ENEA GRID

Seminario avanzato per l'utente di cresco

Autore: Alessandro Secco
alessandro.secco@nice-italy.com

Seminario avanzato per l'utente di Cresco

- Riepilogo utilizzo di base di LSF
- LSF: job paralleli avanzati
- LSF: i flussi di job
- LSF: gruppi di job
- Riferimenti

Riepilogo utilizzo di base di LSF

Riepilogo utilizzo di base di LSF

- Nei seminari precedenti sono state visitate tutte le principali opzioni di lancio di un job tramite LSF:
  - Lancio di un job seriale
  - Lancio di un job seriale multi-case
    - Code seriali:
      - **cresco_serh48** e **cresco_serh72**
  - Lancio di un job parallelo openmpi
  - Lancio di un job parallelo mvapich
    - Code parallele:
      - **cresco_512h2**, **cresco_512h8**, **cresco_512h24** (> 512 processori)
      - **cresco_128h2**, **cresco_128h8**, **cresco_128h24** (> 128 e <= 512 processori)
      - **cresco_16h2**, **cresco_16h8**, **cresco_16h24** (> 16 e <= 128 processori)
      - **cresco_h2**, **cresco_h8**, **cresco_h24** (> 8 e <= 16 processori)

Riepilogo utilizzo di base di LSF

- I seminari hanno dimostrato la compilazione tramite mpicc di un semplice job mpi ed il lancio tramite LSF

```
/* hello.c
*
* Simple "Hello World" program in MPI.
* NOTA: Programma di test preso in prestito dal sito del CERN
*/
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
int numprocs; /* Number of processors */
int procnum; /* Processor number */
/* Initialize MPI */
MPI_Init(&argc, &argv);
/* Find this processor number */
MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
/* Find the number of processors */
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
printf ("Hello world! from processor %d out of %d\n", procnum, numprocs);
/* Shut down MPI */
MPI_Finalize();
return 0;
}
```

Riepilogo utilizzo di base di LSF

- Script generico di lancio per job basati su openmpi:

```
#!/bin/sh

PROCS=`cat $LSB_DJOB_HOSTFILE | wc -l`

mpirun -n $PROCS --mca pls_rsh_agent "blaunch.sh" –hostfile $LSB_DJOB_HOSTFILE ./a.out
```

- Script generico di lancio per job basati su mvapich:

```
#!/bin/sh

PROCS=`cat $LSB_DJOB_HOSTFILE | wc -l`

mpirun_rsh -rsh -np $PROCS -hostfile $LSB_DJOB_HOSTFILE ./a.out
```

- Sottomissione del job:

```
bsub -q cresco_128h2 -n 200 -o test.%J ./lancio.sh
```

LSF: job paralleli avanzati

Regole generali

- Nei primi mesi di cresco, LSF è stato integrato con molti codici
- Nello script di lancio, le regole sono:
  - Sostituire “ssh” con “blaunch.sh”:
    - Con mvapich basta usare **mpirun_rsh -rsh**
    - Con openmpi bisogna mettere in testa a $PATH **/afs/enea.it/software/bin/support/ssh**, o usare l'opzione di mpirun **–mca pls_rsh_agent “blaunch.sh”** (se possibile)
  - Contare i processori passati da LSF
    - PROCS=`cat $LSB_DJOB_HOSTFILE | wc -l`
  - Usare **$LSB_DJOB_HOSTFILE** come parametro da passare a “hostfile” o “machinefile”.

Caso 1: processi eterogenei

- Il caso è stato proposto da un utente che aveva la necessità di lanciare 3 processi diversi all'interno dello stesso job mpi.
- I 3 processi avevano finalità diverse, ma comunicavano tra loro in mpi:
  - I processi andavano lanciati tutti all'interno dello stesso job di LSF
  - Si doveva trovare un modo per distribuire i processi sui nodi
- La soluzione proposta (attualmente in fase di sperimentazione) è basata su uno script.

Caso 1: processi eterogenei (openmpi)

```
#!/bin/sh

PROC1=1
NAME1=./a.out
PROC2=40
NAME2=./b.out
PROC3=15
NAME3=./c.out
PROCS=56

PATH=/afs/enea.it/software/bin/support/ssh:$PATH

cat $LSB_DJOB_HOSTFILE | head -$PROC1 > h1
cat $LSB_DJOB_HOSTFILE | head -`expr $PROC2 + $PROC1` | tail -$PROC2 > h2
cat $LSB_DJOB_HOSTFILE | tail -$PROC3 > h3

rm -f launchfile
for i in `cat ./h1`
do
      echo "-np 1 --host $i $NAME1" >> launchfile
done
for i in `cat ./h2`
do
      echo "-np 1 --host $i $NAME2" >> launchfile
done
for i in `cat ./h3`
do
      echo "-np 1 --host $i $NAME3" >> launchfile
done

mpirun -app ./launchfile
```

Assegnazione processi e processori

PATH per blaunch.sh

Divisione dei processori

Creazione di una riga nel file di lancio per ogni processo

Lancio del job

Caso 2: fluent

- Fluent è un codice commerciale di fluido dinamica
- La sua integrazione con il mondo cresco è stata semplice: è bastato seguire la mini-guida.

```
#!/bin/sh

PATH=/afs/enea.it/software/bin/support/ssh:$PATH
export PATH

PROCS=`cat $LSB_DJOB_HOSTFILE | wc -l`
fluent63 -t$PROCS -cnf=$LSB_DJOB_HOSTFILE $*
```

Altri casi

- Altri codici sono stati integrati:
  - OpenFOAM
  - CPMD
  - MCNPX
  - ...
- In linea di massima non cambia nulla rispetto a quanto detto nelle regole generali, ma è necessario contattare gli amministratori per avere dettagli
- Si vedano i riferimenti per integrare nuovi codici o utilizzare integrazioni già fatte.

LSF: I flussi di job

LSF: i flussi di job

- Forse pochi utenti sanno che lsf può gestire i flussi di job.
- I flussi sono utili in molti ambiti:
  - Job dipendente dal file di output di un altro
  - Job che collezioni l'output di diversi job precedenti
  - Sequenze di job automatizzate (loop)
- Per pianificare un flusso di job, è necessario conoscere un po' di scripting, alcune nozioni di LSF ed un minimo di progettazione di algoritmi.

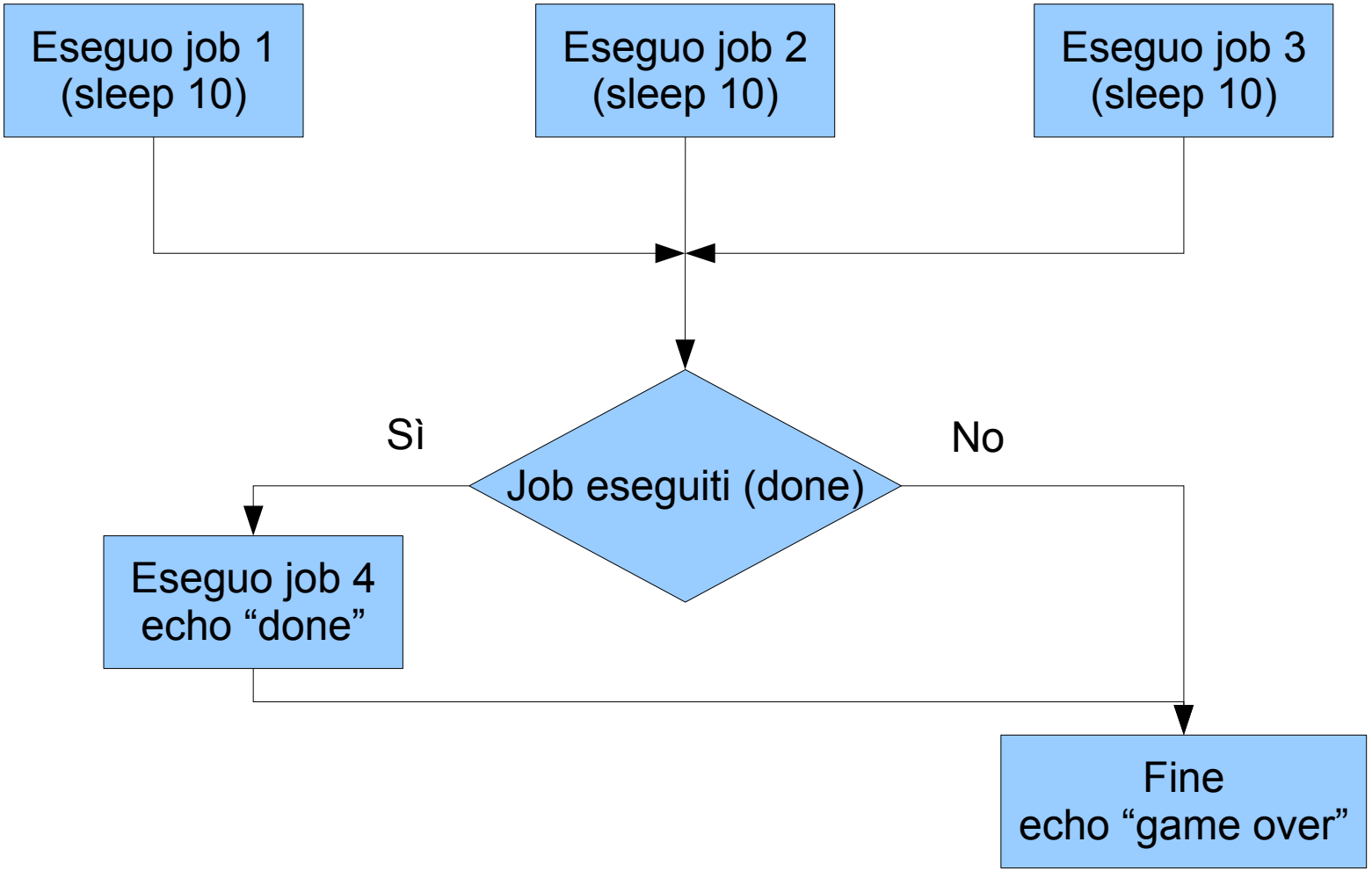
Un utile trucco

- In molti casi servirà avere esattamente il jobid come output di bsub:
  - bsub ... | awk -F'<' '{print $2}' | awk -F'>' '{print $1}'
- Si potrebbe mettere il filtro in uno script dentro la propria home directory (jobid.sh):

```
#!/bin/sh

awk -F'<' '{print $2}' | awk -F'>' '{print $1}'
```

Esercizio: lancio di 3 job ed attesa



Esercizio: lancio di 3 job ed attesa

sub.sh:

```
#!/bin/sh
JOBID1=`bsub sleep 10 | ./jobid.sh`
JOBID2=`bsub sleep 10 | ./jobid.sh`
JOBID3=`bsub sleep 10 | ./jobid.sh`
export JOBID1 JOBID2 JOBID3

DEPEND1=`echo "done($JOBID1) && done($JOBID2) && done($JOBID3)"`
JOBID4=`bsub -w "$DEPEND1" -o done ./done.sh | ./jobid.sh`
export JOBID4

DEPEND2=`echo "exit($JOBID1) || exit($JOBID2) || exit($JOBID3) ||
ended($JOBID4)"`
JOBID5=`bsub -w "$DEPEND2" -o clean ./clean.sh`
```

done.sh:

```
#!/bin/sh
echo “$JOBID1, $JOBID2, $JOBID3 done”
```

clean.sh:

```
bkill $JOBID1 $JOBID2 $JOBID3 $JOBID4 >/dev/null 2>&1
echo “game over”
```

Esercizio: utilizzo in pratica

- L'esercizio potrebbe essere utile quando
  - è necessario aspettare la fine di una serie di job per collezionare gli output tramite uno script
  - se i job non vanno a buon fine, lanciare uno script di pulizia
- Questo esempio, unito alla funzionalità di job array, o lancio di job paralleli, permette di creare veri e propri algoritmi di lancio dei job
  - Un job in attesa potrebbe, al suo interno, lanciare un nuovo bsub, creando una condizione simile ad un ciclo “while”
- **ATTENZIONE**: la durata del flusso non può andare oltre la durata del token di AFS.

Gruppi di job

Gruppi di job

- In alcune circostanze, può essere importante raggruppare i propri job personali
  - Finalità di controllo.
- In altri casi, potrebbe essere interessante gestire i job del proprio gruppo di lavoro:
  - Gestione centralizzata di gruppi di utenti
  - Gestione gerarchica dei gruppi
- L'uso dei gruppi è utile se si vuole richiedere all'amministratore un “Service Level Agreement”, un accordo per ottenere un certo livello di servizio.

Aggiunta di un gruppo

- Un gruppo può essere creato in 2 modi:
  - bsub -g /nome <nomejob> (se “/nome” è inesistente)
  - bgadd /nome
- L'utente che aggiunge il gruppo ne diventa automaticamente l'amministratore
- Il gruppo viene cancellato in automatico alla fine dei job nel primo caso. Altrimenti:
  - bgdel /nome
- L'opzione -L di bgadd permette di definire un massimo numero di job contemporaneamente in stato “run” per gruppo.

Uso del gruppo e gestione

- Per usare il gruppo si lancia:
  - bsub -g /nome <nomejob>
  - La variabile d'ambiente LSB_DEFAULT_JOBGROUP lancia su un gruppo in automatico
- L'utente amministratore del gruppo, avrà possibilità di gestire tutti i job di tutti gli utenti del proprio gruppo:
  - bkill, bstop, bresume, bmod, bswitch
- Questo è utile per non dover chiamare l'amministratore se un utente lancia un gran numero di job e si deve fermarli quando esso non è presente.

Gerarchia tra gruppi

- All'interno dello stesso gruppo, è possibile definire sottogruppi.
  - bgadd /nome/sottonome
    - L'amministratore di /nome sarà l'amministratore anche di /nome/sottonome
    - Il creatore di /nome/sottonome non è necessariamente il creatore di /nome, e potrà amministrare solo i job del suo sottogruppo.
- Per vedere tutti i gruppi definiti e sapere chi li amministra esiste il comando bjgroup che dà l'elenco di tutti i gruppi e tutti gli amministratori.

Considerazioni sui gruppi

- I gruppi sono gestiti in modo autonomo dagli utenti.
- Ogni utente può decidere di creare gruppi o sottogruppi in modo indipendente dagli amministratori
  - Questo è importante nel caso, per esempio, in cui un utente voglia poter agire da “amministratore locale” all'interno dei job dei propri collaboratori.
  - Una volta organizzato il proprio gruppo, l'utente può contrattare dei servizi speciali con l'amministratore.
- I gruppi non hanno finalità di accounting. L'accounting di gruppo può essere collezionato dall'opzione -J (progetto) di bsub o tramite l'uso di Service Level Agreement.

Applicazione pratica

- Enea è divisa in macro-gruppi (fus, erg, fis, ...)
- All'interno dei gruppi si possono definire dei sottogruppi, ognuno dei quali ha sue finalità e può avere un suo amministratore
- L'amministratore di gruppo può chiedere alla gestione di cresco delle condizioni particolari per soddisfare i propri requisiti di servizio (SLA):
  - Deadline
  - Velocità (maggior numero di job contemporanei)
  - Throughput (maggior numero di job finiti)

Riferimenti

Riferimenti

- http://gridticket.enea.it: permette di aprire “ticket” di supporto. E' il modo consigliato per segnalare problemi inerenti ENEA grid.
- griduser@enea.it: è possibile contattare tutti gli utenti per raccogliere esperienze sull'uso della GRID.
- alessandro.secco@nice-italy.com: problemi relativi a questo corso possono essere indirizzati a me. La collaborazioni di tutti è gradita.