocollo GridFTP come la sicurezza (che affronteremo nel successivo ed ultimo paragrafo), il trasferimento dei dati in parallelo ed i trasferimenti parziali.

2.2.5 GSI - Grid Security Infrastructure

Grid Security Infrastructure è il meccanismo sul quale si basano le procedure di autenticazione degli utenti e delle risorse; esso è pertanto la struttura fondamentale su cui si appoggiano tutte le altre.

Ogni volta che due entità, siano esse persone o risorse di calcolo, vogliono comunicare utilizzando la griglia come infrastruttura, dovranno mutuamente autenticarsi, per essere sicure ciascuna dell'identità della controparte.

Questo risulta necessario per garantire che solo gli utenti autorizzati possano utilizzare le risorse di calcolo condivise, certificando la propria identità.

Non è comunque strettamente necessario che le comunicazioni siano sempre crittografate: tale attività infatti comporterebbe un notevole aumento delle dimensioni dei dati, spesso inutilmente, come nel caso di dati grezzi di esperimenti o test.

Come detto, i tre *pillar* di Globus utilizzano i servizi messi a disposizione dal *Grid Security Infrastructure* per realizzare la comunicazione tra reti aperte in modalità sicura.

GSI fornisce quindi una serie di servizi utili per l'ambiente Grid tra cui l'iscrizione individuale degli utenti per accedere all'ambiente e l'autenticazione tra client e server preventiva.

È possibile riassumere nei punti seguenti i vantaggi che GSI offre all'ambiente

Grid:

- Comunicazioni sicure (autenticate e confidenziali) tra i vari elementi di una griglia;
- Supporto per la sicurezza nelle comunicazioni tra soggetti di organizzazioni diverse;
- Possibilità di effettuare l'iscrizione individuale per gli utenti dell'ambiente Grid, ivi inclusa la delega delle credenziali per calcoli che coinvolgono risorse o siti multipli.

GSI è basato su crittografia a chiave pubblica, certificati X.509 e sul protocollo di comunicazione SSL (*Secure Socket Layer*); la sua implementazione Globus aderisce a GSS-API (*Generic Security Service API*), lo standard API per la sicurezza dei sistemi promosso dall'*Internet Engineering Task Force* (IETF).

Il sistema di certificazione X.509

Il sistema di autenticazione di Globus segue le direttive ratificate nello standard denominato X.509 dalla *International Standards Organization* (ISO), facente parte della serie di standard X.500 che definiscono servizi di directory distribuiti.

In questo standard sono definite le specifiche necessarie per un'autenticazione basata sul sistema dei certificati, detti anche *Digital ID*; essi sono l'equivalente elettronico di un passaporto ed in pratica servono a provare che qualcuno sia effettivamente chi sostiene di essere.

Inoltre essi permettono di effettuare comunicazioni riservate tramite l'utilizzo della tecnica crittografica a chiave pubblica; grazie allo standard X.509, i certificati rappresentano un sistema di autenticazione indipendente dal software utilizzato, cosa che ovviamente ne ha favorito una larga diffusione.

Come si è accennato, il sistema di certificazione X.509 viene utilizzato per due scopi distinti: garantire la sicurezza della comunicazione, cioè impedire che qualcuno possa decifrare o alterare i dati trasferiti tra le due parti, e verificare l'identità dei corrispondenti, ovvero garantire all'utente di una transazione elettronica che l'entità, sia essa una persona fisica od una macchina, all'altro capo della comunicazione, sia la persona effettivamente desiderata.

Nel sistema di certificazione X.509 l'implementazione di queste due funzionalità avviene tramite l'utilizzo di una tecnica chiamata crittografia (asimmetrica) in chiave pubblica: questa prevede che ogni utente possegga due chiavi che servono per criptare o decriptare i messaggi.

La chiave pubblica viene distribuita a chiunque voglia comunicare in modo sicuro con gli altri e rappresenta la chiave con la quale verranno cifrati gli

eventuali dati o messaggi destinati a noi stessi (quindi solamente noi sapremo decifrarli con la nostra chiave privata).

La chiave privata è custodita dall'utente e mantenuta segreta, tipicamente essa è conservata sulla macchina personale dell'utente ed è anch'essa criptata con una password; oltre che decifrare messaggi cifrati con la rispettiva chiave pubblica (è importante ricordare che qualsiasi dato criptato da una delle due chiavi può essere decifrato soltanto dall'altra chiave) essa garantisce l'autenticità della comunicazione da noi instaurata: attraverso la chiave pubblica infatti è possibile accertarsi che il mittente firmatario sia effettivamente chi dice di essere.

La *Certification Authority*

I certificati vengono rilasciati da autorità apposite, chiamate *Certification Authority*, comunemente denominate CA, le quali godono della fiducia di entrambe le parti.

Una *Certification Authority* ha il compito di verificare, al momento del rilascio del certificato, la veridicità delle informazioni ivi contenute, cioè l'identità della persona alla quale il certificato viene rilasciato.

La CA riceve dall'utente, generalmente via E-Mail, una copia della chiave pubblica, generata con un apposito software che viene utilizzato per compilare il certificato, inserendovi le seguenti informazioni: chiave pubblica dell'u-

tente, nome dell'utente, data di scadenza del certificato, nome e firma digitale della CA e numero di serie del certificato.

E' pratica comune che una CA possa a sua volta avere un certificato firmato da un'altra *Authority*, che ne garantisce l'affidabilità; si verrà così a creare una catena di autenticazione fra diverse CA, piuttosto che (come nel nostro caso) si potrà stabilire una relazione di *trust* paritetico fra più *Certification Authority*.

Il sistema di autenticazione in Globus

Il problema quindi consiste nel fornire un sistema di autenticazione, accesso controllato alle risorse ed integrità dei dati; in particolare:

- **One time login**: un utente deve autenticarsi un'unica volta (ad esempio all'inizio della giornata o al momento del lancio del job) e poter lanciare programmi che acquisiscono risorse in tutta la griglia senza dover ripetere la procedura successivamente;
- **Protezione delle credenziali**: le credenziali personali (password o chiavi private) dell'utente devono rimanere al sicuro in ogni momento;
- **Esportabilità del codice**: è necessario che il codice utilizzato non vada in contrasto con le legislazioni dei paesi che collaboreranno ai vari progetti;

- **Sistema di credenziali/certificazione uniforme**: è necessario utilizzare uno standard comune, almeno per il sistema di mutua identificazione tra domini diversi (X.509 per le credenziali ed il software *SSLeay* per la certificazione).

I primi due punti sono stati risolti tramite l'introduzione del concetto di *Proxy User*: esso non è altro che un'entità interna all'architettura (visibile per intero in fig. **2.11**) che fa le veci di un utente.

In pratica, esso consiste di un processo speciale, iniziato dall'utente, che viene eseguito su un computer locale; il meccanismo tramite cui l'utente *proxy* ottiene gli stessi privilegi dell'utente che rappresenta, si basa sul sistema delle catene gerarchiche di certificati: al momento del *login* l'utente provvede a creare il suo *proxy* attraverso un certificato X.509 firmato elettronicamente da lui stesso.

Il *proxy user* disporrà quindi di una chiave privata ed una pubblica; quella privata rimarrà custodita all'interno della macchina che ha creato il *proxy* (tipicamente in `/tmp/x509up_u<UID>` dove l'UID corrisponde all'identificativo dell'utente sul sistema), mentre quella pubblica sarà contenuta nel certificato firmato dall'utente, il quale diviene pertanto la CA che garantisce l'identità del *proxy*.

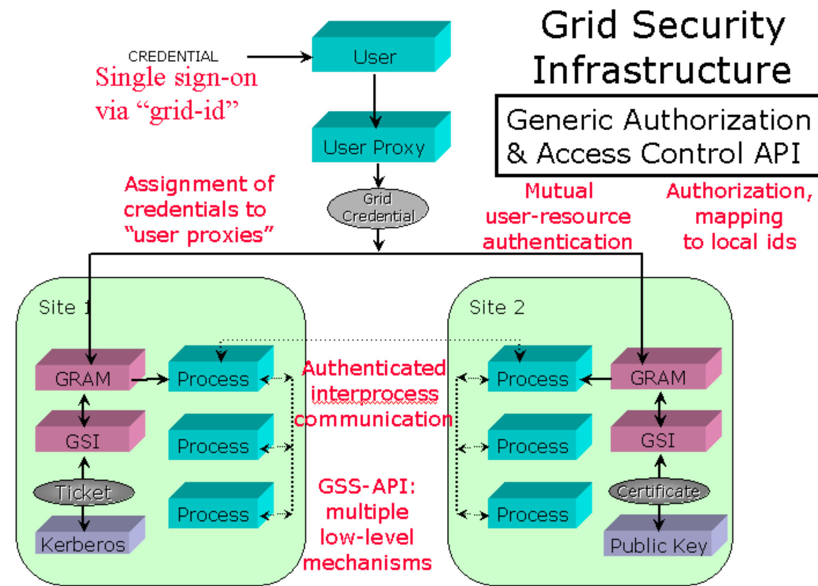

Figura 2.11: **La *Globus Security Infrastructure* (GSI).**

Infine, il certificato del *proxy* (detto anche *proxy certificate*) conterrà, oltre alla chiave pubblica firmata dall'utente, anche l'intervallo di tempo per cui tale certificato avrà validità nonchè ulteriori informazioni relative ai permessi del *proxy*.

La richiesta di un certificato per un'organizzazione (*distinguished name*) avviene attraverso il comando `grid-cert-request`, come mostrato in figura **2.12**.

L'operazione successiva, come indicato, consisterà nell'inviare la richiesta all'appropriata *Certification Authority* la quale provvederà a generare il certificato richiesto ed inviarlo al richiedente.

```
# grid-cert-request
...
Generating a 1024 bit RSA private key
...
writing new private key to '/home/carlo/.globus/userkey.pem'
Enter PEM pass phrase:
...
A private key and a certificate request has been generated with the
subject:
/O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/OU=grid.unipg.it/
CN=Carlo Manuali
Your private key is stored in ~/.globus/userkey.pem
Your request is stored in ~/.globus/usercert_request.pem
Please e-mail the request to the Globus Dynamics UniPG CA:
ca@grid.unipg.it
You may use a command similar to the following:
# cat ~/.globus/usercert_request.pem | mail ca@grid.unipg.it
```

Figura 2.12: **Esempio di richiesta di un certificato alla *Certification Authority* (CA).**

Quindi, prima di poter iniziare a lavorare, è necessario inizializzare il relativo *proxy* attraverso il comando `grid-proxy-init` (si noti come venga definita una validità temporale, peraltro modificabile) come evidenziato in figura **2.13**.

```
# grid-proxy-init
Your identity: /O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/
OU=grid.unipg.it/CN=Carlo Manuali
Enter GRID pass phrase for this identity:
Creating proxy ...................................... Done
Your proxy is valid until Thu Sep 12 00:03:11 2002
```

Figura 2.13: **Inizializzazione dell'ambiente di lavoro Globus (generazione del *proxy*).**

In un sistema Globus esistono quindi diversi tipi di certificati:

- Lo **User Certificate**, memorizzato (come si è visto) in `$HOME/.globus/usercert.pem`;
- Il **Globus CA Certificate**, certificato della stessa *Certification Authority*, memorizzato in `/etc/grid-security/certificates/<HASH>.0`;
- L' **Host Certificate**, certificato che autentica le macchine, memorizzato in `/etc/grid-security/hostcert.pem`, leggibile da tutti;
- L'**LDAP Server Certificate**, utilizzato da MDS, memorizzato in `$GLOBUS_LOCATION/etc/server.cert`.

Una volta ottenuti ed installati tutti i certificati necessari, eventualmente definendo, come nel nostro caso, una CA propria (infatti si può comunque utilizzare la *Certification Authority* di Globus stesso), si terminerà la definizione dell'ambiente GSI della griglia computazionale (operazione quindi che investe l'amministratore di sistema e non l'utente) con la procedura `setup-gsi`, che popola il sistema dei necessari file di configurazione ed installa i relativi certificati ottenuti.

Capitolo 3

Programmazione Parallela in ambiente Grid: MPICH-G2

Negli ultimi anni sono state delineate le principali caratteristiche architetturali delle macchine parallele conformemente ai modelli teorici di calcolo parallelo ed ai metodi usati per misurarne le prestazioni.

Sebbene attualmente vi siano macchine parallele che vengono impiegate come macchine dedicate per supportare applicazioni specifiche (trattamento di immagini, robotica, visione, etc.), è sempre più diffusa la necessità di avere a disposizione sistemi di tipo *general-purpose*.

3.1 Le metodologie ed i paradigmi odierni

La realizzazione di programmi paralleli richiede di affrontare e risolvere problemi che non sono presenti nella programmazione sequenziale, come la crea-

zione di processi, la loro sincronizzazione, la gestione delle comunicazioni tra processi, la prevenzione dello stallo (*deadlock*) e la terminazione dei processi che compongono il programma parallelo.

Processare una mole sempre più grande di dati fa nascere l'esigenza di frammentare l'unico flusso di *input* dei dati stessi in flussi multipli paralleli da distribuire su diverse macchine.

Tipicamente l'I/O resta essenzialmente sequenziale: i dati in ingresso vengono forniti ad un processo *master* il quale li distribuisce sui vari nodi per poi raccogliere i diversi output in un unico flusso di dati in uscita, ottenendo il risultato.

Si hanno varie tecniche di programmazione per l'ambiente parallelo e diverse librerie aggiuntive fornite da alcuni linguaggi come C/C++, Fortran o Java che implementano il concetto precedentemente espresso di *metaelaboratore*; esse verrano impiegate a seconda della particolare architettura hardware ed al tipo di comunicazione esistente tra i processori.

Quando si considera il concetto di *metacomputing*, si deve precisare se si opera in ambiente locale (LAN), metropolitano (MAN) o geografico (WAN); a seconda dei casi si avrà la necessità di essere supportati da sistemi software sempre più avanzati e complessi.
In generale è possibile individuare quattro fasi [20]:

- Il singolo problema viene decomposto in sottoproblemi;
- I nodi comunicano fra di loro a seconda dei risultati;
- I risultati vengono raggruppati e riassegnati ai nodi;
- Avviene il coordinamento finale.

La decomposizione propria della prima fase può avvenire in due modi:

- **Funzionale**: l'algoritmo viene suddiviso in più parti;
- **A livello di domini**: i dati vengono suddivisi in più parti.

La comunicazione tra i nodi nella seconda fase è diversa a seconda dell'organizzazione della memoria, si parla quindi di:

- **Memoria condivisa** (*shared*), dove si ha un accesso simultaneo e più veloce alla memoria, ma si dovranno gestire le problematiche di contesa delle risorse tramite opportuni meccanismi di *locking*;
- **Memoria distribuita**, dove si ha una scalabilità dipendente dal numero di processori, non sono presenti le precedenti problematiche di *locking* per l'accesso alla memoria (la contesa è infatti riconducibile alla particolare rete di interconnessione dei nodi), ma si dovrà gestire l'accesso ai processori per il passaggio dei dati tramite scambio di messaggi (*message passing*), tipicamente con le funzioni *send* e *receive*.

Generalmente, l'architettura a memoria condivisa risulta conveniente quando si hanno sistemi di calcolo con un numero di processori minore di trenta; oltre tale numero infatti la contesa per l'accesso alla memoria diventa il "collo di bottiglia" che pregiudica il raggiungimento di adeguate prestazioni.

È possibile individuare due paradigmi di programmazione, entrambi indipendenti dall'architettura:

- *Message Passing*, dove si hanno diverse tecniche di programmazione come MPI (C/Fortran), MPL (tecnica di programmazione proprietaria degli elaboratori IBM), OpenMP e PVM;
- *Data Parallel*, dove si parla essenzialmente di FORTRAN90/HPF (le direttive fornite al compilatore fanno sì che esso esegua da solo il programma in parallelo).

Message Passing risulta più efficiente in presenza di architetture a memoria distribuita mentre risulta più conveniente il paradigma di *Data Parallel* quando si ha memoria condivisa.

3.1.1 Gli indici di valutazione

L'aumento della velocità ottenibile dall'esecuzione in parallelo su P processori di un programma è regolato dalla *legge di Amdhal* ed è comunque compito del

programmatore progettare l'algoritmo in modo tale da ottenere il massimo beneficio da architetture multi-processore.

La *legge di Amdhal* afferma che se si esegue in parallelo un programma su n processori, il tempo calcolato $T(n)$ è dato dalla relazione:

$$T(n) = T_s + \frac{T_p}{n}$$

dove T_s è il tempo necessario all'esecuzione della parte di codice sequenziale mentre T_p è il tempo necessario per l'esecuzione della parte parallela (e quindi solo quest'ultima beneficierà dell'aumento del numero di processori).

Si può comunque dire che, dato un algoritmo seriale, sia T_s il suo tempo di esecuzione su un certo processore e T_p il tempo di esecuzione dell'equivalente algoritmo parallelo su n processori dello stesso tipo, indicando con S lo *speedup* e con E l'efficienza ottenuta, si ha:

$$S = \frac{T_s}{T_p}$$
$$E = \frac{S}{n}$$

Una macchina parallela ideale produrrebbe quindi un'efficienza uguale all'unità con un programma completamente parallellizzabile; in realtà anche qui intervengono, nel calcolo del tempo di computazione, fattori quali l'*overhead* del software (maggior codice nella versione parallela rispetto a quella seriale), il bilanciamento del carico (*speedup* limitato generalmente dal nodo più lento) e l'*overhead* della comunicazione fra i nodi.

3.2 MPICH-G2

La comunità scientifica, soprattutto grazie allo sviluppo delle librerie MPI e PVM di *message passing*, mette a disposizione un grande numero di funzioni atte a rendere più semplice lo sviluppo di applicazioni parallele.

Fra queste ricordiamo la vasta raccolta di strumenti messi a disposizione dall'*Advanced Computing Lab* del *Los Alamos National Laboratory*, ACL, che sono di utilità generale, ed alcune librerie matematiche, come LINPACK e ScaLAPACK, che implementano routine parallele di algebra lineare.

Nell'ambito delle librerie MPICH riveste un importante ruolo LAM (*Local Area Multicomputer*) che presenta un ambiente di esecuzione completo per programmi basati su MPI.

Tutto questo insieme di strumenti possono solitamente convivere nello stesso ambiente, in particolare è possibile far cooperare insieme le librerie MPI con il software di Grid *Globus Toolkit*, "ottenendo" MPICH-G2.

MPICH-G2 è una nuova implementazione dello standard MPI v1.1 orientato alle applicazioni Grid.

In particolare MPICH-G2 definisce un nuovo *device*, chiamato *globus2*, che estende l'insieme dei *device* di MPICH.

Attraverso i servizi Globus, MPICH-G2 permette di eseguire applicazioni MPI sopra diversi gruppi di macchine, anche composti da host tra loro

eterogenei.

Laddove necessario infatti, i messaggi scambiati tra diverse architetture vengono adeguatamente convertiti ed è possibile effettuare comunicazioni multi-protocollo, via TCP, tra macchine remote che differiscono tra di loro nella soluzione MPI adottata localmente.

3.2.1 Le caratteristiche principali di MPICH-G2

A seconda della classe di problemi con la quale si ha a che fare ed a fronte della possibilità di accedere a diversi computer attraverso reti metropolitane (MAN) o geografiche (WAN), una griglia computazionale si presenta come l'infrastruttura operativa di eccellenza per il calcolo ad alte prestazioni.

In uno scenario tipico, si ha l'esigenza di far cooperare insieme diverse macchine per aumentare la potenza elaborativa e quindi diminuire il tempo necessario per la loro esecuzione.

MPICH-G2 permette di aggiungere risorse di calcolo attraverso **Internet** in maniera efficiente e controllata (considerando ad esempio la latenza della rete o la quantità di banda passante fra i diversi nodi).

Topology-aware Collective Operations

MPICH-G2 è quindi di solito utilizzato per eseguire operazioni su macchine separate sia da distanze brevi (LAN) che molto lunghe; si hanno a disposizio-

ne chiamate MPI ottimizzate (come `MPI_Barrier`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter` ed `MPI_Reduce`) che sono coscienti della particolare topologia sulla quale si sta lavorando, chiamate *Topology–aware Collective Operations*, che provano a minimizzare le comunicazioni sul *link* più lento ed a massimizzarle su quello più veloce.

Per ottenere ciò, MPICH-G2 individua una relazione che è alla base dell'effettivo svolgimento di tutte le operazioni:

$$WAN/TCP < LAN/TCP < intra{-}machine/TCP < vendor{-}suppl/MPI$$

Per cercare di capire questa relazione, supponiamo di avere tre macchine, che chiamaremo A, B e C, ognuna con otto processi MPI in esecuzione, dove la prima e l'ultima implementano anche un gestore MPI locale (oltre che MPICH-G2) mentre la seconda no.
Si consideri la chiamata `MPI_Bcast` sopra tutto l'ambiente `MPI_COMM_WORLD`; l'istruzione è implementata attraverso una sequenza di *broadcast* dove per ogni livello (da **lv0** ad **lv3**) ci sono uno o più insiemi di processi (ogni insieme rappresenta un *broadcast* singolo nel quale il primo processo è quello radice), si avrà:

lv0, *WAN-TCP*:

P0, P8, P16

lv1, *LAN-TCP* (in parallelo):

P0*

P8*

P16*

lv2, *intra-machine-TCP* (in parallelo):

P0*

P8,...,P15

P16*

lv3, *vendor-supplied MPI* (in parallelo):

P0,...,P7

P16,...,P23

Presupposto che *broadcast* sopra insiemi formati da un singolo processo sono sostanzialmente delle "non operazioni" ([*], inserite solamente per chiarezza), si noti come, laddove non sia presente un *vendor-supplied MPI* i *broadcast* avverranno, via TCP, fino al terzo livello **lv2** (macchina B), mentre nella prima (A) e terza (C) macchina avverranno fino a **lv3**, quarto ed ultimo livello.

Ora, se ad esempio le macchine A e B appartenessero alla stessa LAN,

MPICH-G2 potrebbe esserne informato con la macro `GLOBUS_LAN_ID = <id>` nella stringa RSL di sottomissione del job; si otterrebbe:

lv0, *WAN-TCP*:

P0, P16

lv1, *LAN-TCP* (in parallelo):

P0, P8

P16*

lv2, *intra-machine-TCP* (in parallelo):

P0*

P8,...,P15

P16*

lv3, *vendor-supplied MPI* (in parallelo):

P0,...,P7

P16,...,P23

In questo caso quindi, si avrebbero solamente due *broadcast* a livello WAN-TCP ed un broadcast (reale) a livello LAN-TCP.

Topology Discovery Mechanism

Alcune applicazioni MPI possono far uso di questa stratificazione creando dei *communicator* che utilizzano queste informazioni circa la topologia; MPICH-G2 infatti offre due nuovi attributi associati ad ogni *communicator*: `MPICHX_TOPOLOGY_DEPTHS` e `MPICHX_TOPOLOGY_COLORS`.

I processi MPI possono quindi comunicare o con una propondità al terzo livello (dove lv0=WAN-TCP, lv1=LAN-TCP ed lv2=intra-machine TCP), utilizzando esclusivamente TCP, oppure al quarto livello sfruttando anche il gestore MPI locale (lv3=MPI).

Attraverso questo meccanismo (illustrato nel dettaglio nel paragrafo **4.3**) è possibile capire se un certo processo A può comunicare con un altro processo B ad un determinato livello (si dirà che i due processi hanno lo stesso colore); in pratica è possibile scoprire progressivamente la topologia della nostra griglia: infatti due processi che comunicano a livello LAN-TCP possono farlo se e solo se appartengono alla stessa LAN, o allo stesso *cluster*, mentre possono comunicare a livello MPI solo se appartengono allo stesso *subjob RSL* di tipo MPI (direttiva `(jobtype=mpi)`).

IP address range

Con MPICH-G2 è possibile specificare una determinata interfaccia di rete attraverso la variabile di ambiente `MPICH_GLOBUS2_USE_NETWORK_INTERFACE`

inserita nel file RSL, ad esempio:

- Indicando un indirizzo IP specifico:

  `(MPICH_GLOBUS2_USE_NETWORK_INTERFACE 141.250.1.252);`

- Indicando una particolare rete o sottorete:

  `(MPICH_GLOBUS2_USE_NETWORK_INTERFACE 141.250.252.0/255`

  `.255.255.0);`

  oppure nella forma:

  `(MPICH_GLOBUS2_USE_NETWORK_INTERFACE 141.250.252.0/24).`

Port range

È possibile specificare un *range* di porte TCP/IP per le varie comunicazioni (a livello TCP) attraverso la variabile di ambiente (anch'essa da specificare attraverso RSL) `GLOBUS_TCP_PORT_RANGE` nel seguente modo: `(GLOBUS_TCP_PORT_RANGE <min max>)` dove **min** e **max** sono gli estremi del *range* che sarà utilizzato dai job.

3.2.2 Come lavora MPICH-G2

Configurare MPICH-G2 insieme a Globus (quindi configurandolo con il *flavor mpi* di Globus) implicitamente sottointende che tutti i programmi "linkati" a MPICH-G2 debbano poi necessariamente utilizzare un'implementazione MPI locale (raggiungendo quindi il quarto livello).

Dato che MPICH-G2 è essa stessa una libreria MPI, gli sviluppatori hanno provveduto ad adottare una politica di *renaming* per tutti i simboli che potevano creare conflitti fra l'MPI locale e MPICH-G2, preprocessando il relativo codice sorgente MPICH-G2 ed eseguendo sostituzioni del tipo:
`{P}MPI_xxx` diventa `{P}MPQ_xxx`
secondo lo standard MPI v1.1 che prevede:

> *"Programs must not declare variables or functions with name beginning with the prefix, MPI_"*

Allo stato attuale comunque, non è possibile utilizzare MPICH-G2 laddove si abbia, come MPI locale, lo stesso MPICH.

Durante l'esecuzione, `MPI_Init` utilizza le barriere DUROC (cfr. cap. 2.2); attraverso di esse ci si assicura che tutti i processi, di tutte le macchine, siano caricati ed avviati prima di proseguire.

In seguito vengono organizzate tutte le comunicazioni (incluse le *Topology–aware Collective Operation*), secondo i propri componenti costituenti, in modalità *point-to-point*, per poi trasferire il controllo al *device* di più basso livello, **globus2**, che quindi gestirà esclusivamente tali richieste.

La scelta del protocollo di trasporto (TCP piuttosto che l'MPI locale) viene fatta in base alla sorgente ed alla destinazione: avremo quindi *vendor-supplied MPI* per messaggi *intra-machine* e TCP per tutti gli altri.

Nell'implementazione presentata in questo lavoro si è esteso il concetto di *intra-machine* considerando come risorsa unitaria ogni intero cluster (potremmo parlare quindi di *intra-cluster*).

In particolare, nella sperimentazione descritta nel quarto capitolo, la griglia computazionale sarà costituita da tre cluster differenti (due aderenti al modell *Beowulf* puro ed uno ibrido *Mosix-Beowulf*) dove, per ognuno è presente:

- Un unico *Globus Gatekeeper*;
- Un'unica installazione MPICH-G2 linkata all'MPI locale LAM-MPI.

Nel nostro caso quindi, la comunicazione avverrà attraverso TCP per i messaggi *inter-cluster* ed attraverso MPI per quelli *intra-cluster*.
In altre parole potremmo dire che:

> **"un job di tipo *mpi* distribuito sulla griglia, comunica attraverso MPICH-G2, "entra" in un cluster attraverso Globus e viene eseguito localmente su tutti i nodi attraverso LAM-MPI"**

3.2.3 Come utilizzare MPICH-G2

Prima di poter utilizzare MPICH-G2 è necessario avere a disposizione:

- Un *account* di sistema;
- Il *Globus Toolkit* e MPICH-G2 installati;
- Un *Globus Gatekeeper* con almeno un *jobmanager* funzionante;
- Un certificato Globus (*Globus_ID*) registrato, dall'amministratore Globus, in `/etc/grid-security/grid-mapfile`.

A questo punto è possibile compilare ed eseguire applicazioni basate su MPICH-G2 attraverso i seguenti passi:

- Compilare l'applicazione su ogni macchina che si intende utilizzare (nel caso di condivisione del *file system* in un cluster, ad esempio via NFS, basta ovviamente compilarla su uno dei nodi), con uno dei seguenti compilatori:

  `<MPICH-G2_INSTALL_PATH>/bin/mpicc` (**C**)

  `<MPICH-G2_INSTALL_PATH>/bin/mpiCC` (**C++**)

  `<MPICH-G2_INSTALL_PATH>/bin/mpif77` (**Fortran77**)

  `<MPICH-G2_INSTALL_PATH>/bin/mpif90` (**Fortran90**)

- Lanciare l'applicazione attraverso il comando `mpirun` di MPICH-G2; ad ogni invio attraverso il *device globus2* corrisponderà una *Resource*

Specification Language (RSL) script diretta ad una griglia computazionale *Globus enabled*: ogni script RSL è composta infatti da uno o più *subjob*, tipicamente uno per ogni macchina (nel nostro caso **interi cluster**) coinvolta nella computazione.

È importante ricordare che la comunicazione tra *subjob* differenti avviene sempre a livello TCP, mentre verrà utilizzato l'MPI locale (se abilitato e disponibile) solo nelle interazioni fra nodi appartenenti allo stesso *subjob*.

Le Script RSL

Per ottenere queste particolari *script*, definite nel linguaggio RSL, è possibile utilizzare lo stesso comando di MPICH-G2 `mpirun`; a tal fine occorre fornire come argomento un *machine file* attraverso la direttiva `-machinefile` (se tale direttiva non viene specificata verrà cercato il file nella directory corrente e poi come `<MPICH-G2_INSTALL_PATH>/bin/machines`).

Questo file è utilizzato per specificare la lista delle macchine sopra le quali si desidera eseguire la propria applicazione; nel nostro caso, conoscendo a priori quante macchine formano ogni singolo cluster ed utilizzando il solo *front-end* per lo *spawn* dei job, risulterà essere:

```
# cat /usr/local/mpichg2/bin/machines
"fe.giza.unipg.it" 8
"gw.grid.unipg.it" 18
"bwcw1.hpc.thch.unipg.it" 16
```

Ciò specifica che ognuno dei tre cluster (che da ora chiameremo semplicemente GIZA, GRID ed HPC) ha a disposizione rispettivamente 8, 18 e 16 processori (in realtà si hanno 17 nodi bi-processore complessivi e 8 nodi mono-processore), per un totale di 42 cpu disponibili.

In questo contesto, se viene invocato il comando `mpirun` come di seguito:

```
# mpirun -dumprsl -np 42 myapp 5000000
```

verrà esattamente generata (ma non eseguita) la script RSL (si noti il carattere '+' iniziale, necessario per specificare una *multirequest*, e gli indici progressivi per i *subjob*) mostrata in figura **3.1**.

```
+
( &(resourceManagerContact="fe.giza.unipg.it")
   (count=8)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (arguments=" 5000000")
   (directory="/home/carlo")
   (executable="/home/carlo/myapp")
)
( &(resourceManagerContact="gw.grid.unipg.it")
   (count=16)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (arguments=" 5000000")
   (directory="/home/carlo")
   (executable="/home/carlo/myapp")
)
( &(resourceManagerContact="bwcw1.hpc.thch.unipg.it")
   (count=16)
   (jobtype=mpi)
   (label="subjob 2")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (arguments=" 5000000")
   (directory="/home/carlo")
   (executable="/home/carlo/myapp")
)
```

Figura 3.1: **Script RSL generata dal comando mpirun di MPICH-G2 per la griglia.**

Se il parametro `np` fosse superiore, altri *subjob* verrebbero aggiunti alla script RSL utilizzando ciclicamente i diversi *jobmanager* a disposizione.

Ovviamente, se è nota a priori la topologia della griglia computazionale e le risorse a disposizione, è possibile scrivere direttamente a mano script RSL simili; tale azione comunque diviene indispensabile quando si debbono

specificare parametri che differiscono per i vari *jobmanager* (come ad esempio il percorso dell'eseguibile piuttosto che il numero o i valori degli argomenti).

Infine, dopo aver creato la script RSL desiderata, il job può essere lanciato con il comando:

```
# mpirun -globusrsl <myscript>.rsl
```

Se non diversamente specificato, sia lo *standard output* che lo *standard error* saranno inviati in *console*; è comunque possibile modificare tale comportamento indicando i relativi file di *output* ed *error* nella script RSL, con `(stdout=<myapp.out>)` e `(stderr=<myapp.err>)`.

Capitolo 4

Sperimentazione dell'ambiente Grid

In questo ultimo capitolo vengono illustrate le varie fasi operative che hanno contraddistinto il presente lavoro di Tesi: l'installazione dei cluster, l'installazione del *Globus Toolkit*, le personalizzazioni dell'ambiente *Globus* per architetture *Beowulf* (GRAM e MDS), MPICH-G2, le *Certification Authority*, esempi ed un vero e proprio *testbed* per valutare risultati e *performance*.

4.1 Installazione dei Cluster

Al fine di ottenere la griglia computazionale già descritta in precedenza, nell'ambito del laboratorio di *High Performance Computing* del Dipartimento di Matematica ed Informatica, sono stati utilizzati tre cluster differenti, di cui due aderenti al modello *Beowulf* (GIZA e HPC), realizzati rispettivamente

presso il C.A.S.I (Centro d'Ateneo per i Servizi Informatici) e presso il Dipartimento di Chimica, ed uno ad un modello ibrido *Mosix-Beowulf* (GRID), realizzato presso il Dipartimento di Chimica stesso.

In questa sezione si illustreranno brevemente le caratteristiche dei due modelli ed i passaggi principali per l'installazione di GIZA (per *Beowulf*) e GRID (per *Mosix-Beowulf*).

4.1.1 Beowulf

Il cluster *Beowulf* [21] è costituito da un insieme di PC collegati in rete basati su sistemi operativi *Unix-like* di pubblico dominio, in particolare **Linux**; tale insieme di PC viene utilizzato come un unico "supercomputer" parallelo: i nodi sono gestiti dal nodo master.

L'idea risale al 1994 quando *Donald Becker* e *Thomas Sterling* (scienziati della NASA), causa le ristrettezze economiche che impedivano loro di acquistare un supercomputer commerciale per l'analisi dei grandi *dataset* delle missioni spaziali, insieme ai loro colleghi concepirono e dimostrarono *Beowulf*.

Il consacramento ufficiale del progetto è arrivato nel 1997, quando nella *"Top 500 Supercomputers List"*, mantenuta dall'Università di *Mannheim* in Germania e tradizionalmente costituita dai "soliti noti" come Cray, NEC, Silicon Graphics e IBM, sono apparse due *"new entry"* entrambe basate sul

modello *Linux-Beowulf* (*Cplant, Sandia National Labs e Avalon, Los Alamos National Labs*).

Attualmente *Beowulf* [22] è forse il software di *clustering* per Linux più conosciuto e documentato.

Piuttosto il termine identifica un insieme di strumenti software che operano su Linux a livello *kernel* e che includono software di *message passing* (MPI [23] e PVM [24]), modifiche per permettere la gestione di diverse interfacce di rete, driver per reti ad alte prestazioni, cambiamenti al *virtual memory manager* ed ai servizi di IPC (*interprocess* communication) distribuito (DIPC), con la gestione di un *"common global process identifier space"* che permette di accedere ad ogni processo su ogni nodo tramite appunto DIPC.

Nella sua versione più recente, *SCYLD Beowulf*, sono state introdotte alcune interessanti integrazioni per l'implementazione di meccanismi semi–automatici di migrazione dei processi e per il *checkpoint/restart* dei job.

In particolare ci si riferisce a **BProc** (*Beowulf distributed PROCess space*), ovvero uno spazio di processo distribuito che permette agli stessi processi, presenti nella *process-table* del sistema *front-end*, di girare fisicamente sugli altri nodi del cluster.

Il cluster GIZA

Il cluster di classe *Beowulf* realizzato al C.A.S.I. conta sulla potenza di calcolo di 8 macchine (**Appendice A**) monoprocessore.

Oltre agli 8 nodi è stata installata una macchina, con funzione di *front-end* per il cluster, per permettere una più semplice amministrazione del sistema ed una gestione della sicurezza tramite *firewall* (mediante il software `iptables`) tale da costiture l'unico punto d'accesso alla rete Internet.

Il sistema operativo utilizzato è stato **Linux RedHat 7.0** con **kernel 2.2.19** (in effetti la *patch BProc* più recente a disposizione era quella per il kernel versione 2.2.13 che, attraverso opportune modifiche, è stata adattata per la versione 2.2.19).
Si analizza nel dettaglio la realizzazione di tutto il sistema.

Dapprima sono stati installati gli 8 nodi del cluster; ciò che risultava indispensabile era realizzare un'installazione veloce, affidabile ed equivalente su tutte le macchine.

Per ottenere questo risultato si è installato il primo nodo molto attentamente; successivamente si è utilizzato un tool di Redhat (*kickstart*) che permette di produrre un file (`ks.cfg`) con tutte le impostazioni di installazione utilizzate; la funzione *kickstart* è stata quindi utilizzata per replicare l'immagine del primo nodo sugli altri nodi senza ulteriori processi o procedu-

re di installazione ulteriori (per installare Redhat tramite un file di *kickstart*, salvato su un disco magnetico, è necessario avviare l'installazione, da CD o floppy, ed al *prompt* lanciare il processo di *setup* attraverso il comando `linux ks=floppy`).

Il passo successivo è stato quello di installare il *front-end* e di configurare i servizi che esso avrebbe dovuto offrire al cluster, in particolare:

- *Network File System* (NFS);
- *Network Information Service* (NIS);
- *Automounting home directory.*

Per poter avere a disposizione su tutti i nodi lo stesso software si è deciso di condividere via NFS la directory `/usr/local` (difatti l'installazione Globus sarà effettuata a partire da `/usr/local/globus`).

Il NIS è un database amministrativo che fornisce un controllo centrale ed una distribuzione automatica dei più importanti file amministrativi; principalmente vengono controllati dal NIS i file di configurazione di rete e gli utenti.

Tramite il NIS si è condivisa la tabella degli host (i file `/etc/hosts` ed `/etc/networks`), i file di gestione degli utenti (`/etc/groups`, `/etc/passwd`, `/etc/shadow`) ed il file `/etc/netgroups` che permette di creare gruppi di

host ai quali possono essere assegnate le diverse autorizzazioni di accesso; ogni modifica effettuata sul *front-end* si ripercuoterà quindi automaticamente su tutto il cluster migliorando così la gestibilità dell'intero sistema.

Per il corretto funzionamento del cluster si è perciò configurato in modo opportuno il file `/etc/hosts.equiv` che stabilisce quali utenti e quali host possano essere considerati fidati.

Utilizzando *netgroup* è stato creato un gruppo di *trusted host* per il cluster, ovvero il gruppo dei nodi (un utente autenticato su una macchina indicata nel *netgroup* potrà collegarsi liberamente su tutti i nodi del cluster, condividendo la propria *home directory* attraverso *automount*; tali informazioni sono definite nel file `/etc/auto.home` ed anch'esse esportate via NIS):

```
# cat /etc/hosts.equiv
+@nodi
```

L'infrastruttura di comunicazione tra i nodi è stata realizzata mediante tecnologia Fast Ethernet e Gigabit Ethernet, con ciascun nodo connesso ad una porta di uno switch per minimizzare le collisioni.

Inoltre, essendo ogni host dotato di doppia interfaccia di rete, è stato implementato il concetto di *channel bonding*, il quale permette di raddoppiare la banda a disposizione utilizzando due schede di rete al posto di una sola.

Sfruttando quindi una particolare proprietà del *kernel* di Linux si ottiene che i pacchetti di una trasmissione dati in rete verranno inviati attraverso

entrambe le interfacce (definite sullo stesso indirizzo IP) alternativamente; il kernel del sistema infatti mette a disposizione un *device* virtuale `/dev/bond0` che viene utilizzato normalmente dalle applicazioni di più alto livello come device di comunicazione, dove ogni pacchetto proveniente da un applicazione della workstation viene analizzato per determinarne il destinatario.

Come in una qualunque trasmissione dati, verrà quindi richiesto l'indirizzo *ethernet* corrispondente all'indirizzo IP della scheda di rete destinataria del pacchetto, tramite un *broadcast* su entrambe le reti (protocollo ARP); in ogni rete un'interfaccia risponderà con il proprio *MAC address* alla scheda di rete che ne ha fatto richiesta.

Il kernel, a questo punto, invierà il pacchetto ad una delle due schede, predestinando il successivo all'altra interfaccia.

Ricompilato ed installato il nuovo kernel (con la *patch* di *Bproc* ed il suddetto modulo per il *channel bonding*), sul front-end e su tutti i nodi, si è configurato il *device* virtuale (`bond0`) con i parametri di rete come se fosse una scheda di rete vera e propria; ciò avviene attraverso il comando `ifconfig` nel modo seguente (nel caso specifico per il nodo 1):

```
# ifconfig bond0 141.250.252.11 netmask 255.255.255.0 \
  broadcast 141.250.252.255
```

Le due interfacce di rete reali sono state poi collegate al *device* virtuale

tramite i comandi:

```
# ifenslave -v bond0 eth0
# ifenslave -v bond0 eth1
```

Questa operazione è stata quindi replicata su tutti i nodi del cluster, compreso il *front-end*, in quanto per poter sfruttare il concetto di *channel bonding* tutti i nodi debbono risultare configurati in tal senso.

Per quanto riguarda il controllo dei processi, si è utilizzato *beoproc*: tale software è composto di un demone e di alcuni comandi di sistema riscritti per sistemi distribuiti; tra questi si citano le versioni distribuite dei comandi `uptime`, `ps`, `top` e `free` insieme al comando `beoload` (i vari comandi di *beoproc*, che hanno come prefisso "beo", acquisiscono informazioni sullo stato e su i processi dei nodi attraverso messaggi UDP tra tutti i *beoprocd* presenti sulla rete del cluster).

Il carico viene controllato tramite *bWatch*, un tool visuale scritto in **Tcl/tk** che sfrutta *rsh* per interrogare i vari nodi; esso permette di controllare alcuni valori molto importanti riguardanti il carico dei nodi quali *uptime*, memoria utilizzata, numero dei processi e numero degli utenti collegati (fig. **4.1**).

bWatch 1.0.3

Host Name	Num Users	Time	1 min Load	5 min Load	15 min Load	Num procs.	Total Mem	Free Mem	Shared Mem	Buffers	Cache	Total Swap	Free Swap
node01	1 user,	12:18	1.52	1.49	1.29	52	505 Mb	393212 Kb	25 Mb	42 Mb	38 Mb	1027 Mb	1027 Mb
node02	0 users	11:09	1.86	1.67	1.39	55	505 Mb	3856 Kb	23 Mb	442 Mb	11 Mb	1027 Mb	1027 Mb
node03	0 users	3:16	1.84	1.70	1.43	51	505 Mb	241068 Kb	21 Mb	213 Mb	14 Mb	1027 Mb	1027 Mb
node04	0 users	4:32	1.60	1.60	1.36	49	505 Mb	424512 Kb	20 Mb	38 Mb	14 Mb	1027 Mb	1027 Mb
node05	0 users	3:12	2.00	1.91	1.55	50	505 Mb	402572 Kb	20 Mb	57 Mb	15 Mb	1027 Mb	1027 Mb
node06	0 users	11:16	1.63	1.50	1.30	50	505 Mb	405400 Kb	20 Mb	57 Mb	14 Mb	1027 Mb	1027 Mb
node07	0 users	3:17	1.52	1.60	1.36	50	505 Mb	405016 Kb	20 Mb	57 Mb	15 Mb	1027 Mb	1027 Mb
node08	0 users	3:19	1.64	1.46	1.27	50	505 Mb	415788 Kb	20 Mb	45 Mb	15 Mb	1027 Mb	1027 Mb

Refresh
Exit

Figura 4.1: **Monitoraggio dei nodi del cluster GIZA tramite *bWatch*.**

4.1.2 Mosix

La prima versione di MOSIX (*Multicomputer OS for UNIX*) è stata sviluppata alla fine degli anni '70 su un PDP/11 con UNIX/BSD dal Prof. *A. Barak* della *Hebrew University* di Gerusalemme in Israele insieme ai suoi collaboratori (da allora è stato riscritto per diverse architetture e la versione per Linux risale al 1998).

Mosix [25] è un sistema software specificatamente sviluppato per implementare capacità di *clustering* nelle diverse architetture.

Il nucleo è costituito da algoritmi di ottimizzazione in tempo reale del carico, dell'uso della memoria e dell'I/O che corrispondono alle variazioni d'uso delle risorse di un cluster.

Ad esempio, di fronte ad una distribuzione del carico sbilanciata, un *disk swapping* eccessivo o una mancanza di memoria in uno dei membri del clu-

ster, Mosix inizia la migrazione dei processi e del loro spazio immagine (di indirizzamento) da un nodo all'altro per equilibrare il carico, per spostare un processo verso un nodo con memoria libera sufficiente o per ridurre il numero delle operazioni di input/output remote.

Mosix funziona in maniera trasparente all'utente ed alle applicazioni; ciò comporta la possibilità di eseguire applicazioni seriali o parallele come in una macchina SMP (multiprocessore): non bisogna infatti preoccuparsi di dove stiano girando i nostri processi o di cosa stiano facendo gli altri job.

Subito dopo la creazione di un processo, Mosix tenta di assegnarlo al nodo migliore disponibile in quel momento e continua poi a monitorarlo (come per tutti gli altri processi) per migrarlo ancora su altri nodi nel caso ve ne sia la necessità.

Le procedure di Mosix sono decentrate: ogni nodo è sia un *supervisor* per i processi che sono stati creati localmente, che un server per i processi che arrivano dagli altri nodi (ciò significa che i nodi possono essere aggiunti o rimossi in qualunque momento dal cluster in un'assenza teorica di disturbi, ottenendo un'altissima scalabilità).

Il principio della trasparenza di posizione, applicato ai *file system* come NFS, si è esteso con Mosix fino alla distribuzione dei processi tra i nodi di un cluster.

Allo scopo di minimizzare l'impatto del meccanismo di migrazione sul-

l'I/O, il *file system* di Mosix (MFS) presenta le caratteristiche di un *File System ad Accesso Diretto* (DFSA); esso estende la possibilità di un processo migrato a compiere operazioni di I/O nel nodo in cui sta girando: ciò riduce la necessità per i processi con alte richieste di *input/output* di comunicare con il nodo di partenza, permettendo loro di migrare più facilmente da un nodo all'altro.

DFSA funziona per ogni *file system* con le seguenti restrizioni:

- Il *file system* deve essere montato su tutti i nodi nella stessa directory (ad esempio `/mfs`);
- Gli *user(id)-group(id)* degli utenti debbono essere gli stessi su tutto il cluster.

Attualmente il file system di Mosix, MFS, risponde ai requisiti del DFSA; è inoltre in corso di sviluppo l'integrazione con GFS (Global File System).

Mosix può scalare da piccole configurazioni a configurazioni con centinaia di nodi con un impatto minimo sulle prestazioni mentre i nodi possono essere sia normali PC che macchine multiprocessore collegate sia attraverso reti locali standard che attraverso dispositivi ad alte prestazioni; inoltre non è necessario che i nodi siano omogenei dal punto di vista della configurazione dell'hardware.

L'attuale seria limitazione di Mosix [26] risiede nell'impossibilità di mi-

grare processi che richiedano interazione con il *network layer* del sistema, ovvero non è possibile migrare da un host ad un altro i *socket* di comunicazione aperti da un processo.

Il cluster GRID e le sue prestazioni

Il cluster GRID è un ottimo esempio di architettura mista *Mosix–Beowulf*, nel quale convivono diversi aspetti che caratterizzano le due architetture.

GRID consta di un totale di 9 macchine (**Appendice A**), tra le quali una adibita ad assolvere le funzioni di *front-end* (fig. **4.2**).

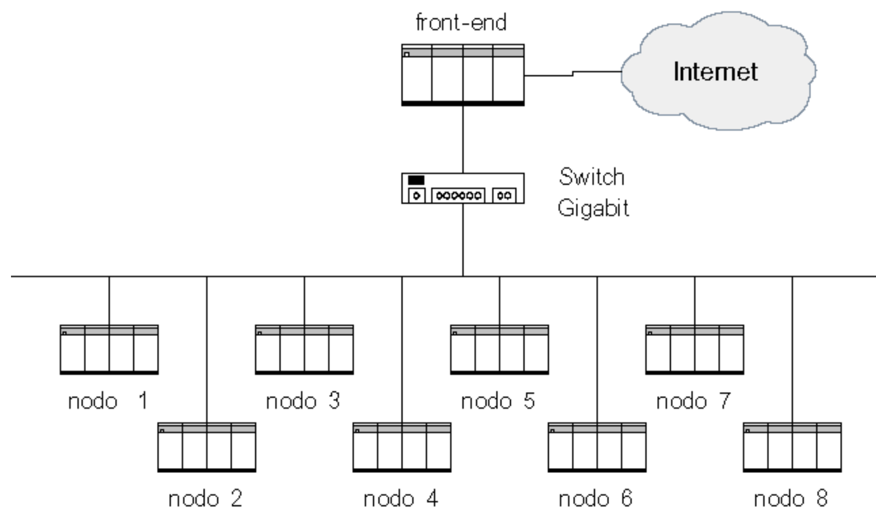

Figura 4.2: **Topologia del cluster GRID.**

Il primo passo da compiere è la ricompilazione del kernel; una volta sistemati i sorgenti del kernel nella directory standard (`/usr/src/linux`) è suffi-

ciente eseguire lo script `mosix.install`, contenuto nel pacchetto *Mosix*, per applicare le modifiche necessarie e lanciare automaticamente il programma di configurazione dei parametri del nuovo *kernel*.

Completata la configurazione, viene automaticamente avviata la compilazione e l'installazione del kernel; al termine di questa fase, viene richiesto di specificare un *mount point* per il *Mosix File System*, di solito la directory `/mfs`.

Il *Mosix File System* è un file system speciale che consente di accedere ai file system reali di ciascun nodo di un cluster *Mosix* tramite la directory `/mfs` la quale dovrà essere presente su tutti i nodi; `/mfs` non ha bisogno di un'effettiva partizione su disco (similmente al *file system* `/proc`), in quanto è un'astrazione realizzata a livello software.

Durante la fase di inizializzazione dei servizi verrà avviato lo script di avvio di Mosix `/etc/rc.d/init.d/mosix`, il quale chiederà all'utente di effettuare la compilazione del file `/etc/mosix.map`, che definisce la mappa degli host partecipanti al cluster, e la selezione dell'identificativo univoco del nodo in questione, che verrà salvato nel file `/etc/mospe`.

La sintassi del file `mosix.map` è piuttosto semplice: consiste in una serie di linee di tre campi, separati da spazi o da tabulazioni, ciascuna indicante un intervallo di indirizzi IP.

I tre campi sono, rispettivamente:

- L'**identificativo univoco** Mosix del primo nodo nell'intervallo;
- L'**indirizzo IP** del primo nodo nell'intervallo;
- Il **numero di nodi** nell'intervallo.

Nel nostro caso, il file di configurazione `/etc/mosix.map` del cluster risulta essere quello mostrato in figura **4.3**.

```
# MOSIX CONFIGURATION
# ===================
# MOSIX-#  IP             number-of-nodes
# ========================================

1          141.250.240.3   1
2          141.250.240.11  8
```

Figura 4.3: **mosix.map, file di configurazione principale di MOSIX (cluster GRID).**

Il corretto funzionamento dell'ambiente *Mosix* può essere verificato controllando che nella directory `/proc/mosix/nodes` compaiano gli identificativi di tutti i nodi.

Essenzialmente, le applicazioni parallele si dividono in due grandi gruppi: quelle di tipo *CPU–intensive* e quelle di tipo *I/O–intensive*.

Le applicazioni *CPU–intensive* sono quelle in cui il tempo impegato per eseguire i calcoli è maggiore rispetto a quello per lo svolgimento delle opera-

zioni di input/output mentre quelle *I/O–intensive* sono quelle in cui la maggior parte del tempo è spesa per effettuare operazioni di lettura/scrittura e di trasmissione di dati.

Risulta evidente come la prima tipologia sarà influenzata principalmente dalla potenza di calcolo dei singoli nodi, mentre la seconda sarà influenzata da vari fattori, tra cui i più importanti sono il *throughput*, la latenza delle interfacce di rete e la velocità di accesso ai dispositivi di memorizzazione di massa.

Quando si effettua il *benchmarking* di un cluster [27] si predilige osservare il comportamento del sistema durante l'esecuzione di applicazioni di tipo *CPU–intensive*, spingendole fino al limite che le separa da quelle di tipo *I/O–intensive*.

Per il calcolo delle prestazioni del cluster realizzato (GRID) si è scelto di utilizzare il pacchetto software HPL [28] (*High–Performance Linpack benchmark*), scritto in C con l'ausilio delle librerie MPI.

HPL è basato su un algoritmo che effettua il calcolo del vettore delle soluzioni di un sistema lineare di tipo $\mathbf{Ax} = \mathbf{b}$ tramite la fattorizzazione LU della matrice dei coefficienti, ripartendo il carico di lavoro in una griglia di P x Q processori.

La scelta della dimensione N del problema da calcolare (cioè della matrice dei coefficienti $\mathbf{A}$) e dei valori di P e Q, il cui prodotto determina il numero

di processori da utilizzare, sono parametri critici strettamente dipendenti dal tipo di architettura utilizzata.

Per il calcolo dell'indice delle prestazioni, espresso in *GFlop/s* (miliardi di istruzioni in virgola mobile per secondo), si tiene conto del fatto che l'algoritmo esegue, per completare la soluzione del sistema lineare, $2N^3/3+2N^2$ operazioni in virgola mobile; quindi, disponendo al termine dell'esecuzione del tempo impiegato per la risoluzione del sistema lineare, è possibile calcolare l'indice delle prestazioni utilizzando la seguente formula:

$$GFlop/s = \frac{2N^3/3 + 2N^2}{T_s}$$

dove T_s indica il tempo impiegato dall'algoritmo per trovare le soluzioni del sistema $\mathbf{Ax} = \mathbf{b}$.

Nel nostro caso si è scelto di utilizzare una griglia di 4x4 processori, in quanto si hanno a disposizione, escluso il *front–end*, 8 macchine dotate di 2 CPU ciascuna.

Inoltre, come grandezza massima del problema da calcolare si è scelto il numero 29000, poiché si è potuto osservare, a seguito di numerose prove sperimentali, che al tendere di N a questo numero l'indice delle prestazioni tende a stabilizzarsi intorno al valore di 7.2 **GFlop/s**.

La tabella 4.1 riporta gli indici misurati in *GFlop/s* ed i tempi di esecu-

N	GFlop/s	Ts
5000	4.190	19.90 s
8000	5.286	64.59 s
10000	5.479	121.71 s
15000	6.318	356.19 s
20000	6.773	787.49 s
25000	7.071	1473.30 s
27000	7.114	1844.78 s
29000	7.240	2245.83 s

Tabella 4.1: **Prestazioni del Cluster GRID per problemi di dimensione N variabile.**

zione T_s, espressi in secondi, relativi a 8 problemi caratterizzati da grandezze diverse.

Per valori di N superiori a 29000 le prestazioni tendono a degradare; questo problema è dovuto al fatto che, per problemi di questa entità, le operazioni di *swap–out* per liberare la memoria fisica e di trasmissione dei blocchi di lavoro tra i nodi prendono il sopravvento su quelle di calcolo vere e proprie, facendo rientrare l'applicazione di *benchmarking* nella tipologia *I/O–intensive.*

4.2 Installazione di Globus Toolkit

Esistono due diverse possibilità per compiere l'installazione di **Globus Toolkit**, ovvero a partire dalla distribuzione dei sorgenti oppure da una distribuzione dei binari già compilati; le architetture correntemente supportate sono Linux/x86, Solaris/SPARC, IRIX/MIPS e AIX5.1/RS6000.

In ogni caso, il primo *tool* da utilizzare è il *Grid Packaging Toolkit* (GPT), utilizzato per installare Globus in entrambe le modalità.

Dopo aver ottenuto e decompresso il relativo software, definito la variabile d'ambiente `GLOBUS_LOCATION` relativa alla locazione del *Globus Toolkit*, si provvederà all'installazione del GPT in questo modo (tipicamente da un utente diverso da `root`, ad esempio l'utente `globus`):

```
# export GLOBUS_LOCATION=/usr/local/globus
# cd gpt-1.0
# ./build_gpt |& tee gpt.log
```

Nel nostro caso si è optato per un'installazione a partire dai sorgenti; infatti in questo modo è possibile ottenere un software specifico (più performante) per la nostra particolare architettura e soprattutto completo di ogni libreria o applicativo.

Per ogni *bundle* che si desidera installare è possibile scegliere varie opzioni e particolari *flavor*; essi abilitano specifiche funzionalità come il supporto al

debug, ai *POSIX–thread* o ai diversi compilatori; dopo l'installazione di ogni *bundle* è necessario lanciare la script:

```
# $GLOBUS_LOCATION/setup/globus-postinstall.sh
```

Dopo aver terminato l'installazione, è opportuno inserire nell'ambiente del sistema (tipicamente in `bashrc` o nei relativi file `.bashrc` per ogni utente) le seguenti informazioni:

```
# export GLOBUS_LOCATION=/usr/local/globus
# . $GLOBUS_LOCATION/etc/globus-user-env.sh
# export PATH=$PATH:$GLOBUS_LOCATION
```

In seguito è necessario ottenere almeno tre certificati: uno per un utente di *test*, uno per l'host che dovrà assolvere le funzioni di *Gatekeeper* ed uno per *ldap* (se lo si vuole attivare).

Infine, verranno definiti i nuovi servizi di rete disponibili; in particolare si aggiungerà, al file `/etc/services`, l'*entry* seguente:

```
gsigatekeeper      2119/tcp              # Globus Gatekeeper
```

Se si desidera attivare il *Gatekeeper* mediante il demone **inetd**, è possibile installare il nuovo servizio aggiungendo la riga seguente in `/etc/inetd.conf`:

```
gsigatekeeper stream tcp nowait root                     \
GLOBUS_LOCATION/sbin/globus-gatekeeper globus-gatekeeper \
-conf GLOBUS_LOCATION/etc/globus-gatekeeper.conf
```

dove si sostituirà a `GLOBUS_LOCATION` il valore di `$GLOBUS_LOCATION` reale.

Se si desidera invece utilizzare il demone **xinetd**, si aggiungerà un file chiamato *globus-gatekeeper* nella directory `/etc/xinetd.d/`, riportato in figura **4.4**.

```
service gsigatekeeper
{
socket_type = stream
protocol    = tcp
wait        = no
user        = root
server      = GLOBUS_LOCATION/sbin/globus-gatekeeper
server_args = -conf GLOBUS_LOCATION/etc/globus-gatekeeper.conf
disable     = no
}
```

Figura 4.4: **globus-gatekeeper, file di configurazione del *Globus Gatekeeper* per xinetd.**

La stessa cosa può essere effettuata, per il software relativo al *Data Management Pillar*, installando il *GridFTP server* (wu-ftpd) ed aggiungendo l'*entry* relativa al GridFTP:

```
gsiftp 2811/tcp
```

nel file `/etc/services`.

Attivando GridFTP mediante il demone **inetd** va aggiunta la riga seguente nel file `/etc/inetd.conf`:

```
gsiftp stream tcp nowait root \
GLOBUS_LOCATION/sbin/in.ftpd in.ftpd -l -a
```

Se si preferisce invece attivarlo mediante **xinetd** si aggiungerà, nella directory `/etc/xinetd.d/`, il file mostrato in figura **4.5**.

```
service gsiftp
{
instances       = 1000
socket_type     = stream
wait            = no
user            = root
server          = GLOBUS_LOCATION/sbin/in.ftpd
server_args     = -l -a -G GLOBUS_LOCATION
log_on_success += DURATION USERID
log_on_failure += USERID
nice            = 10
disable         = no
}
```

Figura 4.5: **globus-ftp, file di configurazione di *GridFTP* per xinetd.**

In questo lavoro, si ha un'installazione Globus completa nel cluster GRID, mentre negli altri due, HPC e GIZA, è presente solo il *pillar* relativo al *Resource Management*, GRAM; su tutti e tre i cluster è poi installato un ambiente MPICH-G2 completo e tre relative *Certification Authority* reciprocamente fidate (fig. **4.6**).

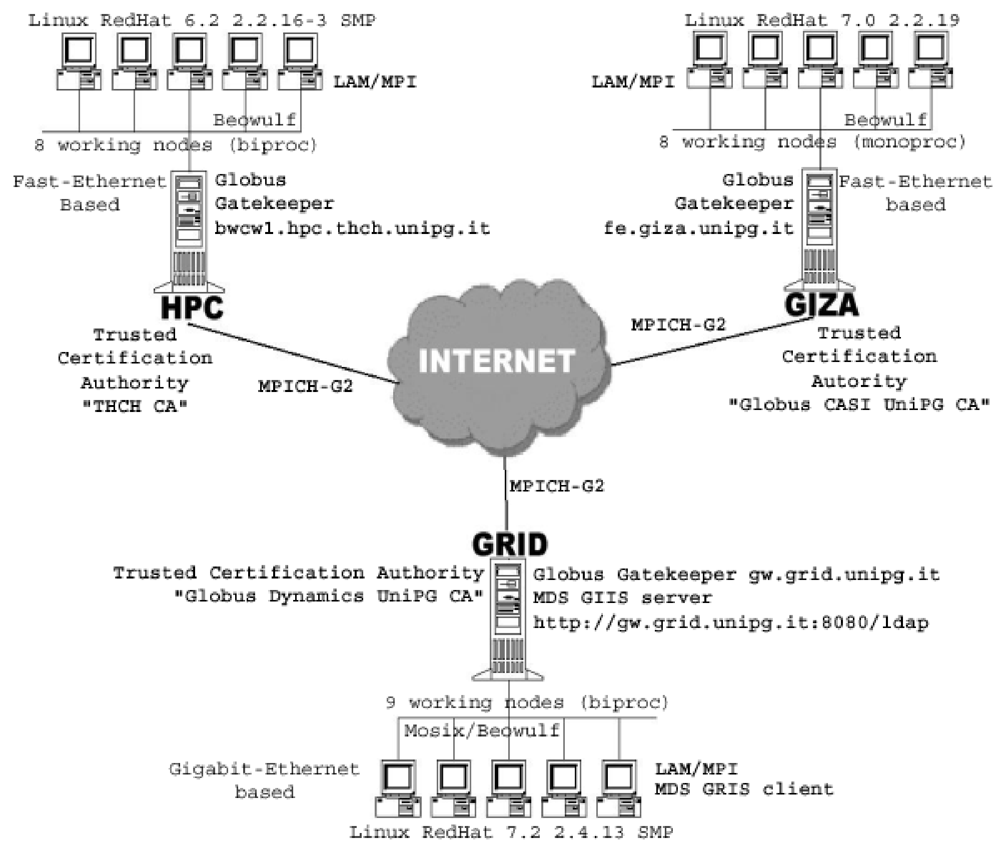

Figura 4.6: **L'architettura realizzata.**

4.2.1 GRAM e MPICH-G2 su un'architettura *Beowulf*

Si è quindi deciso di installare il *Resource Management Pillar* (GRAM) sul *front-end* di ogni cluster; in particolare i pacchetti software *Client*, *Server* e *SDK*.

Con i seguenti comandi viene installato il software relativo al *Client bundle* di GRAM:

```
# $GLOBUS_LOCATION/sbin/globus-build -install-only  \
globus_resource_management_bundle-client-src.tar.gz \
gcc32 -verbose | & tee build_gram_client.log
```

Con i seguenti comandi viene invece installato il software relativo al *Server bundle* di GRAM:

```
# $GLOBUS_LOCATION/sbin/globus-build -install-only  \
globus_resource_management_bundle-server-src.tar.gz \
-static=1 gcc32 -verbose | & tee build_gram_server.log
```

Con i seguenti comandi viene poi installato il software relativo all'*SDK bundle* di GRAM:

```
# $GLOBUS_LOCATION/sbin/globus-build -install-only  \
globus_resource_management_bundle-sdk-src.tar.gz    \
gcc32 -verbose | & tee build_gram_sdk.log
```

A questo punto viene effettuata la definizione dell'ambiente, per ciò che concerne la sicurezza, tramite l'esecuzione della script `setup-gsi`.

In particolare tale comando (da eseguire come `root`) effettuerà le seguenti azioni:

- Creerà la directory `/etc/grid-security` popolandola con i file di configurazione necessari;
- Installerà il certificato della *Certification Authority* e la sua relativa *policy* nella directory `/etc/grid-security/certificates`.

Per generare la richiesta del certificato per il *Gatekeeper* (memorizzata nel file `hostcert.req`) si utilizzerà il comando:

```
# grid-cert-request -gatekeeper ‘globus-hostname‘  \
    -key     $GLOBUS_LOCATION/etc/hostkey.pem      \
    -cert    $GLOBUS_LOCATION/etc/hostcert.pem     \
    -request $GLOBUS_LOCATION/etc/hostcert.req
```

Il file `hostcert.req` va quindi inviato alla CA di competenza (nel nostro caso la CA locale, come vedremo più avanti) con un comando del tipo:

`# cat $GLOBUS_LOCATION/etc/hostcert.req |mail` *ca@grid.unipg.it*

Una volta ottenuto il certificato si provvederà a copiarlo nel file `hostcert.pem`, fino ad allora vuoto, ed a copiare i file `hostkey.pem` ed `hostcert.pem` nella directory `/etc/grid-security`.

Dopo l'installazione del certificato, la script di post-installazione (da avviare dopo la procedura di `setup-gsi`) produrrà un *output* che riporta la creazione del *Gatekeeper* e del suo *jobmanager* di default, come evidenziato

in figura **4.7**.

```
Setting up job manager scheduler scripts...
  Done
  Setting up fork job manager
  ------------------------------
   Creating job manager configuration file...
     - Getting gatekeeper subject
     - Getting gatekeeper port
  Done
  Creating grid service jobmanager...
  Done
```

Figura 4.7: **Creazione del *Gatekeeper* e del jobmanager (default) di Globus.**

Dopo aver installato GRAM, è possibile installare anche gli altri *pillar* di *Globus* secondo una procedura pressochè analoga; l'unico accorgimento da adottare è quello di installare entrambi i *Client bundle* e *Server bundle* dell'*Information Services Pillar* con il flavor `gcc32pthr` (POSIX *thread* compatibile) invece che con il flavor `gcc32`; un altro aspetto da notare è che nell'installazione del pacchetto *Server* di MDS non compare l'opzione `-static=1` (che è invece sempre presente sia nel pacchetto *Server* di GRAM che in quello del *Data Management Pillar*) [29].

Dopo l'installazione di MDS (cluster GRID) si provvederà alla generazione della richiesta per il relativo certificato:

```
# grid-cert-request -cn ldap/gw.grid.unipg.it         \
```

```
-cert $GLOBUS_LOCATION/etc/server.cert          \
-key  $GLOBUS_LOCATION/etc/server.key           \
-req  $GLOBUS_LOCATION/etc/server.request -nopw  \
-dir  $GLOBUS_LOCATION/etc
```

all'invio della richiesta (file `server.request`) alla CA ed all'installazione del certificato ricevuto copiandolo in `$GLOBUS_LOCATION/etc/server.cert`.

A questo punto si sono realizzati tre *Globus Gatekeeper* funzionanti; ma come utilizzare anche le altre macchine del cluster e farle lavorare insieme ad esempio attraverso MPI?

Esitono almeno tre approcci molto differenti che permettono di affrontare questa problematica:

- Utilizzare un *Gatekeeper* per ogni macchina;
- Utilizzare uno scheduler locale delle risorse (a livello di singolo cluster);
- Utilizzare un ambiente MPI locale (ad esempio LAM [30], *Local Area Multicomputer*) da "collegare" con MPICH-G2.

La prima ipotesi è stata scartata poichè risulta impensabile avere un *Gatekeeper* per ogni macchina (si pensi ad un ambiente con centinaia di nodi) ovvero un'installazione Globus per ogni nodo; inoltre, per ciò che concerne MPI, in questo modo non si riuscirebbe a sfruttare il livello *MPI* tra le macchine in quanto le comunicazioni avverrebbero sempre via TCP (perdendo ad esempio la capacità di sfruttare macchine SMP).

Nella seconda ipotesi, si prevede che ogni cluster abbia uno *scheduler* locale deputato alla gestione dei job ad esso inviati; si era ad esempio pensato a due *jobmanager* definiti in Globus: il primo per l'esecuzione di job "seriali" sul *Gatekeeper* mentre il secondo si sarebbe preoccupato di trasferire a *Condor* (il nostro *scheduler* locale) i job paralleli da far girare sul proprio *pool*.

Tale ipotesi, forse la più valida dal punto di vista teorico, è stata scartata poichè, utilizzando *Condor* come *scheduler* locale, non è ancora stato implementato, ad oggi, il relativo "ponte" *Globus–Condor*; in figura **4.8** si riporta la sezione di codice relativo a questa problematica.

```
# cat /usr/local/globus/libexec/globus-script-condor-submit
...
# 4 jobtypes exist                condor jobtype
# -----------------               -------------------
# jobtype 0 = mpi         ----->  ERROR
# jobtype 1 = single      ----->  set universe to vanilla
# jobtype 2 = multiple    ----->  set universe to vanilla
# jobtype 3 = condor      ----->  set universe to standard
$DEBUG_ECHO "JM_SCRIPT: testing jobtype" >> $grami_logfile
if [ $grami_job_type = "0" ] ; then
   $DEBUG_ECHO "JM_SCRIPT: error: jobtype MPI not supported "
   >> $grami_logfile
   echo "GRAM_SCRIPT_ERROR:$GLOBUS_GRAM_PROTOCOL_ERROR_JOBTYPE_
   NOT_SUPPORTED"
   exit 1
...
```

Figura 4.8: **Codice sorgente di Globus che determina l'attuale mancato supporto di Condor come *scheduler* MPI.**

L'ultima ipotesi, quella adottata, prevede una importante modifica del codice sorgente per far cooperare insieme LAM/MPI e MPICH-G2; infatti, sebbene la documentazione parli di una compatibilità completa fra MPICH-G2 e qualsiasi altra implementazione MPI non *MPICH–based*, si conviene che tale supporto era stato pensato solamente per architetture MPI proprietarie e non per versioni *Open-Source* di MPI come LAM/MPI.

Il primo passo da compiere è quello di ricompilare l'*SDK bundle* relativo al *Resource Management* con il *flavor* **mpi** (ciò serve appunto per far utilizzare a MPICH-G2 l'MPI locale LAM/MPI):

```
# $GLOBUS_LOCATION/sbin/globus-build -install-only  \
globus_resource_management_bundle-sdk-src.tar.gz    \
gcc32mpi -verbose                                   \
| & tee build_gram_sdk.log
```

In questo caso si verificherà un errore nel *package* `globus_core` come mostrato in figura **4.9**.

```
...
checking host system type... i686-pc-linux-gnu
checking for MPI... no
configure: error: This system does not support MPI
ERROR: Build has failed
```

Figura 4.9: **Errore riportato dalla compilazione standard del pacchetto *sdk* di GRAM, con il *flavor mpi*.**

Per ovviare a ciò, si deve ricompilare manualmente solo questo pacchetto (di modo che, rilanciando l'installazione dell'SDK, tale pacchetto venga skippato perchè già compilato) con la serie di comandi mostrati in figura **4.10**.

```
# export GLOBUS_CC=gcc;
# export LDFLAGS='-L/usr/local/globus/lib'
# /usr/local/globus/BUILD/globus_core-2.1/configure
  --with-flavor=gcc32mpi --enable-debug --with-mpi
  --with-mpi-includes=-I/usr/include/
  --with-mpi-libs="-L/usr/lib -lmpi -llam"
# make all
# make install
```

Figura 4.10: **Ricompilazione manuale del componente *globus_core* con *flavor mpi* e LAM/MPI locale.**

A questo punto, prima di procedere con la compilazione di MPICH-G2, è necessario aggiungere alcune `define` al file di *include* `/usr/include/mpi.h` (quello utilizzato da LAM) per Globus; i tipi di dato necessari sono riportati in figura **4.11**.

Ora è finalmente possibile compilare MPICH-G2 attraverso i seguenti comandi:

```
# ./configure -prefix=/usr/local/mpichg2 \
  -device=globus2:-flavor=gcc32mpi
# make; make install
```

Si noti come la compilazione di MPICH-G2 in realtà non sia altro che la compilazione di MPICH con il device specifico **globus2** (ad esempio invece

```
#define MPI_CHARACTER          ((MPI_Datatype) &lam_mpi_character)
#define MPI_COMPLEX            ((MPI_Datatype) &lam_mpi_cplex)
#define MPI_DOUBLE_COMPLEX     ((MPI_Datatype) &lam_mpi_dblcplex)
#define MPI_LOGICAL            ((MPI_Datatype) &lam_mpi_logic)
#define MPI_REAL               ((MPI_Datatype) &lam_mpi_real)
#define MPI_DOUBLE_PRECISION   ((MPI_Datatype) &lam_mpi_dblprec)
#define MPI_INTEGER            ((MPI_Datatype) &lam_mpi_integer)
#define MPI_2INTEGER           ((MPI_Datatype) &lam_mpi_2integer)
#define MPI_2REAL              ((MPI_Datatype) &lam_mpi_2real)
#define MPI_2DOUBLE_PRECISION ((MPI_Datatype) &lam_mpi_2dblprec)
```

Figura 4.11: **Tipi di dato Fortran necessari per l'esecuzione di programmi MPI (Fortran) a fronte di trasferimenti tra macchine eterogenee.**

del classico *ch_p4*); un'ulteriore operazione manuale prevede di copiare alcuni simboli dai sorgenti di *MPICH* agli *include file* di MPICH-G2 in questo modo:

```
# cp mpich-1.2.4/mpid/globus2/global_c_symb.h mpichg2/include/
```

A questo punto è necessario comunicare a Globus che dovrà utilizzare LAM localmente; si è modificata la macro `GLOBUS_GRAM_JOB_MANAGER_MPIRUN` (in `$GLOBUS_LOCATION/libexec/globus-gram-job-manager-tools.sh`) con il valore `/usr/local/globus/bin/mpigrun` che va a sostituire il comando di default `/usr/bin/mpirun`.

`mpigrun` è di fatto una *script shell* il cui codice sorgente viene mostrato in figura **4.12**.

Tale script prima di tutto avvia l'ambiente LAM secondo un file, chiamato `hosts`, che specifica il nome dei nodi ed il numero di processori per nodo (è

```
export LAMRSH=rsh
export LAM_MPI_SOCKET_SUFFIX="GJOB"$$
/usr/bin/lamboot /usr/local/globus/hosts >> /dev/null 2>&1
/usr/bin/mpirun -c2c -O -x `/usr/local/globus/bin/glob_env` $*
rc=$?
/usr/bin/lamhalt >> /dev/null 2>&1
exit $rc
```

Figura 4.12: **mpigrun, script per lo *'spawn'* dei job MPICH-G2 sulle risorse locali (cluster) attraverso LAM/MPI.**

a questo punto che viene sfruttato l'SMP), del cluster; tale file è riportato in figura **4.13** (relativamente al cluster GRID).

In seguito, la script esegue il particolare job (con i suoi argomenti), identificato univocamente attraverso l'uso del costrutto `$$` della shell, in un ambiente definito da un programma chiamato `glob_env` che recupera le variabili d'ambiente correnti (**Appendice B**); il job MPI viene quindi eseguito sui nodi specificati, in parallelo con gli altri cluster che svolgono il medesimo lavoro.

```
gw  cpu=2
n01 cpu=2
n02 cpu=2
n03 cpu=2
n04 cpu=2
n05 cpu=2
n06 cpu=2
n07 cpu=2
n08 cpu=2
```

Figura 4.13: **hosts, file di configurazione dell'ambiente LAM/MPI locale (cluster GRID).**

Affinchè tutto funzioni correttamente è necessario che i singoli nodi sappiano dove reperire le librerie di Globus; a tal fine per ogni nodo sarà necessario impartire i seguenti comandi:

```
# echo "/usr/local/globus/lib" >> /etc/ld.so.conf
# ldconfig
```

In questo modo la directory `/usr/local/globus/lib` verrà aggiunta al percorso di ricerca delle librerie di sistema.

L'*overhead* derivante da questa soluzione è rappresentato dal fatto che ogni job MPI di ogni utente darà luogo ad un diverso ambiente LAM (di fatto se avessimo 3 utenti che hanno lanciato sopra la griglia 4 job mpi ciascuno avremmo 12 `lamboot` in esecuzione).

Se si volesse che i vari job vengano eseguiti solo sugli 8 nodi del cluster (e non sul *front-end*), si dovrà modificare la script "`mpigrun`" con:

```
# /usr/bin/mpirun -c2c -O n1 n2 n3 n4 n5 n6 n7 n8
  -x '/usr/local/globus/bin/glob_env' $*
```

dove `n[1..8]` indicano gli indici degli 8 nodi.

A questo punto non rimane altro che compilare opportunamente i nostri job MPI; un `Makefile` tipico può essere quello mostrato in figura **4.14**. Infine si creerà una script RSL per la griglia (similmente a quelle illustrate in precedenza) ed il job verrà lanciato con il comando `globusrun`:

```
# globusrun -w -f script.rsl
```

```
# assumes MPICH-G2 was installed in /usr/local/mpichg2
#
MPICH_INSTALL_PATH = /usr/local/mpichg2
CC          = $(MPICH_INSTALL_PATH)/bin/mpicc
CCLINKER    = $(MPICH_INSTALL_PATH)/bin/mpiCC
OBJ = mine.o
mine: $(OBJ)
        $(CC) -o $(@) $(OBJ)
clean:
        /bin/rm -rf *.o mine core
```

Figura 4.14: ***Makefile* d'esempio per la compilazione di programmi MPI con MPICH-G2.**

4.2.2 Le *Certification Authority* e gli utenti Globus

Un passo successivo nella costruzione del nostro ambiente è stato quello di utilizzare delle *Certification Authority* locali invece che la CA gestita dal progetto Globus, al fine di ottenere, da parte nostra, il pieno e completo controllo degli utenti che accederanno all'ambiente Grid.

Per ottenere ciò è necessario, per ogni cluster, installare il relativo *package* `globus_simple_ca_bundle-beta-r2` in questo modo:

```
# $GLOBUS_LOCATION/sbin/globus-build -install-only \
  globus_simple_ca_bundle-beta-r2.tar.gz gcc32
```

Eseguito il relativo `globus-postinstall.sh`, si ha la possibilità di impostare un *Subject Name* locale, una password per le operazioni della CA ed un indirizzo E-Mail (ad esempio *ca@grid.unipg.it*) al quale spedire le richieste per i vari certificati.

In corrispondenza al *Subject Name* avremo quindi tre diverse tipologie di certificati X.509, una per ogni cluster (chiamato `access_id_CA`), in particolare:

- GRID:

  '/O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/

  CN=Globus Dynamics UniPG CA'

- GIZA:

  '/O=Grid/OU=Globus-Unipg/OU=simpleCA-fe.giza.unipg.it/

  CN=Globus CASI UniPG CA'

- HPC:

  '/O=Grid/OU=THCH/OU=CA-thch.unipg.it/CN=THCH CA'

Terminata la definizione dei parametri richiesti e la compilazione, è necessario rieseguire la script di `setup-gsi` per installare la nuova CA ottenuta (di default creata in `$CA_HOME/.globus/simpleCA`, dove `$CA_HOME` corrisponde all'*home directory* dell'utente con il quale è stata lanciata la procedura di installazione).

Tale procedura crea automaticamente anche un pacchetto nella forma `globus_simple_ca_<HASH_CA_ID>_setup-0.5.tar.gz` che conterrà tutte le specifiche di questa CA; basterà quindi installare e distribuire questo software sugli altri cluster per avere una mutua fiducia rispetto ai certificati

generati (un utente accreditato su un cluster lo sarà anche nell'intera griglia computazionale).

È infine possibile installare (insieme al GPT ed ai *bundle Client* e *Server* di GRAM) questa CA in una macchina UNIX qualsiasi (o anche in un altra *Grid*) per avere una postazione dalla quale lanciare job sulla griglia ottenuta.

Come si è accennato, gli utenti registrati presso una qualsiasi CA possono autenticarsi sull'intera griglia; ma come fanno gli utenti ordinari a diventare dei cosidetti *"Globus User"*?
Sostanzialmente devono essere eseguiti tre passaggi fondamentali:

- Definizione dell'ambiente operativo;
- Richiesta ed installazione di un certificato valido;
- Richiesta di abilitazione alla griglia secondo il proprio *Globus_ID*.

Il primo passo (già visto in precedenza) consiste nell'impostare variabili d'ambiente opportune relative alla particolare installazione Globus, ad esempio `GLOBUS_LOCATION=/usr/local/globus`.

Il secondo passo prevede l'utilizzo del comando `grid-cert-request` in questo modo:

```
# grid-cert-request -cn "Carlo Manuali"
```

In seguito a tale comando diverse operazioni verranno eseguite, in particolare:

- Verrà creata la directory $HOME/.globus;
- Verrà creata la chiave privata, protetta da password, in $HOME/.globus/userkey.pem;
- Verrà generata una richiesta alla CA di default (a questo punto quella locale) memorizzata in $HOME/.globus/usercert_request.pem.

Tale richiesta (contenente il *Subject* completo del certificato, ad esempio /O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/OU=grid.unipg.it/CN=Carlo Manuali) verrà, come al solito, spedita alla CA di competenza, la quale provvederà alla creazione del certificato ed alla spedizione di quest'ultimo all'utente legittimo.

Una volta ricevuta la richiesta del certificato per posta elettronica, salvato il file relativo in una locazione specifica (<req>), verrà generato il relativo certificato con il comando:

```
# $GLOBUS_LOCATION/bin/grid-ca-sign -in <req> -out <cert>
```

In seguito la CA dovrà provvedere ad inviare il certificato appena generato (di default creato in $CA_HOME/.globus/simpleCA/newcerts) all'utente, il quale lo copierà in $HOME/.globus/usercert.pem, fino ad allora vuoto, e si preoccuperà che ad ogni entità di griglia (nel nostro caso ogni cluster) venga aggiunta, nel file d'accesso /etc/grid-security/grid-mapfile, l'*entry* relativa, del tipo:

```
"/O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/OU=grid.
unipg.it/CN=Carlo Manuali" carlo
```

In questo caso si presuppone che l'utente `carlo` sia presente sul sistema locale; se così non fosse, è importante ricordare che un utente "in entrata" può sempre essere ridefinito come un qualsiasi utente locale, ad esempio:

```
"/O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/OU=grid.
unipg.it/CN=Carlo Manuali" chiara
```

In questo modo l'utente `carlo`, proprietario del certificato, esegue i propri job su un cluster come se fosse l'utente `chiara`.

L'utente inoltre può, in ogni momento, richiedere un nuovo certificato o recuperare le informazioni relative al suo certificato corrente con il comando:

```
# grid-cert-info -subject -f /home/carlo/.globus/usercert.pem
/O=Grid/OU=Globus-Unipg/OU=simpleCA-gw.grid.unipg.it/OU=grid.
unipg.it/CN=Carlo Manuali
```

A questo punto non resta altro che effettuare il *sign* su Globus come mostrato nella figura **2.13** del capitolo 2.2 (inizializzazione del *proxy* attraverso il comando `grid-proxy-init`).

È comunque possibile cambiare l'intervallo di validità (di default 12 ore) del *proxy* attraverso l'opzione `-hours <hours_number>` del comando stesso.

Per verificare che tutto stia funzionando correttamente è possibile effettuare un *test* di connessione con:

```
# globusrun -a -r gw.grid.unipg.it
GRAM Authentication test successful
```

Tale operazione, ripetuta per ogni *Gatekeeper* del cluster restituisce la reale connettività con le risorse della griglia (ovviamente il *test* dovrà andare a buon fine per ogni cluster).

Una volta attivato il *proxy* è possibile cominciare a lanciare job, ad esempio:

```
# globusrun -o -r gw.grid.unipg.it '&(executable=/bin/date)'
Tue Sep 17 15:31:38 CEST 2002
```

In questo caso (tipico *job* di *test*) verrà restituita la data remota del cluster GRID.

Tutto ciò risulta sufficiente per raggiungere lo scopo prefissato: far eseguire job paralleli di tipo MPI su una griglia computazionale composta da diversi cluster eterogenei distribuiti sopra Internet.

4.2.3 MDS su un'architettura *Beowulf*

In un cluster con architettura *Beowulf*, di solito i nodi condividono un *file system* comune, ad esempio tramite NFS, mentre gli utenti possiedono le stesse credenziali (come *login* e *password*) per lavorare sopra l'intero sistema (NIS).

Da ciò ne deriva che installando il *Globus Toolkit* su una locazione condivisa (ad esempio `/usr/local/globus` dove la directory `/usr/local` viene esportata) avremo accesso a tutta la `$GLOBUS_LOCATION` da ogni singola macchina.

Come si è già visto nel secondo capitolo, *Monitoring and Discovery Service* (MDS) crea ed utilizza diversi file di configurazione locati nella directory `/usr/local/globus/etc`; essi sono quindi valutati al momento dell'avvio di MDS attraverso la script `SXXgris`.

L'obiettivo è quello di avere un GRIS per ogni nodo ed un unico GIIS sul *front-end* del cluster; ciò che si è fatto consiste nell'utilizzare un'unica installazione MDS, un'unica script d'avvio per i vari demoni ed un'unica locazione condivisa contenente i file di configurazione dei nodi, ottenendo un unico punto di accesso e di gestione a MDS; infine si è esportata l'intera struttura, per una consultazione ed interrogazione delle risorse disponibili, attraverso il Web.

La chiave di volta è stata la creazione di una directory, chiamata `nodes`, a partire dalla `$GLOBUS_LOCATION`; all'interno di essa sono state quindi create 9 directory, una per ogni macchina del cluster GRID, denominate ognuna con il nome dell'host del singolo nodo, ad esempio:

```
/usr/local/globus/nodes/gw
/usr/local/globus/nodes/n01
```

```
/usr/local/globus/nodes/n02
...
```

Si sono poi copiati tutti i file di configurazione necessari ad MDS dalla directory principale `/usr/local/globus/etc` ad ogni singola *sub–directory* sovraesposta, modificando, per ogni nodo, i relativi parametri di configurazione, in particolare:

- **grid-info.conf**:

  le macro `GRID_INFO_HOST` e `hostname` vengono assegnate al nodo specifico, ad esempio `GRID_INFO_HOST="n01"` e `hostname="n01"` per il primo nodo (sub–directory `n01`);

- **grid-info-site-policy.conf**:

  viene specificata la porta TCP di comunicazione (di solito la 2135) ed una policy (di solito uguale per tutti i nodi) che consenta la registrazione dei GRIS autorizzati, ad esempio:

  `policydata: (&(Mds-Service-hn=*.grid.unipg.it)(Mds-Service -port=2135));`

- **grid-info-resource-ldif.conf**:

  tutte le informazioni dei *provider* disponibili debbono essere relative al nodo specifico (andranno quindi sostituite tutte le occorrenze di un nodo con il nome della macchina desiderata);

- **grid-info-resource-register.conf**:

  il valore di `reghn` sarà sempre quello del *front-end*, esso è l'indirizzo del GIIS al quale i vari GRIS dovranno afferire; il valore di `hn` sarà invece quello del nodo specifico, ovvero il GRIS locale;

- **grid-info-slapd.conf**:

  indica le locazioni di alcuni file di configurazione; per ogni nodo occorrerà quindi inserire il giusto percorso a tali file.

Come già evidenziato, la script d'avvio `SXXgris` (anch'essa locata in una directory condivisa, `/usr/local/globus/sbin`) legge questi file per avviare il demone LDAP `slapd`: anche in questo caso risulta necessaria una piccola modifica al codice originale di Globus; in particolare verrà inserita, all'interno di tale script, un'istruzione del tipo:

```
# export sysconfdir=/usr/local/globus/nodes/`hostname`
```

Tale istruzione andrà a sovrascrivere il valore di default della variabile d'ambiente `sysconfdir`, `/usr/local/globus/etc`, con un valore che dipenderà dal nodo dal quale la script d'avvio è stata invocata, andando quindi a consultare i giusti file di configurazione.

In pratica viene fatto partire MDS da tutti i nodi con lo stesso comando `SXXgris start` (condiviso); esso legge i corretti file di configurazione per

ogni nodo GRIS (suddivisi in sub–directory nel *front-end*), il quale registrerà poi le proprie informazioni sull'unico GIIS definito.

Per verificare quanto realizzato, basterà interrogare MDS in questo modo:

```
# grid-info-search -x -b "Mds-Vo-name=site, o=Grid"
```

oppure con

```
# ldapsearch -x -h gw.grid.unipg.it -p 2135 -b
"Mds-Vo-name=site, o=grid"
```

e verranno restituite tutte le informazioni sulle risorse di tutti i nodi definiti per il sito (compreso il *front-end*); è comunque possibile specificare molteplici parametri per costruire *query* complesse o filtri (si rimanda alle *man page* dei relativi comandi per le sintassi specifiche).

Se si vogliono invece ottenere le stesse informazioni, ma per un solo nodo, si può scrivere:

```
# grid-info-search -h gw -x -b "Mds-Vo-name=local, o=Grid"
```

In questo caso (si noti il flag `-x`, utilizzato per eseguire *query* di tipo *anonymous* in questi esempi) viene quindi effettuata una richiesta a "local" e non a "site", riferita al nodo `gw` (si veda l'**Appendice C** per un tipico *output* relativo al *front-end*).

Come accennato, l'ultimo passo è stato quello di utilizzare un *browser* LDAP (sottoforma di *Applet Java*) per consultare le risorse della griglia com-

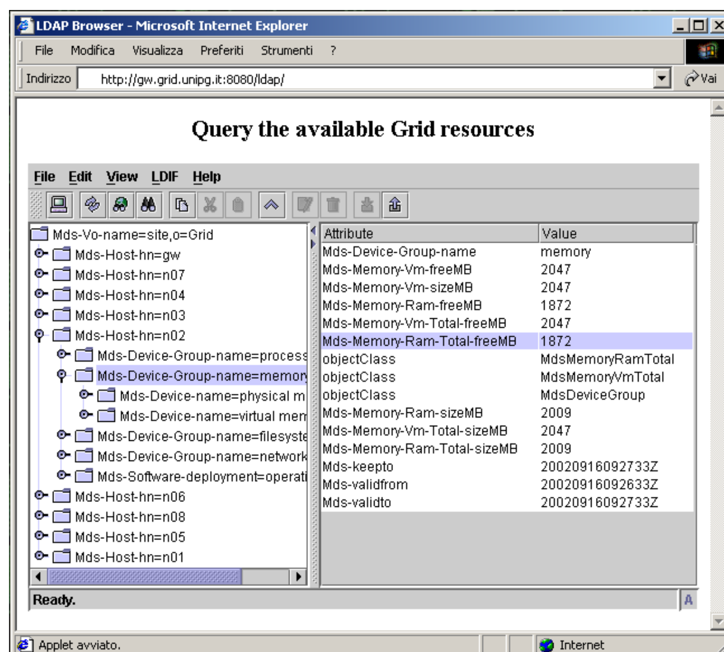

Figura 4.15: **Interrogazione via Web delle risorse Grid.**

putazionale attraverso il Web; in figura **4.15** è visibile una tipica *query* sulle risorse del nodo `n02`.

Tale strumento può inoltre essere utilizzato anche in maniera *stand-alone* sulla propria *workstation*, senza dover quindi necessariamente consultare un sito Web.

Anche in questo caso l'obiettivo prefissato è stato raggiunto: si è infatti ottenuto un unico punto di accesso e di gestione per ogni risorsa di griglia (nel nostro caso un GIIS per ogni cluster); è comunque possibile, laddove esista una relazione di dipendenza gerarchica fra le strutture, organizzare i vari GIIS in modo da costruire degli alberi che rappresentino delle vere e

proprie organizzazioni distribuite (chiamate **VO**, *Virtual Organization*).

In effetti, MDS oggi aiuta l'utente a conoscere le risorse disponibili e le sue caratteristiche; comunque, un notevole valore aggiunto potrebbe essere ottenuto implementando uno *scheduler* (di cui il *Globus Toolkit* ne è privo) che, interagendo con MDS, si preoccupi di organizzare (eventualmente rifiutandole) le varie richieste a livello gerarchico: è infatti ipotizzabile la creazione di un "*meta–scheduler*" (chiamato così per non confonderlo con un qualsiasi *scheduler* locale sulle varie entità di griglia) che operi sopra i vari GIIS definiti in un'unica o più VO.

4.3 MPICH-G2: Esempi

In questa sezione verranno forniti due esempi di applicazioni MPICH-G2 complete: la prima è una tipica applicazione MPI volta a testare il corretto funzionamento dell'architettura realizzata, mentre la seconda illustrerà l'utilizzo delle macro `MPICHX_TOPOLOGY_DEPTHS` e `MPICHX_TOPOLOGY_COLORS` descritte in precedenza.

L'applicazione *ring*

Il programma **ring** illustra come programmi MPI tipici, basati ad esempio sul concetto di *master/slave*, possano lavorare in maniera trasparente sopra una griglia computazionale (formata quindi da entità eterogenee) attraverso MPICH-G2.

In particolare, **ring** realizza uno scambio circolare di un numero intero crescente tra n processori, dove il primo (`my_id = 0`) è il *master*, mentre gli altri sono gli *slave*.

Ciò che accade è che per ogni esecuzione il *master* inizierà l'invio di un nuovo numero, a partire dal numero 1, allo *slave* di *id* successivo; esso lo riceve (`MPI_Recv`), e lo spedisce (`MPI_Send`), incrementato di 1, allo *slave* seguente, e così via: l'ultimo *slave* ritornerà il controllo al *master*.

L'applicazione (il cui codice sorgente è riportato in **Appendice D**, sez.

D-1) può essere compilata con il `Makefile` riportato in figura **4.16**.

```
# Assumes MPICH-G2 was installed in /usr/local/mpichg2
#
MPICH_INSTALL_PATH = /usr/local/mpichg2
ring: force
        $(MPICH_INSTALL_PATH)/bin/mpicc -o ring ring.c
force:
clean:
        /bin/rm -rf *.o ring
```

Figura 4.16: ***Makefile* per la compilazione dell'applicazione d'esempio *ring*.**

In seguito è possibile eseguirla sulla griglia con un comando del tipo:

```
# globusrun -w -f ring.rsl
```

attraverso la script RSL riportata in figura **4.17** (in questo caso vengono richiesti tutti i 42 processori della griglia computazionale); l'*output* ottenuto è quindi:

```
Master: end of trip 1 of 1: after receiving passed_num=42
(should be =trip*numprocs=42) from source=41
```

Questo risultato si può interpretare dicendo che, in un unico "trip" (letteralmente *viaggio*), 42 processori (indipendentemente dalla loro locazione fisica) si sono passati il numero `passed_num`, a partire dal *master*, fino a che l'ultimo *slave* (`source=41`) ha inviato l'ultimo numero di nuovo al controllore.

```
+
( &(resourceManagerContact="fe.giza.unipg.it")
   (count=8)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/ring")
)
( &(resourceManagerContact="gw.grid.unipg.it")
   (count=18)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/ring")
)
( &(resourceManagerContact="bwcw1.hpc.thch.unipg.it")
   (count=16)
   (jobtype=mpi)
   (label="subjob 2")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/ring")
)
```

Figura 4.17: **ring.rsl, script RSL per l'esecuzione dell'applicazione d'esempio *ring*.**

Il programma **ring** inoltre, può ricevere due parametri opzionali, `t` (che indica il numero di *trip*) e `v` (che restituirà un *output* dettagliato); se volessimo inserire questi due nuovi attributi (che di default sono settati rispettivamente a 1 e 0) nella script RSL basterà aggiungere un'*entry* così fatta:

```
(arguments= "-t" "2" "-v")
```

per ogni *subjob*.

Per chiarezza, si riporta in figura **4.18** l'*output* restituito dalla script precedente impostando il parametro `t` a 2 (senza il *verbose*).

```
Master: end of trip 1 of 2: after receiving passed_num=42
(should be =trip*numprocs=42) from source=41
Master: end of trip 2 of 2: after receiving passed_num=84
(should be =trip*numprocs=84) from source=41
```

Figura 4.18: ***Output* dell'applicazione d'esempio *ring* sulla griglia computazionale (opzione *trip* = 2).**

Si noti che i processori sono sempre 42 (da 0 a 41); questa volta però arrivano a scambiarsi `passed_num` fino al valore 84, dato che vengono compiuti 2 viaggi.

Un'ultimo risultato viene mostrato per chiarire definitivamente l'esempio e per illustrare un'alternativa al comando `globusrun` per l'invio dei job; in effetti potremmo lanciare qualsiasi job MPICH-G2 anche in questo modo:

```
# /usr/local/mpichg2/bin/mpirun -np 4 ring -t 2 -v
```

ovvero attraverso la sintassi consueta del comando `mpirun` (MPICH-G2).

In questo caso, vengono richiesti solo 4 processori, 2 viaggi e l'*output* esteso; laddove non si utilizzino script RSL (peraltro create automaticamente dal comando `mpirun` per l'effettivo `submit` e poi rimosse) oltre che una più difficile esplicitazione della distribuzione delle risorse, avremmo una possibilità ridotta di parametrizzare il job sopra la griglia (ad esempio non risulta possibile specificare percorsi diversi degli eseguibili sulle varie entità di griglia) ma soprattutto occorrerà prestare attenzione al file `machines` (descritto in **3.2**) il quale determinerà la creazione o meno dei diversi *subjob* (si noti come

questo file assuma importanza *solo* quandi si utilizza il comando `mpirun` al posto di `globusrun`).

Infatti, se tale file fosse così definito:

```
"gw.grid.unipg.it" 18
```

i 4 processori dell'esempio precedente verrebbero "presi" tutti dall'entità specificata ed avremmo un unico *subjob*; mentre se fosse:

```
"gw.grid.unipg.it" 2
"fe.giza.unipg.it" 2
```

verrebbero presi 2 processori per ogni cluster (e quindi 2 *subjob*).

In ogni caso comunque, **ring** dimostra che il risultato ottenuto è trasparente dalla locazione delle varie entità coinvolte; infatti in entrambi i casi l'*output* ottenuto da `mpirun` è quello mostrato in figura **4.19**.

```
my_id 0 numprocs 4
Master: starting trip 1 of 2: before sending num=1 to dest=1
Master: inside trip 1 of 2: before receiving from source=3
Master: end of trip 1 of 2: after receiving passed_num=4
(should be =trip*numprocs=4) from source=3
Master: starting trip 2 of 2: before sending num=5 to dest=1
Master: inside trip 2 of 2: before receiving from source=3
Master: end of trip 2 of 2: after receiving passed_num=8
(should be =trip*numprocs=8) from source=3

my_id 1 numprocs 4
Slave 1: top of trip 1 of 2: before receiving from source=0
Slave 1: inside trip 1 of 2: after receiving passed_num=1
from source=0
Slave 1: inside trip 1 of 2: before sending passed_num=2
to dest=2
Slave 1: bottom of trip 1 of 2: after send to dest=2
Slave 1: top of trip 2 of 2: before receiving from source=0
Slave 1: inside trip 2 of 2: after receiving passed_num=5
from source=0
Slave 1: inside trip 2 of 2: before sending passed_num=6
to dest=2
Slave 1: bottom of trip 2 of 2: after send to dest=2

my_id 2 numprocs 4
Slave 2: top of trip 1 of 2: before receiving from source=1
Slave 2: inside trip 1 of 2: after receiving passed_num=2
from source=1
Slave 2: inside trip 1 of 2: before sending passed_num=3
to dest=3
Slave 2: bottom of trip 1 of 2: after send to dest=3
Slave 2: top of trip 2 of 2: before receiving from source=1
Slave 2: inside trip 2 of 2: after receiving passed_num=6
from source=1
Slave 2: inside trip 2 of 2: before sending passed_num=7
to dest=3
Slave 2: bottom of trip 2 of 2: after send to dest=3

my_id 3 numprocs 4
Slave 3: top of trip 1 of 2: before receiving from source=2
Slave 3: inside trip 1 of 2: after receiving passed_num=3
from source=2
Slave 3: inside trip 1 of 2: before sending passed_num=4
to dest=0
Slave 3: bottom of trip 1 of 2: after send to dest=0
Slave 3: top of trip 2 of 2: before receiving from source=2
Slave 3: inside trip 2 of 2: after receiving passed_num=7
from source=2
Slave 3: inside trip 2 of 2: before sending passed_num=8
to dest=0
Slave 3: bottom of trip 2 of 2: after send to dest=0
```

Figura 4.19: ***Output* dell'applicazione d'esempio *ring* su 4 processori indipendentemente dalla loro locazione fisica (opzione *trip* = 2).**

Si noti infine come tali operazioni possano essere rappresentate attraverso una macchina a stati finiti (fig. **4.20**), determinati dai 4 processori.

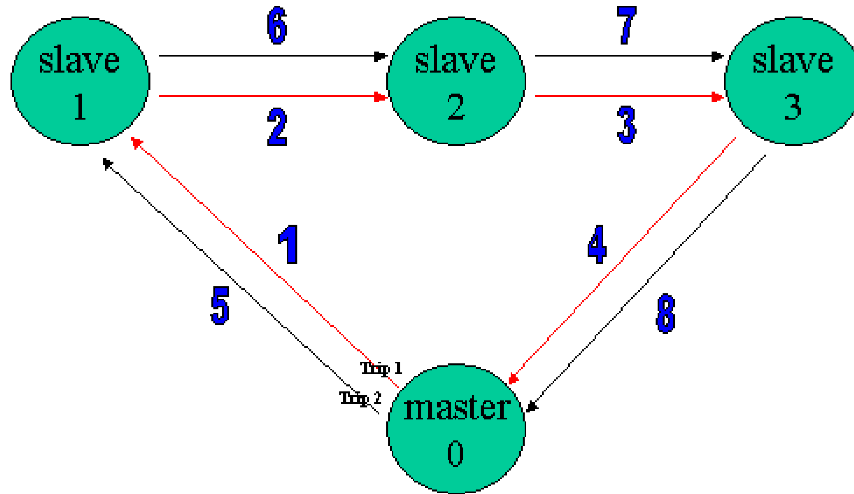

Figura 4.20: **Rappresentazione dell'applicazione *ring* attraverso una macchina a stati finiti.**

L'applicazione *report_colors*

Il programma **report_colors** illustra l'utilizzo degli attributi relativi alla scoperta e definizione della topologia della griglia computazionale.

In particolare, attraverso l'utilizzo delle due macro già evidenziate nel preambolo di questa sezione, si vuole rispondere a questa domanda:

> Può un processo **A** comunicare con un processo **B** ad un certo livello ***i***?

Secondo quanto già evidenziato, ci si aspetta che due processi sulla griglia possano parlare fino al livello **lv3**, cioè quello *intra-machine MPI* (quarto ed ultimo livello); l'applicazione (il cui codice sorgente è riportato in **Appendice D**, sez. **D-2**) può essere compilata con il `Makefile` mostrato in figura

4.21.

```
# Assumes MPICH-G2 was installed in /usr/local/mpichg2
#
MPICH_INSTALL_PATH = /usr/local/mpichg2
report_colors: force
        $(MPICH_INSTALL_PATH)/bin/mpicc -o report_colors \
report_colors.c
force:
clean:
        /bin/rm -rf *.o report_colors
```

Figura 4.21: ***Makefile*** **per la compilazione dell'applicazione d'esempio** ***report_colors*.**

In seguito è possibile eseguirla sulla griglia con un comando del tipo:

```
# globusrun -w -f report_colors.rsl
```

attraverso la script RSL riportata in figura **4.22** (vengono richiesti 8 processori sia per il cluster GRID che per il cluster GIZA).

```
+
( &(resourceManagerContact="gw.grid.unipg.it")
   (count=8)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/report_colors")
)
( &(resourceManagerContact="fe.giza.unipg.it")
   (count=8)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
                (LD_LIBRARY_PATH /usr/local/globus/lib/))
   (directory="/home/carlo")
   (executable="/home/carlo/report_colors")
)
```

Figura 4.22: **report_colors.rsl, script RSL per l'esecuzione dell'applicazione d'esempio *report_colors*.**

In questo caso vengono lanciati in parallelo 8 processi per ogni *Gatekeeper*; l'*output* ottenuto è riportato in figura **4.23**.

Tale risultato si può intepretare dicendo che non si verifica alcun dialogo a livello lv0=WAN-TCP (essendo gli 8 processi su due *subjob* separati) mentre si hanno comunicazioni sino al livello lv3=*intra-machine MPI*.

```
proc     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
Depths   4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4
lvl 0    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
lvl 1    0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
lvl 2    0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
lvl 3    0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
```

Figura 4.23: ***Output* dell'applicazione d'esempio *report_colors* su una griglia formata da 16 processori (cluster GRID e GIZA).**

In particolare si noti che la profondità della comunicazione è sempre 4 (`MPICHX_TOPOLOGY_DEPTHS`) e che i vettori (`MPICHX_TOPOLOGY_COLORS`), identificati dalle varie colonne, hanno sempre una lunghezza pari a 4 (a partire da **lv0** fino a **lv3**); ovvero possono sempre, in potenza, dialogare o meno fino a quel livello (ovviamente l'effettiva comunicazione rilevata dipenderà poi dal particolare programma, dai suoi parametri e dalle entità di griglia utilizzate).

4.4 Testbed

In quest'ultima sezione vengono riportati i risultati ottenuti attraverso i programmi di *benchmark* MPICH standard, utilizzabili con qualsiasi implementazione MPI, `mpptest` e `goptest`, contenuti nel pacchetto dei sorgenti di MPICH nella directory `examples/perftest`.

Tali test, utilizzati fra l'altro nel sito ufficiale di MPICH-G2 per la valutazione di ambienti analoghi su sistemi SGI, IBM e SUN, sono stati ovviamente modificati in modo da poter essere eseguiti sopra la griglia.
In particolare, sono stati compilati con MPICH-G2 in questo modo:

```
# ./configure --with-mpich=/usr/local/mpichg2/; make
```

Successivamente, si sono modificate le script d'avvio dei test avvinchè generassero i file RSL (a volte prodotti manualmente), necessari per `globusrun`, con comandi del tipo:

```
# ./runmpptest -short -pair -blocking -givedy -gnuplot      \
 -fname p2p-short.mpl -mpirunpgm                            \
 /usr/local/mpichg2/bin/mpirun -mpirun \-dumprsl
# ./runmpptest -long -pair -nonblocking -givedy -gnuplot    \
 -fname nb-p2p-long.mpl -mpirunpgm                          \
 /usr/local/mpichg2/bin/mpirun -mpirun \-dumprsl
# ./rungoptest -maxnp 42 -add -bcast -gnuplot -fname        \
 bcast.mpl -mpirunpgm /usr/local/mpichg2/bin/mpirun         \
 -mpirun \-dumprsl
```

Ovviamente, prima di effettuare il *run* vero e proprio, è necessario copiare

(preferibilmente sugli stessi *path*) i programmi di test su tutti i tre cluster (per qualsiasi errore relativo a GRAM consultare l'**Appendice E**).

Si ricorda inoltre che è possibile eseguire, attraverso MPICH-G2, qualsiasi programma già scritto secondo il paradigma di programmazione MPI standard; l'unica differenza, in contrapposizione ad un codice adattato per sfruttare la topologia della griglia (ad esempio un algoritmo che cerca di ridurre al minimo le comunicazioni WAN o MAN attraverso la costituzione di insiemi di processori di livello **lv3** che comunicano fra loro meno volte possibile, assolvendo invece, 'al loro interno', le parti che richiedono frequenti comunicazioni), è che tali *job* lavoreranno sulla base della conoscenza della topologia sottostante, ovvero, in questo caso, una griglia computazionale che non tiene conto della locazione fisica delle risorse.

4.4.1 mpptest

mpptest è un programma che analizza le *performance* di routine MPI standard (tipicamente le chiamate `MPI_Send` e `MPI_Receive`) in diverse situazioni [31], quali ad esempio la valutazione di sistemi paralleli Grid *MPICH-G2 enabled*, per ciò che concerne la misura delle comunicazioni in relazione alle dimensioni del problema (ad esempio la dimensione in *bytes* dei messaggi), il protocollo di comunicazione sincrono (bloccante) o asincrono (non bloccante) ed ovviamente al tempo di elaborazione.

I risultati vengono presentati in 5 sezioni: prestazioni dei cluster GRID, HPC e GIZA, prestazioni *point–to–point* tra i vari cluster e *performance* dell'intera griglia computazionale.

Come verrà discusso, il cluster GRID presenta prestazioni nettamente superiori agli altri due cluster dato che la rete di connessione tra i nodi è di tipo *Gigabit Ethernet*.

Il cluster GIZA presenta prestazioni superiori al cluster HPC in quanto ogni nodo è interconnesso agli altri tramite due NIC *Fast Ethernet* in *channel bonding*.

Infine, dato che i test sono stati eseguiti a macchine dedicate, durante *week–end* per essere in presenza di un traffico ridotto tra i cluster, non si osserva alcuna differenza significativa tra l'utilizzo di un protocollo bloccante o non bloccante.

Prestazioni del cluster GRID

Si illustrano i grafici dei risultati al variare dei messaggi (*short* e *long*, rispettivamente fino a 1K e 64K) ed al tipo di protocollo utilizzato (sincrono o asincrono, rispettivamente utilizzando *send/receive* bloccanti o non bloccanti); si calcoli (a meno dell'incertezza, trascurabile, a volte rilevata) come il cluster GRID raggiunga circa i $50 Kbytes/sec$, per messaggi fino ad 1K, e circa i $125 Kbytes/sec$ per messaggi fino a 64K, con entrambi i protocolli (fig. **4.24**, **4.25** e fig. **4.26**).

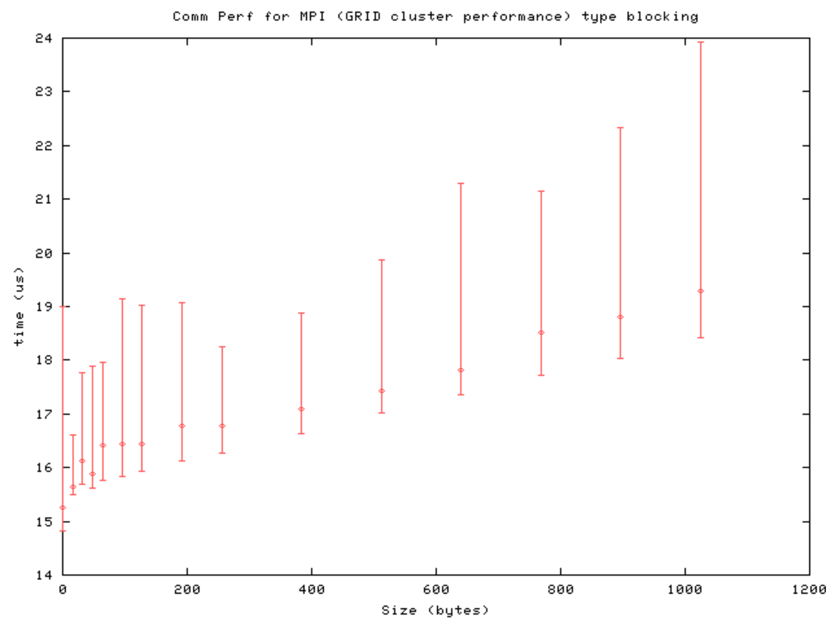

Figura 4.24: **Prestazioni del cluster GRID con messaggi corti e protocollo sincrono.**

Come illustrato nell'**Appendice A**, le caratteristiche del cluster GRID sono molto avanzate dal punto di vista della tecnologia disponibile, in quanto

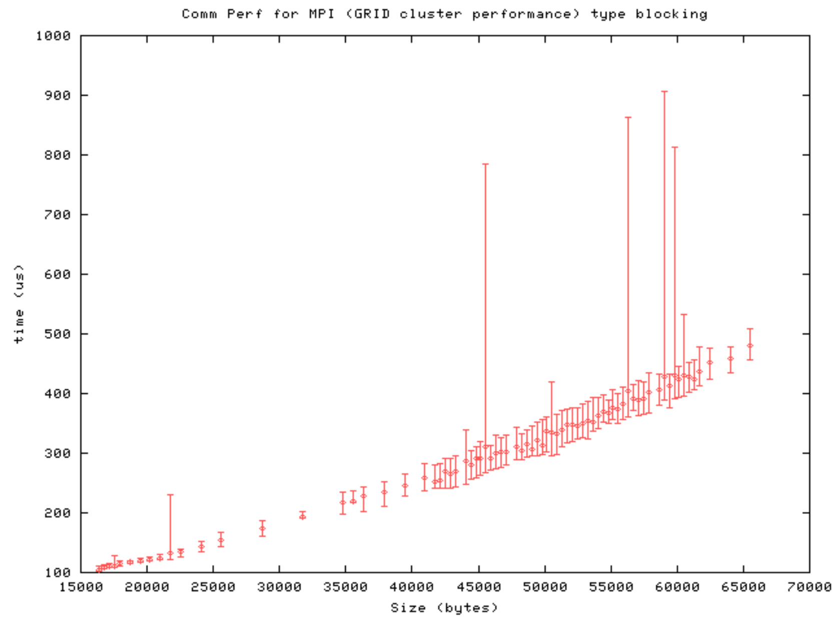

Figura 4.25: **Prestazioni del cluster GRID con messaggi lunghi e protocollo sincrono.**

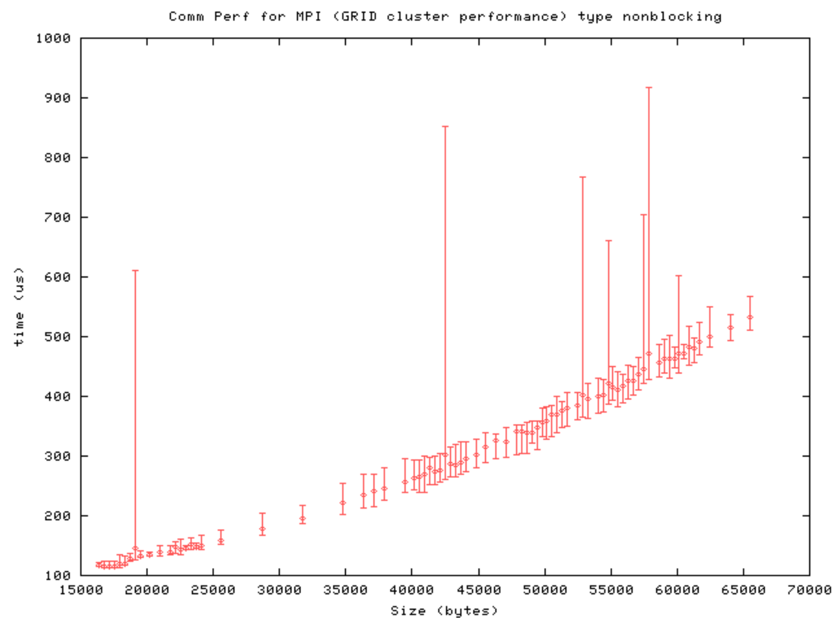

Figura 4.26: **Prestazioni del cluster GRID con messaggi lunghi e protocollo asincrono.**

si hanno processori molto recenti e l'infrastruttura di collegamento tra i nodi è *Gigabit Ethernet*.

Questo spiega perchè nei grafici si osservano tempi di comunicazione ottimali e delle barre di deviazione d'errore eccellenti (le sole misure d'incertezza sono relative a meno di 6 millisecondi).

Prestazioni del cluster GIZA

Ancora, si illustrano i grafici dei risultati al variare dei messaggi ed al tipo di protocollo utilizzato; si calcoli come il cluster GIZA raggiunga circa i $4 Kbytes/sec$, per messaggi fino ad 1K, e circa i $18 Kbytes/sec$ per messaggi fino a 64K, con entrambi i protocolli (fig. **4.27**, **4.28** e fig. **4.29**).

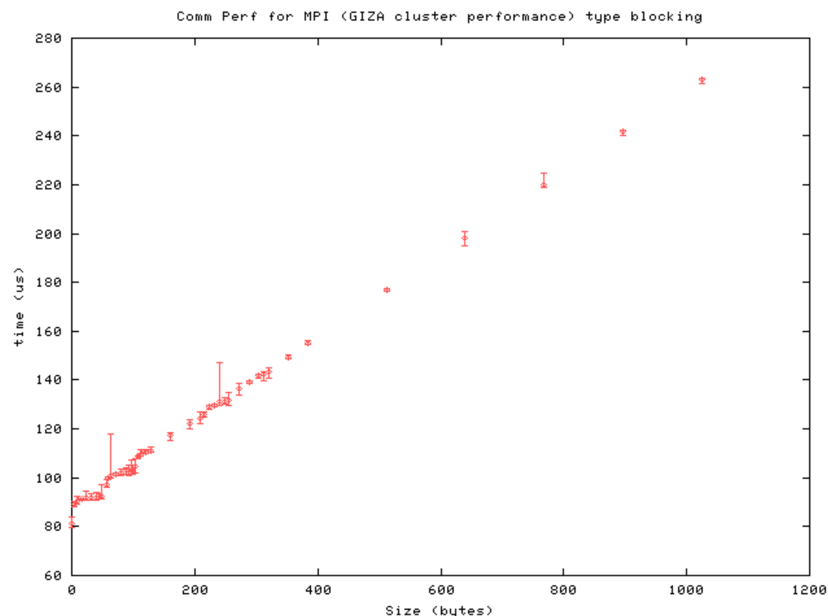

Figura 4.27: **Prestazioni del cluster GIZA con messaggi corti e protocollo sincrono.**

Come si evince dall'**Appendice A**, il cluster GIZA è dotato di macchine monoprocessore e di una rete di interconnessione dei nodi con 2 NIC *Fast Ethernet* in *channel bonding*.

Questo rende conto del fatto che le prestazioni del cluster GIZA sono superiori a quelle di HPC (trattato nella prossima sezione), pur mantenendosi

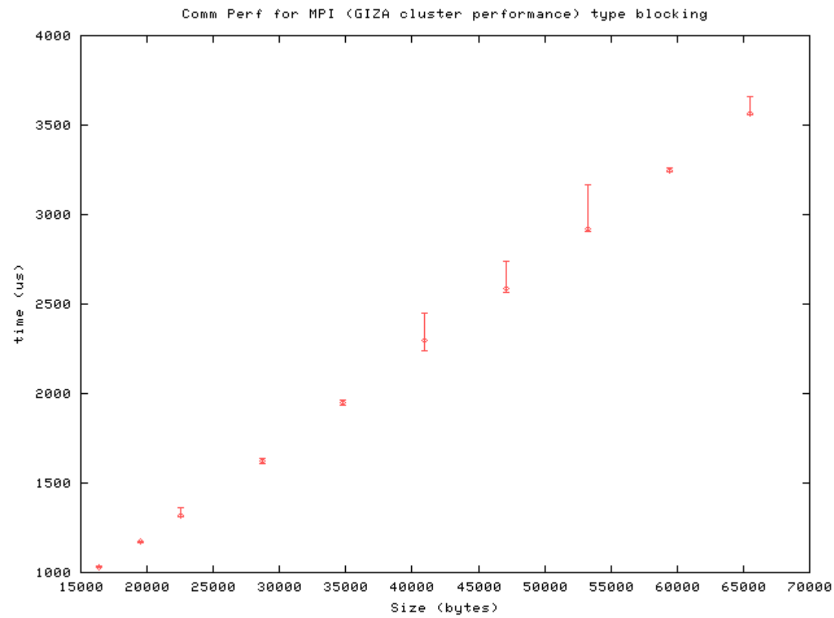

Figura 4.28: **Prestazioni del cluster GIZA con messaggi lunghi e protocollo sincrono.**

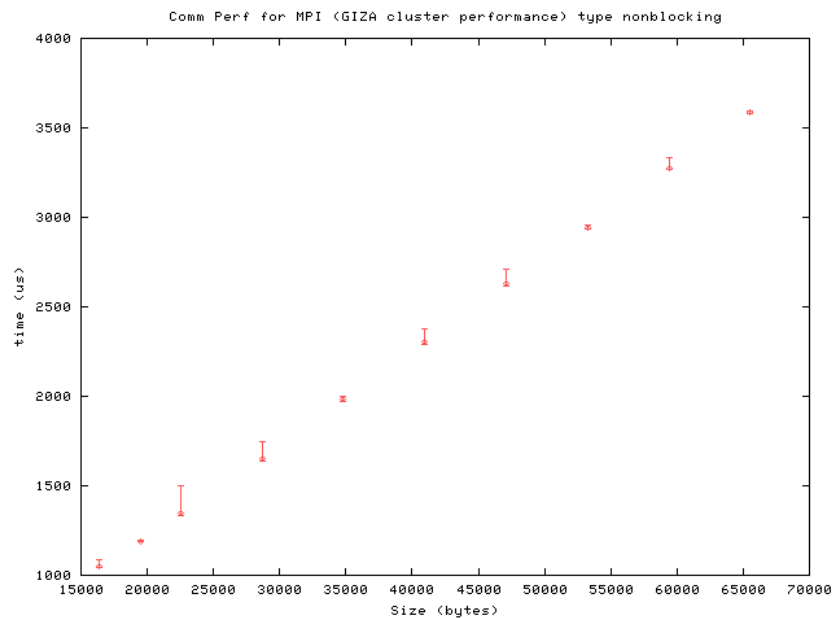

Figura 4.29: **Prestazioni del cluster GIZA con messaggi lunghi e protocollo asincrono.**

nettamente inferiori a quelle del cluster GRID.

Prestazioni del cluster HPC

Di nuovo, si illustrano i grafici dei risultati al variare dei messaggi (*short* e *long*) ed al tipo di protocollo utilizzato (sincrono o asincrono); si calcoli (trascurando l'incertezza) come il cluster HPC raggiunga circa i $3Kbytes/sec$, per messaggi fino ad 1K, e circa i $10Kbytes/sec$ per messaggi fino a 64K, con entrambi i protocolli (fig. **4.30**, **4.31** e fig. **4.32**).

In questo caso, come si può osservare in **Appendice A**, le caratteristiche del cluster HPC sono peggiorative (soprattutto per ciò che riguarda la rete di interconnessione tra i nodi) rispetto a quelle di GRID e GIZA, come conferma il valore assoluto dei tempi di comunicazione riportati.

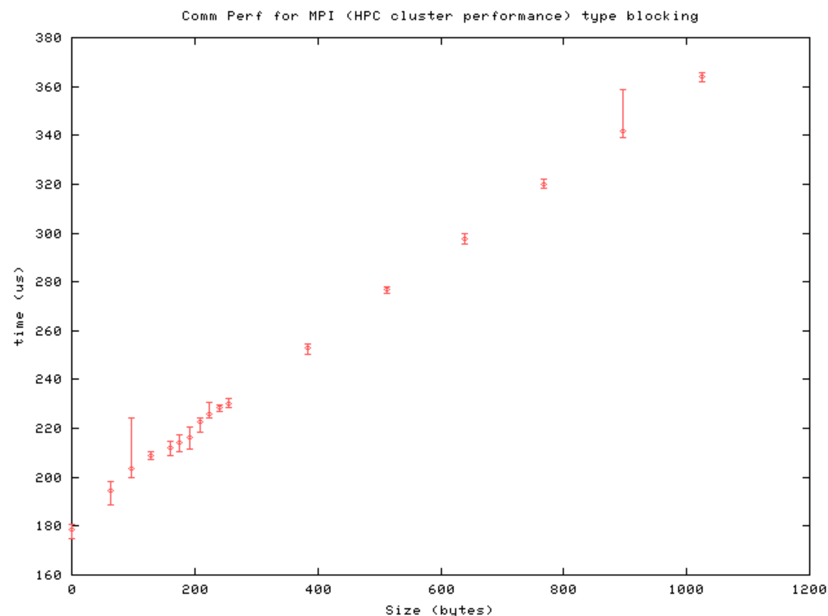

Figura 4.30: **Prestazioni del cluster HPC con messaggi corti e protocollo sincrono.**

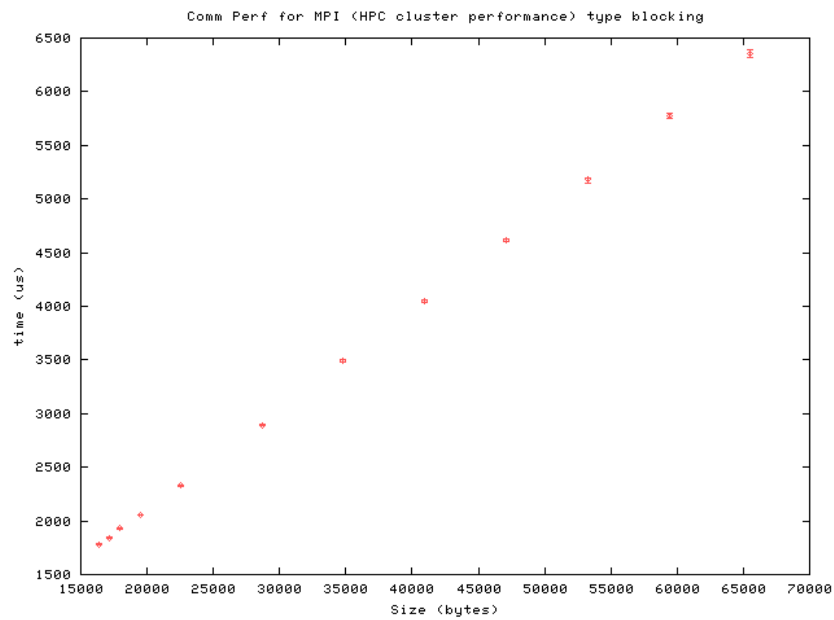

Figura 4.31: **Prestazioni del cluster HPC con messaggi lunghi e protocollo sincrono.**

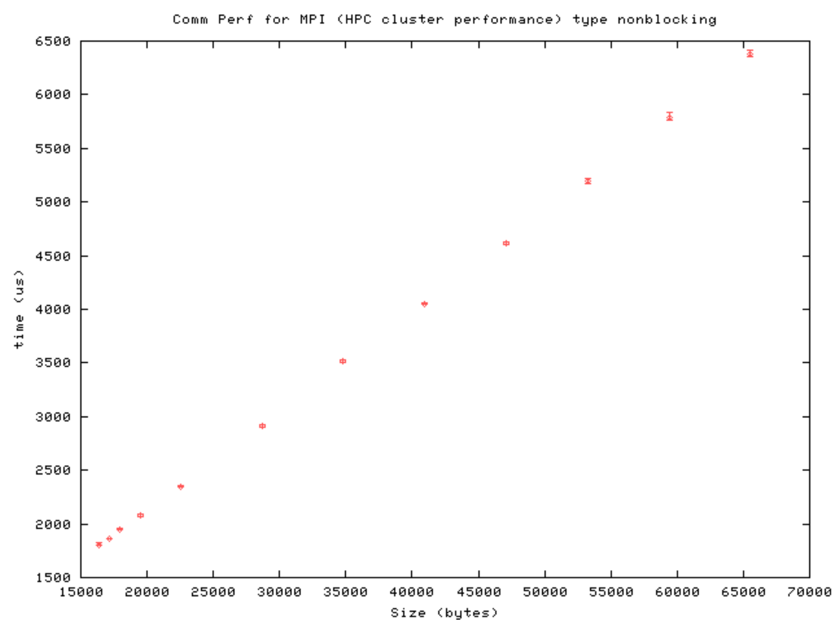

Figura 4.32: **Prestazioni del cluster HPC con messaggi lunghi e protocollo asincrono.**

Prestazioni *point–to–point inter-cluster*

In questa sezione, si illustrano i risultati dei test *point–to–point* (**p2p**) tra i diversi cluster (con messaggi corti e *send/receive* bloccanti), rispettivamente tra HPC e GRID, HPC e GIZA e tra GRID e GIZA (fig. **4.33**, **4.34** e fig. **4.35**).

I tempi della misura aumentano drasticamente rispetto a quelli delle prestazioni *intra-cluster*, in particolare quando viene coinvolto il cluster GIZA (quello fisicamente più lontano), in quanto GRID e HPC sono interconnessi attraverso una rete locale, mentre GIZA è raggiungibile attraverso una rete WAN ATM a 16 Mbit.

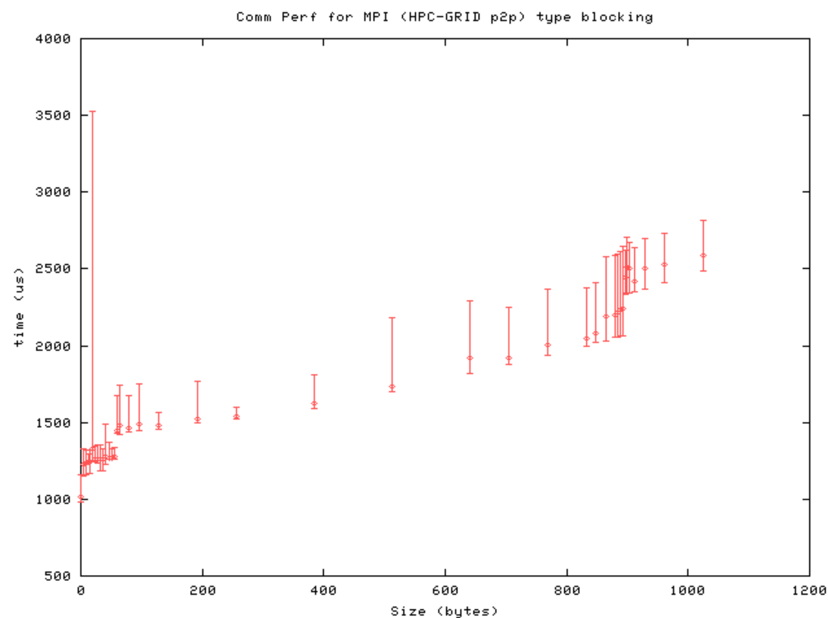

Figura 4.33: **Comunicazioni *point–to–point* tra HPC e GRID con messaggi corti e protocollo sincrono.**

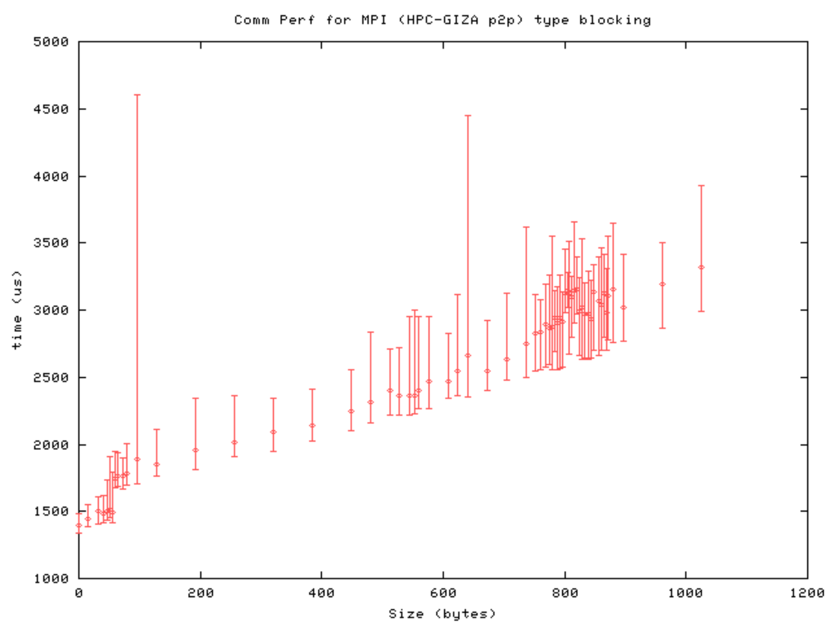

Figura 4.34: **Comunicazioni *point–to–point* tra HPC e GIZA con messaggi corti e protocollo sincrono.**

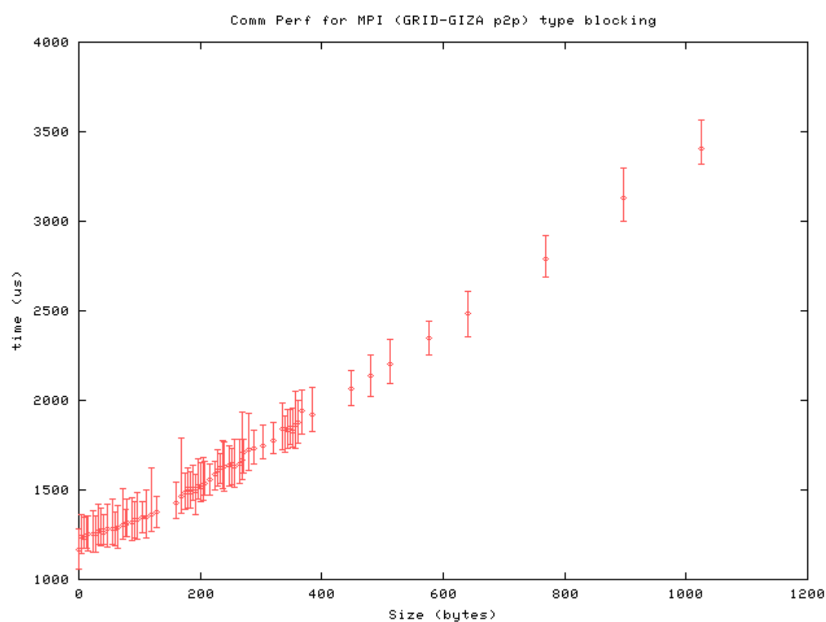

Figura 4.35: **Comunicazioni *point–to–point* tra GRID e GIZA con messaggi corti e protocollo sincrono.**

Prestazioni della griglia computazionale

In quest'ultima sezione di **mpptest** si illustrano due grafici che riportano le prestazioni di comunicazioni MPI sopra l'intera griglia, rispettivamente con messaggi *short* e *long* (fig. **4.36** e **4.37**), a protocollo di comunicazione sincrono con *message pattern* di tipo *roundtrip* (sarà quindi il *sender* che comperà tutte le operazioni per il calcolo del tempo impiegato) in assenza di bisezione o *overlapping* (la *bisect* infatti prevede di suddividere in due parti uguali il numero dei processori disponibili per calcolare la banda necessaria alla comunicazione con l'altra metà, cosa che non avrebbe molto senso in un sistema non omogeneo) dove ogni `MPI_Receive` può ricevere da qualsiasi *sender* (`MPI_ANY_SOURCE` *option*).

In questo contesto, vengono raggiunti circa i $18 Kbytes/sec$, per messaggi fino ad 1K, e circa i $121 Kbytes/sec$ per messaggi fino a 64K in presenza di un protocollo di comunicazione bloccante, ovvero dove un *sender* non può inviare fino a che un *receiver* non abbia interamente ricevuto il messaggio a lui precedentemente spedito.

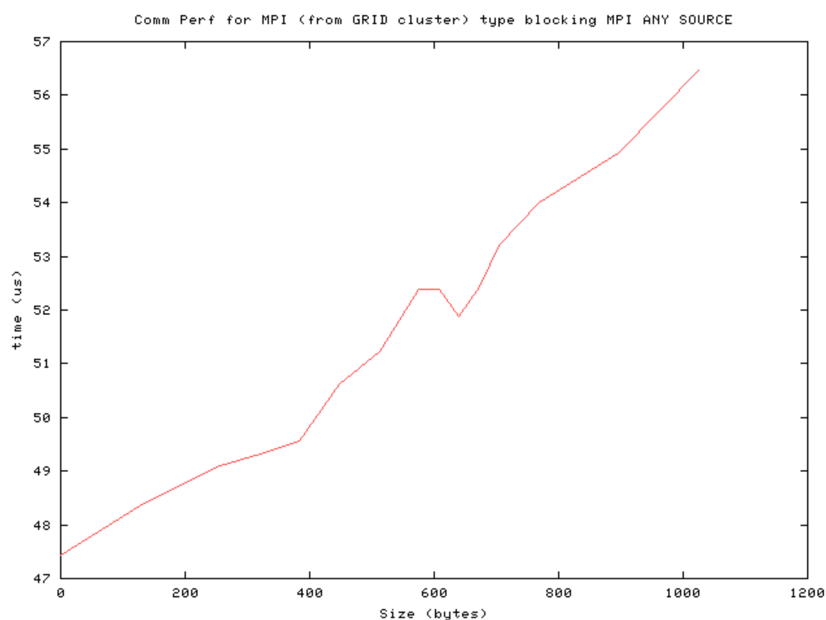

Figura 4.36: ***Performance* dell'intera griglia computazionale con messaggi *short* e protocollo sincrono.**

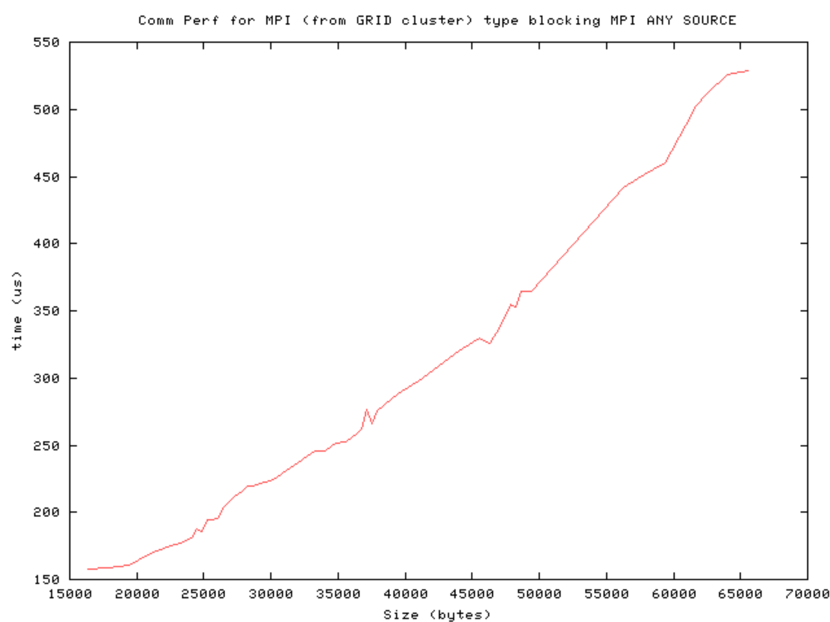

Figura 4.37: ***Performance* dell'intera griglia computazionale con messaggi *long* e protocollo sincrono.**

Alla luce delle misure riportate per l'intera griglia computazionale, considerate le caratteristiche dei singoli cluster, possiamo dire di aver ottenuto degli ottimi risultati, raggiungendo prestazioni che si avvicinano, soprattutto al crescere delle dimensioni dei messaggi (fino a 64K) sia al cluster GRID, il nodo computazionale della griglia più performante, che ai risultati disponibili in letteratura [32] circa gli elaboratori paralleli che adottano MPICH-G2.

4.4.2 goptest

goptest invece è stato utilizzato per misurare le prestazioni della griglia computazionale attraverso broadcast MPI (opzione `-bcast`), evidenziando quindi fattori come la scalabilità dell'intero sistema al crescere del numero dei processori coinvolti.

Di seguito vengono riportati i grafici relativi al variare del numero dei processori da 0 a 42, ovvero arrivando a sfruttare l'intera griglia computazionale, prodotti a partire rispettivamente dai cluster GRID ed HPC (fig. **4.38** e **4.39**).

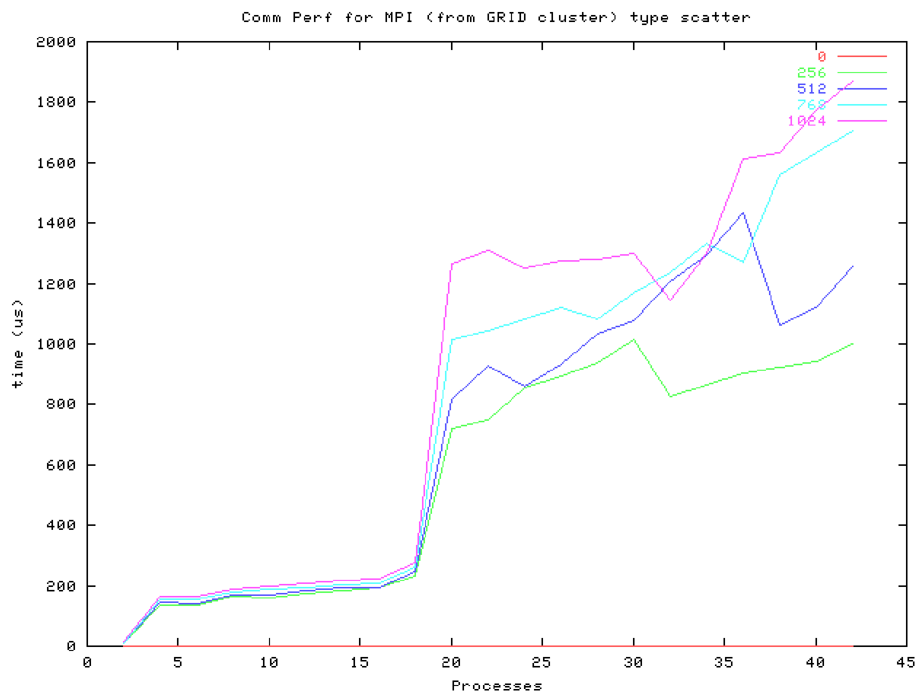

Figura 4.38: **Broadcast MPI sull'intera griglia a partire dal cluster GRID.**

Ciò significa che, nel primo caso, i broadcast avverranno all'interno del

cluster GRID fino a 18 processori, fra i processori dei cluster GRID e HPC fino a 34 e fra tutti i processori dei 3 cluster al coinvolgimento degli ultimi 8; nel secondo caso si avrà lo stesso procedimento a partire dai 16 processori del cluster HPC, a cui si aggiungeranno i 18 processori del cluster GRID ed infine gli 8 processori del cluster GIZA.

Si noti la scalabilità all'interno di ogni singolo cluster, praticamente lineare per il cluster GRID (molto evidente in fig. 4.38 e 4.40, dove GRID viene coinvolto per primo) e rispetto all'intero sistema completo (variando infatti fino a 42 cpu si hanno, al massimo, tempi di comunicazione sempre inferiori ai 2 secondi).

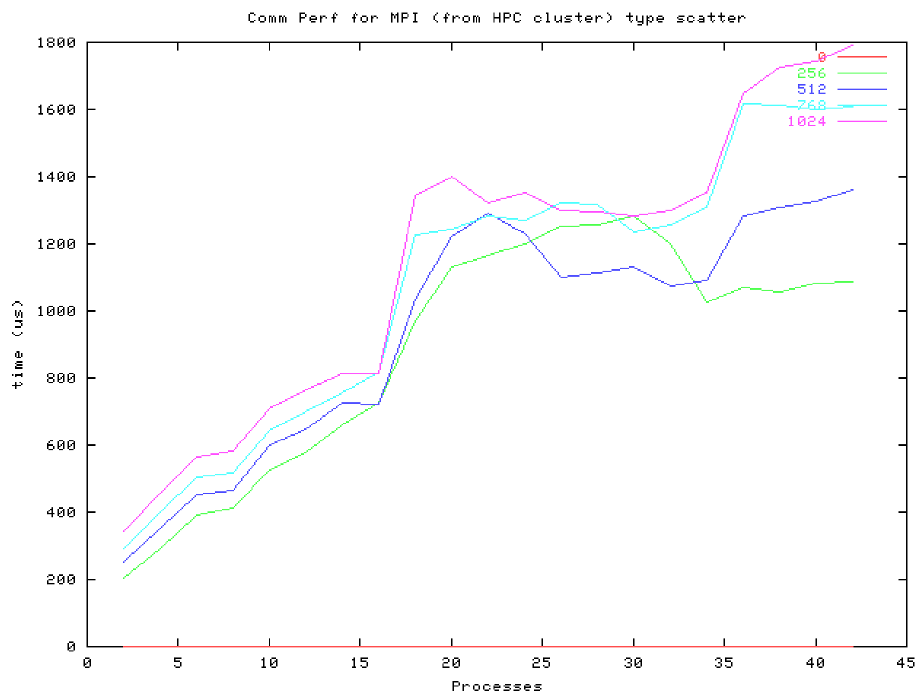

Figura 4.39: **Broadcast MPI sull'intera griglia a partire dal cluster HPC.**

Gli stessi test sono stati poi effettuati utilizzando 8 cpu per cluster (questa volta quindi ogni 8 processori verrà coinvolto il cluster successivo); si osservi come i risultati ottenuti siano pressochè simili, a partire rispettivamente dai cluster GRID e HPC (fig. **4.40** e **4.41**).

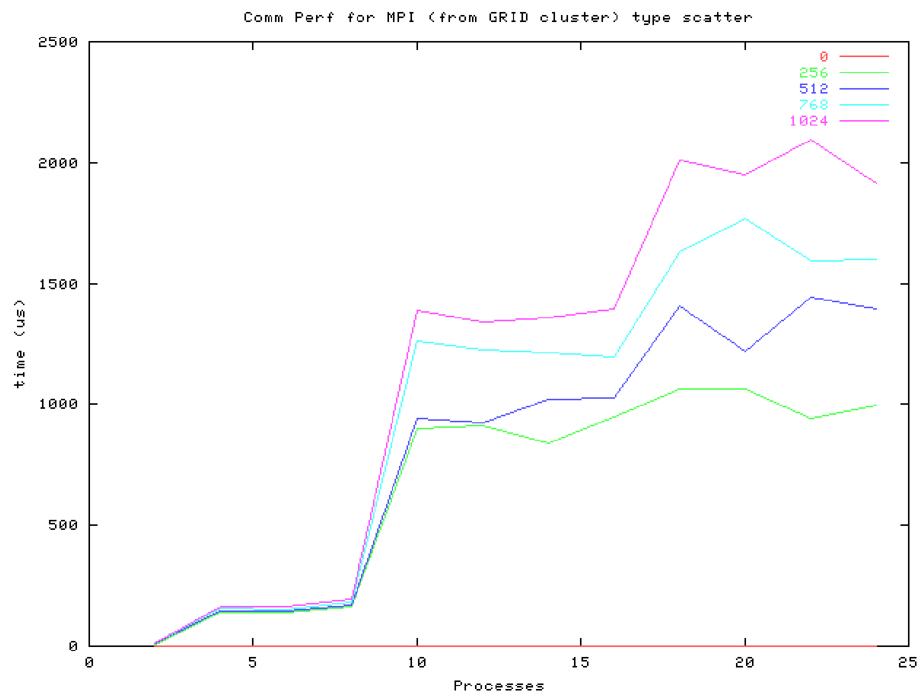

Figura 4.40: **Broadcast MPI fra 24 processori a partire dal cluster GRID.**

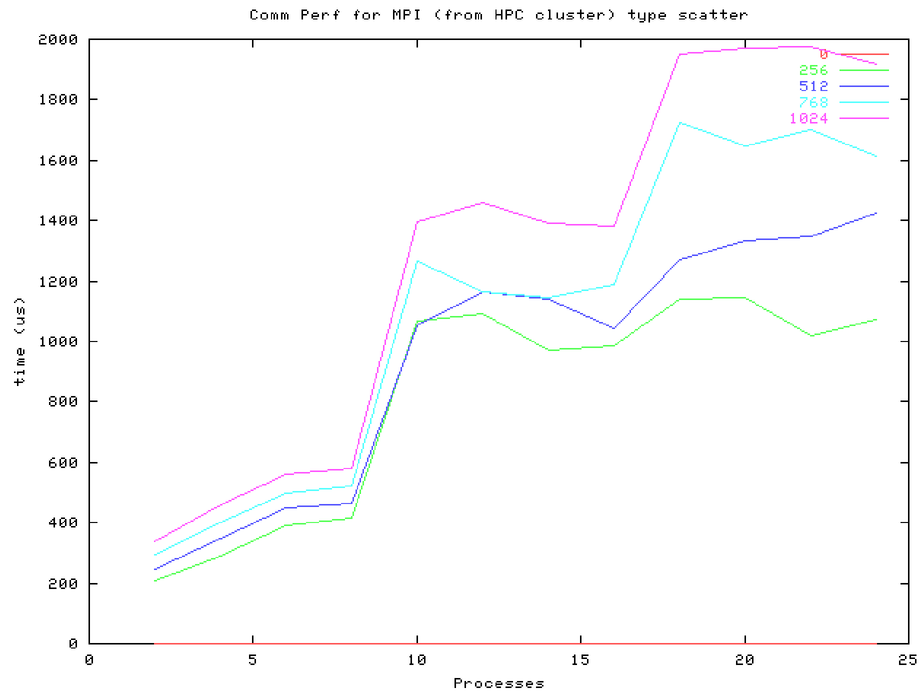

Figura 4.41: **Broadcast MPI fra 24 processori a partire dal cluster HPC.**

La griglia computazionale scala quindi correttamente sia se si considera la singola *cpu* per ogni singolo nodo che l'SMP completo (ovvero tutti i processori di tutti i nodi).

4.4.3 Valutazioni finali

Come risulta dalle diverse misure riportate, la griglia computazionale realizzata, oltre che dimostrare un corretto funzionamento ed una certa stabilità, presenta spunti notevoli per ciò che concerne le prestazioni, in particolare modo circa la scalabilità (sia per variazioni a 24 o 42 processori, quindi una *cpu* per ogni nodo della griglia contro l'SMP completo, che per le prestazioni *intra–cluster*, fortemente evidenti con il cluster *GRID*) ed i tempi di comunicazione sia con messaggi corti (fino a 1024 *bytes*) che lunghi (fino a 64 *Kbytes*) sopra il sistema completo (tipicamente una rete metropolitana).

E' possibile apprezzare come, riducendo al minimo i disturbi (ovvero il traffico sul *link* di rete ed il carico dei singoli cluster), la griglia, seppur considerata piatta, non sembra risentire più di tanto di questa sua limitazione e di altri fattori quali ad esempio il tempo necessario all'autenticazione preliminare (che viene effettuata una sola volta sul *gatekeeper* di ogni cluster) ed all'avvio (`globusrun`) dei job paralleli; è quindi prevedibile che, se si fosse in presenza di un programma sviluppato *"ad hoc"* per *Grid* di questo tipo, il guadagno ottenuto sarebbe ancora più evidente.

C'e da notare infine, che i risultati illustrati in questa sede (relativamente ad **mpptest**, sopra l'intera griglia con messaggi *short* e *long* a protocollo di comunicazione sincrono con *message pattern* di tipo *roundtrip* in assen-

za di bisezione od *overlapping*) sono confrontabili con quelli pubblicati nel sito ufficiale di MPICH-G2 [32], alla sezione '**MPICH-G2 Performance Evalutation**'.

Conclusioni

Negli intenti di questo lavoro, era forte l'esigenza di presentare prima di tutto un approccio efficace ed innovativo, andando a costituire un buon punto di partenza per la realizzazione di ambienti atti ad assolvere funzioni di calcolo ad alte prestazioni a basso costo.
Spero di esserci riuscito.

Ciò che tengo a sottolineare infatti è che l'implementazione dell'ambiente **Grid** realizzato, utilizzando architetture per il calcolo parallelo e distribuito basate su modelli *Open Source* come ad esempio *Beowulf*, rappresenta forse la prima installazione, coerente e funzionante, a livello mondiale, di cui si abbia una evidente traccia; in altre parole, ad oggi non si hanno analoghi *feedback* da parte della comunità scientifica che attestino gli stessi risultati ottenuti.

In ogni caso, la teoria tecnica e l'evoluzione dei rudimenti applicativi presentati in questa sede, ci porta a presupporre che, dopo notevoli rivoluzioni in campo socio-politico o industriale-economico, tutto sembra volgere verso

quella che personalmente definisco, forse in maniera impropria, una possibile *rivoluzione temporale*.

Infatti, rimandando a voi le discussioni circa la bontà dell'approccio presentato in queste pagine piuttosto che altre tecnologie o sistemi che verranno, appare chiaro come ci avviciniamo rapidamente ad un mondo informatico che possa permettere di ottimizzare ciò che, nonostante tutto, rimarrà sempre e comunque una risorsa limitata: il tempo.

In futuro, è possibile ipotizzare la definizioni di particolari strutture (sullo stile dei più classici 'portali' odierni) che permettano di 'noleggiare' o perchè no, vendere tempo di elaborazione.

A questo punto resta solamente aperta la questione di come *re-investire* il tempo risparmiato...

Ma con le applicazioni scientifiche questo non è mai un problema!

Bibliografia

[1] O. Gervasi, A. Laganà, F. Sportolari: *A prototype of a Problem Solving Environment for an a priori Molecular Simulator on the Grid*, Journal of Computational Methods in Sciences and Engineering 2, 1s-2s, 69-75 (2002)

[2] A. Laganà: *METACHEM: Metalaboratories for cooperative innovative computational chemical applications*, METACHEM workshop: `http://www.metacomputing.org/metachem99.html`, Bruxelles, (1999)

[3] I. Foster, C. Kesselman: *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publisher, Inc. (1998)

[4] A. Beguelin, J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam: *A user's guide to PVM Parallel Virtual Machine*, Oak Ridge National Laboratory, Tennessee, (1992)

[5] G. F. Pfister: *In search of Clusters - Second Edition*, Prentice Hall (1998)

[6] M. Davini, I. Lisi: *Cluster - Linux&C n.14*, Piscopo Editore s.r.l. (anno 3)

[7] I. Foster: *What is the Grid? A Three Point Checklist*, Argonne National Laboratory & University of Chicago (2002)

[8] O. Gervasi: *SIMBEX: A metalaboratory for the a priori simulation of Molecular Beam experiments*, EU COST Action D23, Project 003/2001: `http://www.unil.ch/cost/chem/docs/D23/d23-03-01.htm`, (2001)

[9] O. Gervasi, A. Laganà, M. Lobbiani: *Towards a GRID based Portal for an a Priori Molecular Simulation of Chemical Reactivity*, Lecture Notes in Computer Science 2331, 956-967 (2002)

[10] The Condor Project official site: `http://www.cs.wisc.edu/condor/`

[11] A. Laganà, O. Gervasi: *A Distributed Computing Approach to simulate Elementary Reactions*, Minisymposium on Grid Computing, SIMAI 2002, Chia Laguna (CA, Italy) (2002)

[12] M. Smir, S. Otto, S. Huss-Ledermam, D. Walker, J. Dongarra: *MPI: The complete reference*, MIT Press (1996)

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for*

Networked Parallel Computing, MIT Press, Scientific and Engineering Computation, J. Kowalik Editor, Massachusetts Institute of Technology (1994)

[14] The Portable MPI Implementation (MPICH) official site: http://www-unix.mcs.anl.gov/mpi/mpich/

[15] The Globus Project official site: http://www.globus.org/

[16] A. Guarise: *Grid computazionali, soluzione di problemi di calcolo per la fisica di LHC*, Tesi di Laurea – Relatore: Prof. G. Pollarolo – Università degli Studi di Torino (2000)

[17] A. Laganà, O. Gervasi, R. Baraglia, D. Laforenza, Int. J. Quantum Chem.: Quantum Chem. Symp., 28, 85 (1994)

[18] The Globus Project: *GridFTP - Universal Data Transfer for the Grid*, White Paper, University of Chicago and University of Southern California (2000)

[19] The Globus API Documentation: http://www-unix.globus.org/api/c/

[20] S. Tasso: *Introduzione al calcolo parallelo*, Centro d'Ateneo per i Servizi Informatici (C.A.S.I.) - Università di Perugia

[21] T. L. Sterling, J. Salmon, D. J. Becker, Savarese, D. F. Savarese: *How to Build a Beowulf*, MIT Press (1999)

[22] The Beowulf Project official site: `http://www.beowulf.org/`

[23] The Message Passing Interface (MPI) official site: `http://www-unix.mcs.anl.gov/mpi/`

[24] The PVM - Parallel Virtual Machine - official site: `http://www.csm.ornl.gov/pvm/`

[25] L. Amar, A. Barak, A. Eizenberg, A. Shiloh: *The MOSIX Scalable Cluster File Systems for LINUX*, Mosix Project (2000)

[26] The MOSIX Project official site: `http://www.mosix.org/`

[27] F. Sportolari: *SIMGATE: un protocollo di comunicazione in ambiente distribuito*, Tesi di Diploma – Relatori: Prof. O. Gervasi, Prof. A. Laganà – Università degli Studi di Perugia (2001)

[28] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary: *HPL – A Portable Implementation of the High–Performance Linpack Benchmark for Distributed–Memory Computers*, University of Tennessee, Computer Science Department: `http://www.netlib.org/benchmark/hpl/`

[29] The Globus Toolkit Install Guide:
http://www.globus.org/gt2/install/

[30] The LAM-MPI Parallel Computing official site:
http://www.lam-mpi.org/

[31] W. Gropp, E Lusk: *Reproducible Measurement of MPI Performance Characteristics*, Argonne National Laboratory

[32] The MPICH-G2 official site: http://www.niu.edu/mpi/

Appendice A

Le caratteristiche tecniche dei cluster

A.1 GRID

CPU	Due Pentium III Coppermine 1.0 Ghz
RAM	2048 KB
HARD-DISK	IDE ATA100 da 40 GB
NETWORK	NIC Intel e1000 PCI Gigabit Ethernet
	Due NIC Intel e100 pro Fast Ethernet integrate

Tabella 2: **Configurazione hardware dei nodi del cluster GRID.**

A.2 GIZA

CPU	Pentium III ISP1100 800 Mhz
RAM	512 KB
HARD-DISK	IDE da 20 GB
NETWORK	Due NIC Intel e100 pro Fast Ethernet in *channel bonding*

Tabella 3: **Configurazione hardware dei nodi del cluster GIZA.**

A.3 HPC

CPU	Pentium II 400-500 Mhz
RAM	512 KB
HARD-DISK	IDE da 16.5 GB
NETWORK	NIC Intel e100 pro Fast Ethernet

Tabella 4: **Configurazione hardware dei nodi del cluster HPC.**

Appendice B
Il codice del programma glob_env

B-1 File *glob_env.c*

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

extern char **environ;

int main( int argc, char **argv )

{
    int i;
    int rc;
    char *s;
    char *e;
    char *n;
    int l;
    for ( i = 0; environ[i] != NULL; ) {
        e = environ[i];
        s = strchr ( e , '=' );
        l = s - e;
        n = calloc ( 1, l+1 );
        strncpy ( n, e, l );
        printf ( "%s", n );
        i++;
        if ( environ[i] != NULL )
            printf ( "%c" , ',' );
        free ( n );
    }
    printf ( "\n" );
    exit ( 0 );
}
```

Appendice C
L'output del comando grid-info-search

```
# grid−info−search −h gw −x −b "Mds−Vo−name=local,␣o=Grid"
version: 2

#
# filter : ( objectclass=*)
# requesting: ALL
#

# gw, local, grid
dn: Mds−Host−hn=gw,Mds−Vo−name=local,o=grid
objectClass: MdsComputer
objectClass: MdsComputerTotal
objectClass: MdsCpu
objectClass: MdsCpuCache
objectClass: MdsCpuFree
objectClass: MdsCpuSmp
objectClass: MdsCpuTotal
objectClass: MdsCpuTotalFree
objectClass: MdsFsTotal
objectClass: MdsHost
objectClass: MdsMemoryRamTotal
objectClass: MdsMemoryVmTotal
objectClass: MdsNet
objectClass: MdsNetTotal
objectClass: MdsOs
Mds−Computer−isa: IA32
Mds−Computer−platform: i686
Mds−Computer−Total−nodeCount: 1
Mds−Cpu−Cache−l2kB: 256
Mds−Cpu−features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 mmx fxsr sse
Mds−Cpu−Free−15minX100: 100
Mds−Cpu−Free−1minX100: 100
Mds−Cpu−Free−5minX100: 100
Mds−Cpu−model: Pentium III (Coppermine)
Mds−Cpu−Smp−size: 2
Mds−Cpu−speedMHz: 996
Mds−Cpu−Total−count: 2
Mds−Cpu−Total−Free−15minX100: 100
Mds−Cpu−Total−Free−1minX100: 100
Mds−Cpu−Total−Free−5minX100: 100
Mds−Cpu−vendor: GenuineIntel
Mds−Cpu−version: 6.8.10
Mds−Fs−freeMB: 1472
Mds−Fs−freeMB: 2060
```

```
Mds−Fs−freeMB: 2736
Mds−Fs−freeMB: 3774
Mds−Fs−freeMB: 438
Mds−Fs−freeMB: 58
Mds−Fs−freeMB: 8902
Mds−Fs−freeMB: 959
Mds−Fs−sizeMB: 1011
Mds−Fs−sizeMB: 13110
Mds−Fs−sizeMB: 3026
Mds−Fs−sizeMB: 3976
Mds−Fs−sizeMB: 4037
Mds−Fs−sizeMB: 438
Mds−Fs−sizeMB: 68
Mds−Fs−Total−count: 8
Mds−Fs−Total−freeMB: 20399
Mds−Fs−Total−sizeMB: 28692
Mds−Host−hn: gw
Mds−keepto: 20020918072444Z
Mds−Memory−Ram−freeMB: 616
Mds−Memory−Ram−sizeMB: 876
Mds−Memory−Ram−Total−freeMB: 616
Mds−Memory−Ram−Total−sizeMB: 876
Mds−Memory−Vm−freeMB: 2047
Mds−Memory−Vm−sizeMB: 2047
Mds−Memory−Vm−Total−freeMB: 2047
Mds−Memory−Vm−Total−sizeMB: 2047
Mds−Net−addr: 127.0.0.1
Mds−Net−addr: 141.250.240.3
Mds−Net−addr: 141.250.9.124
Mds−Net−name: eth0
Mds−Net−name: eth2
Mds−Net−name: lo
Mds−Net−netaddr: 127.0.0.0/8
Mds−Net−netaddr: 141.250.240.0/24
Mds−Net−netaddr: 141.250.9.0/24
Mds−Net−Total−count: 3
Mds−Os−name: Linux
Mds−Os−release: 2.4.13
Mds−validfrom: 20020918072444Z
Mds−validto: 20020918072444Z

# processors, gw, local , grid
dn: Mds−Device−Group−name=processors, Mds−Host−hn=gw,Mds−Vo−name=local,o=grid
objectClass: MdsCpu
objectClass: MdsCpuSmp
objectClass: MdsCpuTotal
objectClass: MdsCpuCache
objectClass: MdsCpuFree
objectClass: MdsCpuTotalFree
objectClass: MdsDeviceGroup
Mds−Device−Group−name: processors
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z
Mds−Cpu−Cache−l2kB: 256
Mds−Cpu−features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 mmx fxsr sse
Mds−Cpu−Free−15minX100: 100
Mds−Cpu−Free−1minX100: 100
Mds−Cpu−Free−5minX100: 100
Mds−Cpu−model: Pentium III (Coppermine)
Mds−Cpu−Smp−size: 2
Mds−Cpu−speedMHz: 996
Mds−Cpu−Total−count: 2
Mds−Cpu−Total−Free−15minX100: 100
Mds−Cpu−Total−Free−1minX100: 100
Mds−Cpu−Total−Free−5minX100: 100
Mds−Cpu−vendor: GenuineIntel
Mds−Cpu−version: 6.8.10
```

```
# cpu 0, processors, gw, local , grid
dn: Mds−device−name=cpu 0, Mds−Device−Group−name=processors, Mds−Host−hn=gw,Md
 s−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsCpu
objectClass: MdsCpuCache
Mds−Device−name: cpu 0
Mds−Cpu−vendor: GenuineIntel
Mds−Cpu−model: Pentium III (Coppermine)
Mds−Cpu−version: 6.8.10
Mds−Cpu−features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 mmx fxsr sse
Mds−Cpu−speedMHz: 996
Mds−Cpu−Cache−l2kB: 256
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z

# cpu 1, processors, gw, local , grid
dn: Mds−device−name=cpu 1, Mds−Device−Group−name=processors, Mds−Host−hn=gw,Md
 s−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsCpu
objectClass: MdsCpuCache
Mds−Device−name: cpu 1
Mds−Cpu−vendor: GenuineIntel
Mds−Cpu−model: Pentium III (Coppermine)
Mds−Cpu−version: 6.8.10
Mds−Cpu−features: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmo
 v pat pse36 mmx fxsr sse
Mds−Cpu−speedMHz: 996
Mds−Cpu−Cache−l2kB: 256
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z

# memory, gw, local, grid
dn: Mds−Device−Group−name=memory, Mds−Host−hn=gw,Mds−Vo−name=local,o=grid
objectClass: MdsMemoryRamTotal
objectClass: MdsMemoryVmTotal
objectClass: MdsDeviceGroup
Mds−Device−Group−name: memory
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z
Mds−Memory−Ram−Total−sizeMB: 876
Mds−Memory−Ram−Total−freeMB: 617
Mds−Memory−Vm−Total−sizeMB: 2047
Mds−Memory−Vm−Total−freeMB: 2047
Mds−Memory−Ram−sizeMB: 876
Mds−Memory−Ram−freeMB: 617
Mds−Memory−Vm−sizeMB: 2047
Mds−Memory−Vm−freeMB: 2047

# physical memory, memory, gw, local, grid
dn: Mds−Device−name=physical memory, Mds−Device−Group−name=memory, Mds−Host−hn
 =gw,Mds−Vo−name=local,o=grid
objectClass: Mds
objectClass: MdsDevice
objectClass: MdsMemoryRam
Mds−Device−name: physical memory
Mds−Memory−Ram−sizeMB: 876
Mds−Memory−Ram−freeMB: 617
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z

# virtual memory, memory, gw, local, grid
dn: Mds−Device−name=virtual memory, Mds−Device−Group−name=memory, Mds−Host−hn=
 gw,Mds−Vo−name=local,o=grid
```

```
objectClass: Mds
objectClass: MdsDevice
objectClass: MdsMemoryVm
Mds−Device−name: virtual memory
Mds−Memory−Vm−sizeMB: 2047
Mds−Memory−Vm−freeMB: 2047
Mds−validfrom: 20020918072445Z
Mds−validto: 20020918072545Z
Mds−keepto: 20020918072545Z

# filesystems, gw, local , grid
dn: Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,Mds−Vo−name=local,o=grid
objectClass: MdsFsTotal
objectClass: MdsDeviceGroup
Mds−Device−Group−name: filesystems
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z
Mds−Fs−freeMB: 1472
Mds−Fs−freeMB: 2060
Mds−Fs−freeMB: 2736
Mds−Fs−freeMB: 3774
Mds−Fs−freeMB: 438
Mds−Fs−freeMB: 58
Mds−Fs−freeMB: 8902
Mds−Fs−freeMB: 959
Mds−Fs−sizeMB: 1011
Mds−Fs−sizeMB: 13110
Mds−Fs−sizeMB: 3026
Mds−Fs−sizeMB: 3976
Mds−Fs−sizeMB: 4037
Mds−Fs−sizeMB: 438
Mds−Fs−sizeMB: 68
Mds−Fs−Total−count: 8
Mds−Fs−Total−freeMB: 20399
Mds−Fs−Total−sizeMB: 28692

# /, filesystems , gw, local , grid
dn: Mds−Device−name=/, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,Mds−V
 o−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /
Mds−Fs−sizeMB: 3026
Mds−Fs−freeMB: 2736
Mds−Fs−mount: /
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /boot, filesystems, gw, local , grid
dn: Mds−Device−name=/boot, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,M
 ds−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /boot
Mds−Fs−sizeMB: 68
Mds−Fs−freeMB: 58
Mds−Fs−mount: /boot
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /home, filesystems, gw, local , grid
dn: Mds−Device−name=/home, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,M
 ds−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /home
Mds−Fs−sizeMB: 13110
```

```
Mds−Fs−freeMB: 8902
Mds−Fs−mount: /home
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /dev/shm, filesystems, gw, local, grid
dn: Mds−Device−name=/dev/shm, Mds−Device−Group−name=filesystems, Mds−Host−hn=g
 w,Mds−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /dev/shm
Mds−Fs−sizeMB: 438
Mds−Fs−freeMB: 438
Mds−Fs−mount: /dev/shm
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /tmp, filesystems, gw, local, grid
dn: Mds−Device−name=/tmp, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,Md
 s−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /tmp
Mds−Fs−sizeMB: 1011
Mds−Fs−freeMB: 959
Mds−Fs−mount: /tmp
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /usr, filesystems, gw, local, grid
dn: Mds−Device−name=/usr, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,Md
 s−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /usr
Mds−Fs−sizeMB: 3026
Mds−Fs−freeMB: 1472
Mds−Fs−mount: /usr
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /usr/local, filesystems, gw, local, grid
dn: Mds−Device−name=/usr/local, Mds−Device−Group−name=filesystems, Mds−Host−hn
 =gw,Mds−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /usr/local
Mds−Fs−sizeMB: 4037
Mds−Fs−freeMB: 2060
Mds−Fs−mount: /usr/local
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z

# /work, filesystems, gw, local, grid
dn: Mds−Device−name=/work, Mds−Device−Group−name=filesystems, Mds−Host−hn=gw,M
 ds−Vo−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsFs
Mds−Device−name: /work
Mds−Fs−sizeMB: 3976
Mds−Fs−freeMB: 3774
Mds−Fs−mount: /work
Mds−validfrom: 20020918071126Z
Mds−validto: 20020918072626Z
Mds−keepto: 20020918072626Z
```

```
# networks, gw, local, grid
dn: Mds−Device−Group−name=networks, Mds−Host−hn=gw,Mds−Vo−name=local,o=grid
objectClass: MdsNetTotal
objectClass: MdsNet
objectClass: MdsDeviceGroup
Mds−Device−Group−name: networks
Mds−validfrom: 20020918071127Z
Mds−validto: 20020918072627Z
Mds−keepto: 20020918072627Z
Mds−Net−addr: 127.0.0.1
Mds−Net−addr: 141.250.240.3
Mds−Net−addr: 141.250.9.124
Mds−Net−name: eth0
Mds−Net−name: eth2
Mds−Net−name: lo
Mds−Net−netaddr: 127.0.0.0/8
Mds−Net−netaddr: 141.250.240.0/24
Mds−Net−netaddr: 141.250.9.0/24
Mds−Net−Total−count: 3

# eth0, networks, gw, local, grid
dn: Mds−Device−name=eth0, Mds−Device−Group−name=networks, Mds−Host−hn=gw,Mds−V
 o−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds−Device−name: eth0
Mds−Net−name: eth0
Mds−Net−netaddr: 141.250.9.0/24
Mds−Net−addr: 141.250.9.124
Mds−validfrom: 20020918071127Z
Mds−validto: 20020918072627Z
Mds−keepto: 20020918072627Z

# eth2, networks, gw, local, grid
dn: Mds−Device−name=eth2, Mds−Device−Group−name=networks, Mds−Host−hn=gw,Mds−V
 o−name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds−Device−name: eth2
Mds−Net−name: eth2
Mds−Net−netaddr: 141.250.240.0/24
Mds−Net−addr: 141.250.240.3
Mds−validfrom: 20020918071127Z
Mds−validto: 20020918072627Z
Mds−keepto: 20020918072627Z

# lo, networks, gw, local, grid
dn: Mds−Device−name=lo, Mds−Device−Group−name=networks, Mds−Host−hn=gw,Mds−Vo−
 name=local,o=grid
objectClass: MdsDevice
objectClass: MdsNet
Mds−Device−name: lo
Mds−Net−name: lo
Mds−Net−netaddr: 127.0.0.0/8
Mds−Net−addr: 127.0.0.1
Mds−validfrom: 20020918071127Z
Mds−validto: 20020918072627Z
Mds−keepto: 20020918072627Z

# operating system, gw, local, grid
dn: Mds−Software−deployment=operating system, Mds−Host−hn=gw,Mds−Vo−name=local
 ,o=grid
objectClass: MdsSoftware
objectClass: MdsOs
Mds−Software−deployment: operating system
Mds−Os−name: Linux
Mds−Os−release: 2.4.13
Mds−validfrom: 20020918071127Z
Mds−validto: 20020918072627Z
```

```
Mds−keepto: 20020918072627Z

# jobmanager−condor, gw.grid.unipg.it, local, grid
dn: Mds−Software−deployment=jobmanager−condor, Mds−Host−hn=gw.grid.unipg.it,Md
 s−Vo−name=local,o=grid
objectClass: Mds
objectClass: MdsSoftware
objectClass: MdsService
objectClass: MdsServiceGram
objectClass: MdsComputer
objectClass: MdsOs
Mds−Software−deployment: jobmanager−condor
Mds−Service−type: x−gram
Mds−Service−hn: gw.grid.unipg.it
Mds−Service−port: 2119
Mds−Service−url: x−gram://gw.grid.unipg.it:2119/jobmanager−condor:/O=Grid/O=Gl
 obus/CN=gw.grid.unipg.it
Mds−Service−protocol: 0.1
Mds−Computer−isa: i686
Mds−Os−release: 2.4.13
Mds−Os−name: Linux
Mds−Computer−manufacturer: pc
Mds−Service−Gram−schedulertype: condor
Mds−validfrom: 200209180724.45Z
Mds−validto: 200209180725.15Z
Mds−keepto: 200209180725.15Z

# unknown−unknown, jobmanager−condor, gw.grid.unipg.it, local, grid
dn: Mds−Job−Queue−name=unknown−unknown, Mds−Software−deployment=jobmanager−con
 dor, Mds−Host−hn=gw.grid.unipg.it,Mds−Vo−name=local,o=grid
objectClass: Mds
objectClass: MdsSoftware
objectClass: MdsJobQueue
objectClass: MdsComputerTotal
objectClass: MdsComputerTotalFree
objectClass: MdsGramJobQueue
Mds−Job−Queue−name: unknown−unknown
Mds−Computer−Total−nodeCount: 0
Mds−Computer−Total−Free−nodeCount: 0
Mds−Memory−Ram−Total−sizeMB: 0
Mds−Memory−Ram−sizeMB: 0
Mds−Gram−Job−Queue−maxtime: 0
Mds−Gram−Job−Queue−maxcputime: 0
Mds−Gram−Job−Queue−maxcount: 0
Mds−Gram−Job−Queue−maxrunningjobs: 0
Mds−Gram−Job−Queue−maxjobsinqueue: 0
Mds−Gram−Job−Queue−whenactive: 0
Mds−Gram−Job−Queue−status: 0
Mds−Gram−Job−Queue−dispatchtype: batch
Mds−Gram−Job−Queue−priority: NULL
Mds−Gram−Job−Queue−jobwait: NULL
Mds−Gram−Job−Queue−schedulerSpecific: NULL
Mds−validfrom: 200209180724.45Z
Mds−validto: 200209180725.15Z
Mds−keepto: 200209180725.15Z

# jobmanager, gw.grid.unipg.it, local, grid
dn: Mds−Software−deployment=jobmanager, Mds−Host−hn=gw.grid.unipg.it,Mds−Vo−na
 me=local,o=grid
objectClass: Mds
objectClass: MdsSoftware
objectClass: MdsService
objectClass: MdsServiceGram
objectClass: MdsComputer
objectClass: MdsOs
Mds−Software−deployment: jobmanager
Mds−Service−type: x−gram
Mds−Service−hn: gw.grid.unipg.it
Mds−Service−port: 2119
Mds−Service−url: x−gram://gw.grid.unipg.it:2119/jobmanager:/O=Grid/O=Globus/CN
```

```
 =gw.grid.unipg.it
Mds-Service-protocol: 0.1
Mds-Computer-isa: i686
Mds-Os-release: 2.4.13
Mds-Os-name: Linux
Mds-Computer-manufacturer: pc
Mds-Service-Gram-schedulertype: fork
Mds-validfrom: 200209180724.45Z
Mds-validto: 200209180725.15Z
Mds-keepto: 200209180725.15Z

# default, jobmanager, gw.grid.unipg.it, local, grid
dn: Mds-Job-Queue-name=default, Mds-Software-deployment=jobmanager, Mds-Host-h
 n=gw.grid.unipg.it,Mds-Vo-name=local,o=grid
objectClass: Mds
objectClass: MdsSoftware
objectClass: MdsJobQueue
objectClass: MdsComputerTotal
objectClass: MdsComputerTotalFree
objectClass: MdsGramJobQueue
Mds-Job-Queue-name: default
Mds-Computer-Total-nodeCount: 2
Mds-Computer-Total-Free-nodeCount: 1
Mds-Memory-Ram-Total-sizeMB: 0
Mds-Memory-Ram-sizeMB: 0
Mds-Gram-Job-Queue-maxtime: 0
Mds-Gram-Job-Queue-maxcputime: 0
Mds-Gram-Job-Queue-maxcount: 0
Mds-Gram-Job-Queue-maxrunningjobs: 0
Mds-Gram-Job-Queue-maxjobsinqueue: 0
Mds-Gram-Job-Queue-whenactive: 0
Mds-Gram-Job-Queue-status: 0
Mds-Gram-Job-Queue-dispatchtype: Immediate
Mds-Gram-Job-Queue-priority: NULL
Mds-Gram-Job-Queue-jobwait: NULL
Mds-Gram-Job-Queue-schedulerSpecific: NULL
Mds-validfrom: 200209180724.45Z
Mds-validto: 200209180725.15Z
Mds-keepto: 200209180725.15Z

# local, Grid
dn: Mds-Vo-name=local,o=Grid
objectClass: GlobusStub

# search result
search: 2
result: 0 Success

# numResponses: 27
# numEntries: 26
```

Appendice D
MPICH-G2: Il codice dei programmi d'esempio

D-1 File *ring.c*

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

/* command line configurables */
int Ntrips;  /* -t <ntrips> */
int Verbose; /* -v */

int parse_command_line_args(int argc, char **argv, int my_id)

{
    int i;
    int error;

    /* default values */
    Ntrips = 1;
    Verbose = 0;

    for (i = 1, error = 0; !error && i < argc; i ++)
    {
        if (!strcmp(argv[i], "-t"))
        {
            if (i + 1 < argc && (Ntrips = atoi(argv[i+1])) > 0)
                i ++;
            else
                error = 1;
        }
        else if (!strcmp(argv[i], "-v"))
            Verbose = 1;
        else
            error = 1;
    } /* endfor */
    if (error && !my_id)
    {
        /* only Master prints usage message */
        fprintf(stderr, "\n\tusage: %s {-t <ntrips>} {-v}\n\n", argv[0]);
        fprintf(stderr, "where\n\n");
        fprintf(stderr,
            "\t-t <ntrips>\t- Number of trips around the ring.  "
            "Default value 1.\n");
```

```
        fprintf(stderr,
            "\t-v\t\t- Verbose.  Master and all slaves log each step. \n");
        fprintf(stderr, "\t\t\t  Default value is FALSE.\n\n");
    } /* endif */
    return error;
} /* end parse_command_line_args() */

main(int argc, char **argv)

{
    int numprocs, my_id, passed_num;
    int trip;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    if (parse_command_line_args(argc, argv, my_id))
    {
        MPI_Finalize();
        exit(1);
    } /* endif */
    if (Verbose)
        printf("my_id %d numprocs %d\n", my_id, numprocs);
    if (numprocs > 1)
    {
        if (my_id == 0)
        {
            /* I am the Master */
            passed_num = 0;
            for (trip = 1; trip <= Ntrips; trip++)
            {
                passed_num++;
                if (Verbose)
                    printf("Master: starting trip %d of %d: "
                        "before sending num=%d to dest=%d\n",
                        trip, Ntrips, passed_num, 1);

                MPI_Send(&passed_num, /* buff   */
                        1,            /* count */
                        MPI_INT,      /* type  */
                        1,            /* dest  */
                        0,            /* tag   */
                        MPI_COMM_WORLD); /* comm */

                if (Verbose)
                    printf("Master: inside trip %d of %d: "
                        "before receiving from source=%d\n",
                        trip, Ntrips, numprocs-1);

                MPI_Recv(&passed_num, /* buff   */
                        1,            /* count  */
                        MPI_INT,      /* type   */
                        numprocs-1,   /* source */
                        0,            /* tag    */
                        MPI_COMM_WORLD, /* comm */
                        &status);     /* status */

                printf("Master: end of trip %d of %d: "
                    "after receiving passed_num=%d "
                    "(should be =trip*numprocs=%d) from source=%d\n",
                    trip, Ntrips, passed_num, trip*numprocs, numprocs-1);
            } /* endfor */
        }
        else
        {
            /* I am a Slave */
            for (trip = 1; trip <= Ntrips; trip++)
            {
                if (Verbose)
                    printf("Slave %d: top of trip %d of %d: "
```

```
                    "before_receiving_from_source=%d\n",
                    my_id, trip, Ntrips, my_id-1);

            MPI_Recv(&passed_num, /* buff    */
                    1,                /* count  */
                    MPI_INT,          /* type   */
                    my_id-1,          /* source */
                    0,                /* tag    */
                    MPI_COMM_WORLD, /* comm */
                    &status);         /* status */

            if (Verbose)
                printf("Slave %d: inside trip %d of %d: "
                    "after receiving passed_num=%d from source=%d\n",
                    my_id, trip, Ntrips, passed_num, my_id-1);
            passed_num ++;
            if (Verbose)
                printf("Slave %d: inside trip %d of %d: "
                    "before sending passed_num=%d to dest=%d\n",
                    my_id, trip, Ntrips, passed_num, (my_id+1)%numprocs);

            MPI_Send(&passed_num,     /* buff   */
                    1,                    /* count */
                    MPI_INT,              /* type  */
                    (my_id+1)%numprocs, /* dest */
                    0,                    /* tag   */
                    MPI_COMM_WORLD); /* comm */

            if (Verbose)
                printf("Slave %d: bottom of trip %d of %d: "
                    "after send to dest=%d\n",
                    my_id, trip, Ntrips, (my_id+1)%numprocs);
        } /* endfor */
      } /* endif */
    }
    else
      printf("numprocs = %d, should be run with numprocs > 1\n", numprocs);
    MPI_Finalize();
} /* end main() */
```

D-2 File *report_colors.c*

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

void print_topology(int me, int size , int *depths, int **colors)

{
    int i , j , max = 0;
    FILE *fp;
    char fname[100];
    sprintf(fname, "colors.%d", me);
    if (!( fp = fopen(fname, "w")))
    {
        fprintf(stderr , "ERROR: could not open fname >%s<\n", fname);
        MPI_Abort(MPI_COMM_WORLD, 1);
    } /* endif */
    fprintf(fp , "proc\t");
    for (i = 0; i < size ; i++)
        fprintf(fp , "% 3d", i);
    fprintf(fp , "\nDepths\t");
    for (i = 0; i < size ; i++)
    {
        fprintf(fp , "% 3d", depths[i]);
        if ( max < depths[i] )
            max = depths[i];
    } /* endfor */
    for (j = 0; j < max; j++)
    {
        fprintf(fp , "\nlvl %d\t", j);
        for (i = 0; i < size ; i++)
            if ( j < depths[i] )
                fprintf(fp , "% 3d", colors[i ][ j ]);
            else
                fprintf(fp , "   ");
    } /* endfor */
    fprintf(fp , "\n");
    fclose(fp);
    return;
} /* end print_topology() */

int main (int argc, char *argv[])

{
    int me, nprocs, flag , rv;
    int *depths;
    int **colors;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    rv = MPI_Attr_get(MPI_COMM_WORLD, MPICHX_TOPOLOGY_DEPTHS, &depths, &flag);
    if ( rv != MPI_SUCCESS )
    {
        printf("MPI_Attr_get(depths) failed, aborting\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    } /* endif */
    if ( flag == 0 )
    {
        printf("MPI_Attr_get(depths): depths not available...\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    } /* endif */
    rv = MPI_Attr_get(MPI_COMM_WORLD, MPICHX_TOPOLOGY_COLORS, &colors, &flag);
    if ( rv != MPI_SUCCESS )
    {
        printf("MPI_Attr_get(colors) failed, aborting\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    } /* endif */
```

```
    if ( flag == 0 )
    {
        printf("MPI_Attr_get(colors): depths not available...\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    } /* endif */
    print_topology(me, nprocs, depths, colors);
    MPI_Finalize();
    return 0;
} /* end main() */
```

Appendice E
GRAM error codes list

000 one of the RSL parameters is not supported
001 the RSL lentgh is greater than the maximum allowed
002 of an unused NO_RESOURCES
003 jobmanager unable to set **default** to the directory requested
004 the executable does not exist
005 of an unused INSUFFICIENT_FUNDS
006 authentication with the remote server failed
007 of an unused USER_CANCELLED
008 of an unused SYSTEM_CANCELLED
009 data transfer to the server failed
010 the stdin file does not exist
011 the connection to the server failed (check host and port)
012 the provided RSL 'maxtime' value is not an integer
013 the provided RSL 'count' value is not an integer
014 the job manager received an invalid RSL
015 the job manager failed in allowing others to make contact
016 the job failed when the job manager attempted to run it
017 an invalid paradyn was specified
018 the provided RSL 'jobtype' value is invalid
019 the provided RSL 'myjob' value is invalid
020 the job manager failed to locate an internal script argument file
021 the job manager failed to create an internal script argument file
022 the job manager detected an invalid job state
023 the job manager detected an invalid script response
024 the job manager detected an invalid job state
025 the provided RSL 'jobtype' value is not supported by this job manager
026 unused ERROR_UNIMPLEMENTED
027 the job manager failed to create an internal script submission file
028 the job manager cannot find the user proxy
029 the job manager failed to open the user proxy
030 the job manager failed to cancel the job as requested
031 system memory allocation failed
032 the interprocess job communication initialization failed
033 the interprocess job communication setup failed
034 the provided RSL 'host count' value is invalid
035 one of the provided RSL parameters is unsupported
036 the provided RSL 'queue' parameter is invalid
037 the provided RSL 'project' parameter is invalid
038 the provided RSL string includes variables that could not be identified
039 the provided RSL 'environment' parameter is invalid
040 the provided RSL 'dryrun' parameter is invalid
041 the provided RSL is invalid (an empty string)
042 the job manager failed to stage the executable
043 the job manager failed to stage the stdin file
044 the requested job manager type is invalid
045 the provided RSL 'arguments' parameter is invalid
046 the gatekeeper failed to run the job manager
047 the provided RSL could not be properly parsed
048 there is a version mismatch between GRAM components
049 the provided RSL 'arguments' parameter is invalid

050 the provided RSL 'count' parameter is invalid
051 the provided RSL 'directory' parameter is invalid
052 the provided RSL 'dryrun' parameter is invalid
053 the provided RSL 'environment' parameter is invalid
054 the provided RSL 'executable' parameter is invalid
055 the provided RSL 'host_count' parameter is invalid
056 the provided RSL 'jobtype' parameter is invalid
057 the provided RSL 'maxtime' parameter is invalid
058 the provided RSL 'myjob' parameter is invalid
059 the provided RSL 'paradyn' parameter is invalid
060 the provided RSL 'project' parameter is invalid
061 the provided RSL 'queue' parameter is invalid
062 the provided RSL 'stderr' parameter is invalid
063 the provided RSL 'stdin' parameter is invalid
064 the provided RSL 'stdout' parameter is invalid
065 the job manager failed to locate an internal script
066 the job manager failed on the system call pipe()
067 the job manager failed on the system call fcntl()
068 the job manager failed to create the temporary stdout filename
069 the job manager failed to create the temporary stderr filename
070 the job manager failed on the system call fork()
071 the executable file permissions **do** not allow execution
072 the job manager failed to open stdout
073 the job manager failed to open stderr
074 the cache file could not be opened in order to relocate the user proxy
075 cannot access cache files in $HOME/.globus/.gass_cache, check permissions, quota, and disk space
076 the job manager failed to insert the contact in the client contact list
077 the contact was not found in the job manager's client contact list
078 connecting to the job manager failed. Possible reasons: job terminated, invalid job contact, network problems, ...
079 the syntax of the job contact is invalid
080 the executable parameter in the RSL is undefined
081 the job manager service is misconfigured. condor arch undefined
082 the job manager service is misconfigured. condor os undefined
083 the provided RSL 'min_memory' parameter is invalid
084 the provided RSL 'max_memory' parameter is invalid
085 the RSL 'min_memory' value is not zero or greater
086 the RSL 'max_memory' value is not zero or greater
087 the creation of a HTTP message failed
088 parsing incoming HTTP message failed
089 the packing of information into a HTTP message failed
090 an incoming HTTP message did not contain the expected information
091 the job manager does not support the service that the client requested
092 the gatekeeper failed to find the requested service
093 the jobmanager does not accept any new requests (shutting down)
094 the client failed to close the listener associated with the callback URL
095 the gatekeeper contact cannot be parsed
096 the job manager could not find the 'poe' command
097 the job manager could not find the 'mpirun' command
098 the provided RSL 'start_time' parameter is invalid
099 the provided RSL 'reservation_handle' parameter is invalid
100 the provided RSL 'max_wall_time' parameter is invalid
101 the RSL 'max_wall_time' value is not zero or greater
102 the provided RSL 'max_cpu_time' parameter is invalid
103 the RSL 'max_cpu_time' value is not zero or greater
104 the job manager is misconfigured, a scheduler script is missing
105 the job manager is misconfigured, a scheduler script has invalid permissions
106 the job manager failed to signal the job
107 the job manager did not recognize/support the signal type
108 the job manager failed to get the job id from the local scheduler
109 the job manager is waiting **for** a commit signal
110 the job manager timed out **while** waiting **for** a commit signal
111 the provided RSL 'save_state' parameter is invalid
112 the provided RSL 'restart' parameter is invalid
113 the provided RSL 'two_phase' parameter is invalid
114 the RSL 'two_phase' value is not zero or greater
115 the provided RSL 'stdout_position' parameter is invalid
116 the RSL 'stdout_position' value is not zero or greater
117 the provided RSL 'stderr_position' parameter is invalid
118 the RSL 'stderr_position' value is not zero or greater
119 the job manager restart attempt failed

```
120 the job state file doesn't exist
121 could not read the job state file
122 could not write the job state file
123 old job manager is still alive
124 job manager state file TTL expired
125 it is unknown if the job was submitted
126 the provided RSL 'remote_io_url' parameter is invalid
127 could not write the remote io url file
128 the standard output/error size is different
129 the job manager was sent a stop signal (job is still running)
130 the user proxy expired (job is still running)
131 the job was not submitted by original jobmanager
132 the job manager is not waiting for that commit signal
```