PARALLEL HIGH
PERFORMANCE COMPUTING
CdS magistrale in informatica

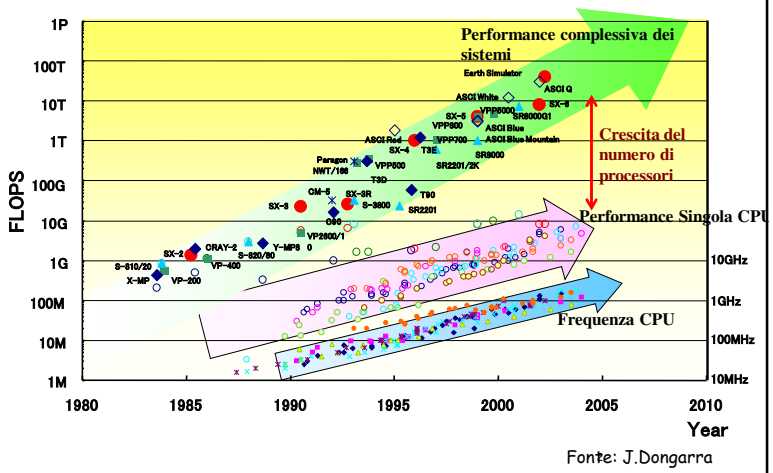
13 – La fault tolerance

Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

wpage.unina.it/lapegna

1

Crescita del numero di processori



Fonte: J.Dongarra

2

Crescita del numero di processori

Anni 80: un sistema con
64 processori → sistema di grandi dimensioni

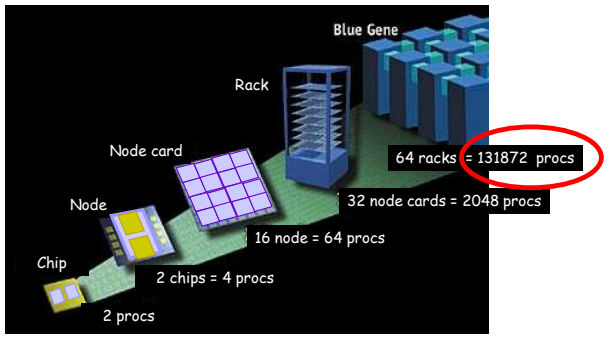
Anni 90: un sistema con
64 processori → sistema di medie dimensioni
1024 processori → sistema di grandi dimensioni

Anni 2000: un sistema con
64 processori → sistema di piccole dimensioni
1024 processori → sistema di medie dimensioni
32000 processori → sistema di grandi dimensioni

Anni 2010:
sistemi con 100000 processori (ad es. IBM Blue Jene)

3

IBM Blue Gene



Totale =131872 processori
(2004: 32768 procs)

4

MTTF (mean time to failure)

Tempo medio per un guasto: e' il tempo (medio) di funzionamento di un processore

- E' una misura dell'affidabilita' delle componenti hardware
- Un valore comune per il MTTF e': ~ 10^5 ore per processore (circa 11 anni)

Ogni quanto tempo si guastera'
un processore dell' IBM Blue Gene?

5

Guasto dell'IBM Blue Gene

IBM Blue Gene = 131872 processori

MTTF per uno dei 131872 processori
=
10^5 / 131872 = 0.75 ore = 45 min. !!

Ci si deve aspettare che (mediamente)
ogni 45 minuti potrebbe rendersi indisponibile
1 processore dell' IBM Blue Gene

6

La fault tolerance

Esigenze
- Crescita del numero di processori
- Molte applicazioni sono sviluppate per esecuzioni lunghe giorni

Opportunita'
- in ambienti distribuiti le risorse sono indipendenti. Se una diventa non disponibile, le altre non ne risentono

Le applicazioni distribuite possono (e devono)
essere tolleranti ai guasti (fault tolerant)

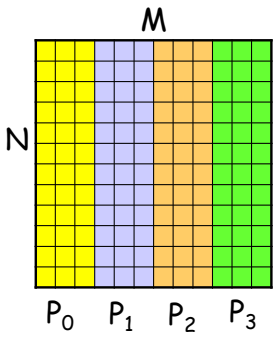
Un algoritmo e' detto

Fault Tolerant

Se e' in grado di continuare correttamente l'esecuzione anche se uno dei nodi si rende indisponibile

7

esempio



- matrice N x M
- p=4 processi
- matrice distribuita per colonne

- calcolare

$$\max_{i=0,N-1} \sum_{j=0}^{M-1} |a_{ij}|$$

Massimo sulle righe

Somma sulle colonne

8

Possibile algoritmo parallelo

```
P =  numero di processi
Myid = mio identificativo

jstart = Myid*M/P
jend = (Myid+1)*M/P -1

max = 0
for i = 0, N-1 do
        sumloc = 0
        for j = jstart, jend
                sumloc = sumloc + |a(i,j)|
        endfor
        globalsum (sumloc, sumglob )

        if (sumglob > max ) then
                max = sumglob
        endif
endfor
```

Il ciclo parte sempre da 0

Somma sulle colonne locali ad ogni proc

Somma sui processi

Massimo sulle righe

problema

Se uno dei processi cade

Il valore di sumloc per quel processo non e' disponibile → impossibile calcolare sumglob

Inoltre l'algoritmo non puo' continuare e tutto il lavoro eseguito fino a quel punto non e' piu' recuperabile

Bisogna ricominciare daccapo

soluzione

Il contenuto delle memorie di massa persiste al guasto dei processi

- Salvare periodicamente sul disco lo stato del calcolo (checkpoint)
- In caso di guasto, far ripartire l'applicazione dall'ultimo checkpoint (ripristino)
  - Sostituire il processo caduto oppure
  - Continuare con un processo in meno

Fault tolerance a livello dell'algoritmo

Fault tolerance = checkpoint + ripristino

QUANDO e COSA
salvare sul disco?

QUANDO: ad esempio ogni 10 righe

COSA:
- il valore del massimo (la variabile max)
- la riga a cui si e' arrivati (la variabile i)

In caso di errore, il calcolo puo' riprendere dall'ultimo checkpoint

Esempio: ripristino con sostituzione

```
P =  numero di processi;
Myid = mio identificativo
i = 0;   max = 0;

if (restart == true) then
        restoredata (isaved, maxsaved)
        i = isaved;   max = maxsaved;
endif

jstart = Myid*M/P
jend =     (Myid+1)*M/P -1
```

Ripristino dopo una failure:
Ogni proc legge i dati salvati in un checkpoint precedente

NB: La prima volta si esegue con restart = false

Cont….

Esempio: (cont.)

```
while ( i  <=  N-1 ) do

        sumloc = 0
        for j = jstart, jend
                sumloc = sumloc + |a(i,j)|
        endfor

        k = numero di proc indisponibili
        if ( k != 0) then
                procedura di ripristino
        endif

        globalsum (sumloc, sumglob )
        if (sumglob > max ) then
                max = sumglob
        endif

        if   (i/10*10 == i)   savedata(i, max)

        i=i+1
endwhile
```

Il ciclo non parte sempre da 0

Fase di ripristino — *Vedi dopo*

Fase di checkpoint

Procedura di ripristino

```
scegli tra i sopravvissuti un 'master' (ad es. quello con Myid minimo)

if (Myid == master){
        Manda in esecuzione K nuovi processi con restart = TRUE
}

Restoredata (isaved, maxsaved)
i = isaved
max = maxsaved

sumloc = 0
for j = jstart, jend
        sumloc = sumloc + |a(i,j)|
endfor
```

Sostituzione processi non piu' disponibili

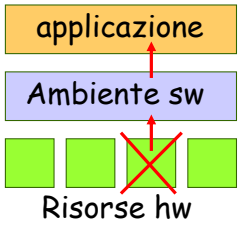
Anche i proc sopravvissuti ripristinano lo stato dell'ultimo checkpoint

Quindi

La gestione della fault tolerance coinvolge 2 fasi specifiche

- checkpoint periodico
- ripristino in caso di risorse di calcolo non piu' disponibili

osservazione

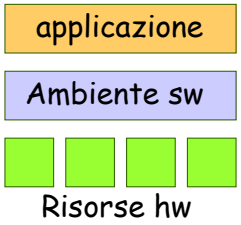


Individuazione di risorse non piu' disponibili

Ruolo dell' ambiente software

L'ambiente software

L'ambiente software media tra le risorse hw e l'applicazione



comandi tipo:
- determina il numero di processi sopravvissuti…
- Manda in esecuzione nuovi processi …

dipendono dall'ambiente di calcolo utilizzato

Ambienti sw Fault Tolerance

Un ambiente sw e' detto

Fault Tolerant

Se e' in grado di comunicare all'applicazione un cambiamento dello stato delle risorse e gestire la dinamicita' delle stesse

Quindi…

FAULT TOLERANCE

Individuazione dello stato delle risorse → A carico dell' ambiente sw

Checkpoint e ripristino dei dati → A carico dell' applicazione

Non tutti gli ambienti sw sono fault tolerant !!
(ad esempio alcune implementazioni di MPI non lo sono !!)

Ambienti sw Fault Tolerant

Non tutti gli ambienti sw sono fault tolerant
(ad esempio lo MPI non lo e' !!)

Implementazioni (non standard) di MPI per renderlo FT
- FT-MPI (icl.cs.utk.edu/ftmpi)
- OPEN-MPI

Ambienti sw fault tolerant:
- PVM

Osservazione:

Nell'esempio mostrato, ogni proc effettua il checkpoint solo sul proprio disco

Ma cio' non e' sempre possibile.
(se il processore non ha il disco, o non si hanno i permessi di scrittura)

Ne' sempre opportuno
(se il processore diventa indisponibile, anche il disco lo diventa)

E' necesario un altro disco per
il checkpoint di tutti i processi

problema

Il tempo di accesso alle memorie di massa e' dell'ordine di 10^{-3}sec.

Tale tempo va moltiplicato per il numero di dati da salvare/recuperare

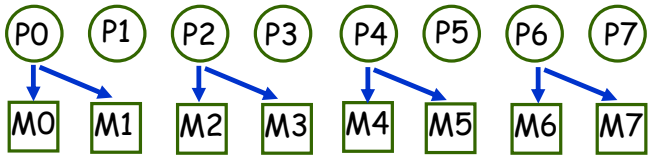
Tale tempo va inoltre moltiplicato per il numero di processi che deve effettuare il checkpoint/restart

In un sistema con 10^5 processori tale fase potrebbe durare ore!!

Una soluzione:

Utilizzare processi in piu' rispetto a quelli utilizzati per il calcolo, per effettuare le fasi di checkpoint/ripristino
(diskless checkpoint)

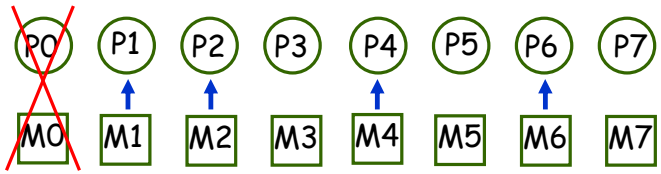
Esempio 1: mirroring - checkpoint



Ogni processo ha un proprio processo per il checkpoint

Ogni processo effettua il checkpoint oltre che nella propria memoria, anche in quella del processo di checkpoint

Esempio 1: mirroring - ripristino



Se il processo P0 cade, i dati possono essere recuperati dalla memoria M1 e l'applicazione puo' ripartire dall'ultimo checkpoint

Il processo di checkpoint P1 sostituisce "naturalmente" P0

Esempio 1: mirroring

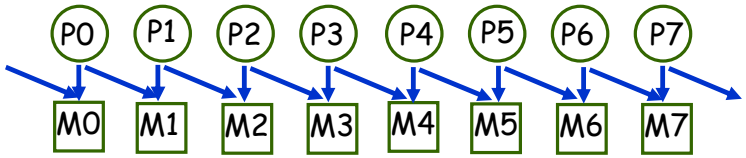
Per ogni processo dedicato al calcolo, esiste un processo dedicato al checkpoint

Detto n il numero complessivo di proc, possono essere tollerati fino a n/2 guasti
(a patto che non si guastino contemporaneamente i due processi della coppia)

VANTAGGIO: processi gia' pronti per la sostituzione

SVANTAGGIO: solo n/2 proc sono dedicati al calcolo

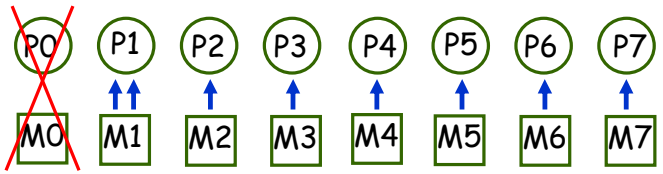
Esempio 2: ring neighbor - checkpoint



Tutti i processi sono dedicati al calcolo e organizzati "ad anello"

Ogni processo effettua il checkpoint oltre che nella propria memoria, anche in quella del processo vicino

Esempio 2: ring neighbor - ripristino



Se il processo P0 cade, i dati possono essere recuperati dalla memoria M1 e l'applicazione puo' ripartire dall'ultimo checkpoint

P1 si fa carico del lavoro di P0
(oppure si fa partire un nuovo processo)

Esempio 2: ring neighbor

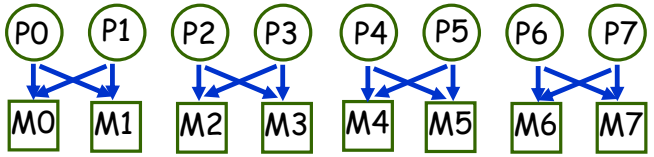
Tutti i processi sono dedicati al calcolo

Detto n il numero complessivo di proc, possono essere tollerati fino a n/2 guasti
(a patto che non si guastino contemporaneamente due proc vicini)

VANTAGGIO: non ci sono risorse inutilizzate

SVANTAGGIO: sovraccarico di alcuni processi o necessita' di nuovi processi in caso di ripristino dei dati

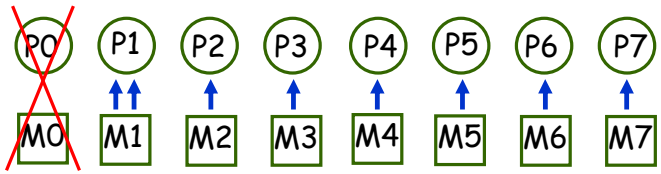
Esempio 3: pair neighbor - checkpoint



Tutti i processi sono dedicati al calcolo e organizzati "a coppie"

Ogni processo effettua il checkpoint oltre che nella propria memoria, anche in quella del processore della stessa coppia

Esempio 3: pair neighbor - ripristino



Se il processo P0 cade, i dati possono essere recuperati dalla memoria M1 e l'applicazione puo' ripartire dall'ultimo checkpoint

P1 si fa carico del lavoro di P0
(oppure si fa partire un nuovo processo)

Esempio 3: pair neighbor

Tutti i processi sono dedicati al calcolo

Detto n il numero complessivo di proc, possono essere tollerati fino a n/2 guasti
(a patto che non si guastino contemporaneamente due proc della stessa coppia)

VANTAGGIO: non ci sono risorse inutilizzate e probabilita' inferiore rispetto al ring neighbor di un blocco dell'applicazione (es. guasto dei processi P1 e P2)

SVANTAGGIO: sovraccarico di alcuni processi o necessita' di nuovi processi

Neighbor based checkpoint

I tre schemi descritti appartengono alla classe dei neighbor based checkpoint

1. Ogni proc effettua il checkpoint nella propria memoria e in quella di un suo processo vicino
2. Se un nodo fallisce, i dati possono essere recuperati da tale processo
3. Assenza di operazioni globali per il checkpoint/recupero → basso overhead

Problema

Gli schemi neighbor-based per il checkpoint conservano lo stato di tutti i processi

Se i dati sono tanti e i processi numerosi, tali schemi possono richiedere molti Tbytes di memoria solo per il checkpoint

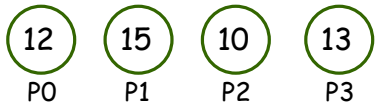
Necessita' di schemi alternativi

Idea

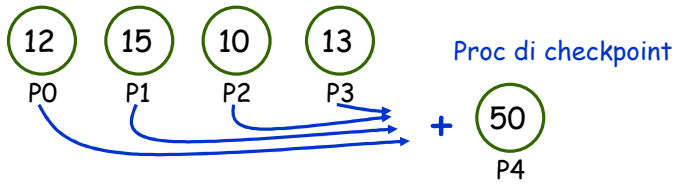
"compattare" i dati dei checkpoint di tutti i processi in un solo dato da conservare in un solo processo per il checkpoint

Esempio: 4 processi di calcolo e 1 per il checkpoint

Si vuole effettuare il checkpoint dei seguenti dati nei 4 processi



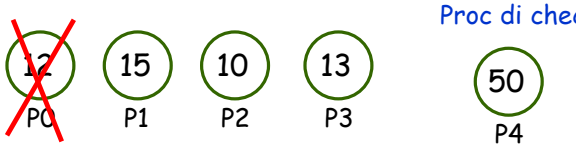
Esempio 1: codifica di base - checkpoint



Nella memoria del processore di checkpoint viene conservata la somma dei dati di cui si vuole il checkpoint

Esempio 1: codifica di base - ripristino

Se un processo cade



Come recuperare il dato di P0?

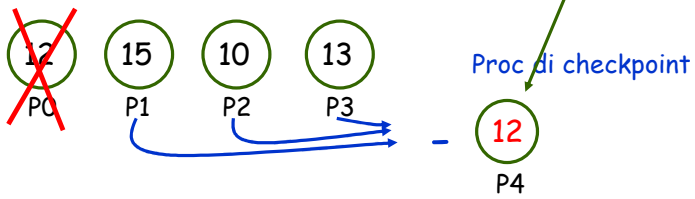
$$X0 + 15 + 10 + 13 = 50$$

Equazione in una incognita

37

Esempio 1: codifica di base - ripristino

$$X0 = 50 - (15 + 10 + 13) = 12$$



Il calcolo puo' continuare con i processi P1, P2, P3 e P4

38

Esempio 1: codifica di base

Presenza di 1 processore di checkpoint in cui compattare i dati di checkpoint di tutti i processi

VANTAGGIO: richiesta poca memoria per il checkpoint e processo gia' disponibile per la sostituzione

SVANTAGGIO: Puo' essere tollerato solo 1 guasto e errori di round-off nel ripristino

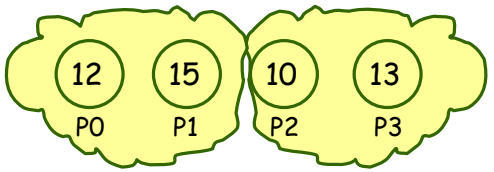
39

Esempio 2: codifica a gruppi

La limitazione della codifica di base puo' essere superata facilmente utilizzando m processi di checkpoint

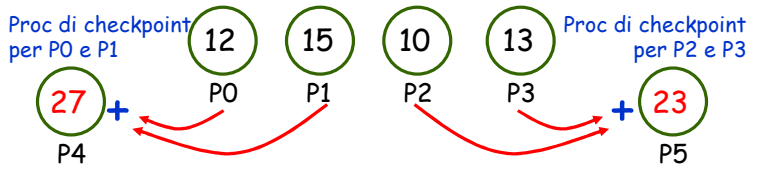
Esempio: 4 processi di calcolo e 2 di checkpoint

Si possono raggruppare i processi a gruppi di 2



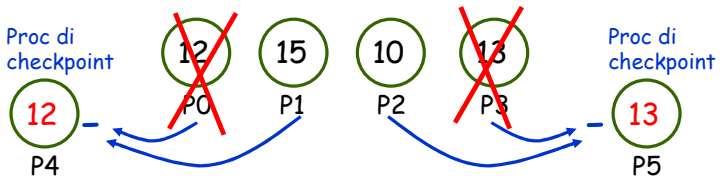
40

Esempio 2: codifica a gruppi: checkpoint



Nella memoria dei processori di checkpoint vengono conservate le somme dei dati di un gruppo di processi

Esempio 2: codifica a gruppi: ripristino



$$X0 = 27 - 15 = 12 \qquad X3 = 23 - 10 = 13$$

Se 2 processi cadono i dati possono essere recuperati in maniera analoga alla codifica di base
Il calcolo continua con P1, P2, P4 e P5

Esempio 2: codifica a gruppi

Presenza di piu' processori di checkpoint in cui conservare i dati di tutti i processi (1 per gruppo)

VANTAGGIO:
- richiesta poca memoria per i checkpoint
- processi gia' disponibili per la sostituzione
- possono essere tollerati fino a m guasti
- possibilita' di codifiche in parallelo

SVANTAGGIO:
- puo essere tollerato 1 solo fallimento per ogni gruppo di processi

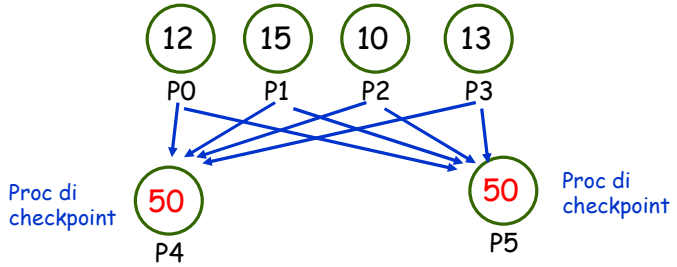
problema

E' possibile rimuovere questa ultima limitazione?

Rimuoviamo la condizione che ci sia 1 processo di checkpoint in ogni gruppo

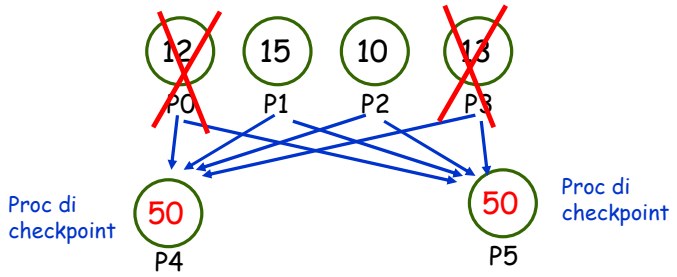
Esempio:
Si consideri il caso di n=4 processi di calcolo e m=2 processi per il checkpoint

Esempio 3:



I processi di checkpoint P4 e P5 sono al servizio di tutti i processi di calcolo

Se 2 processi cadono



P4 $\Rightarrow$ $X0+15+10+X3=50$

P5 $\Rightarrow$ $X0+15+10+X3=50$

Problema

$$\begin{cases} X0+X3=25 \\ X0+X3=25 \end{cases}$$

Sistema di 2 equazioni (uguali) in 2 incognite *X0* e *X3*

Sistema indeterminato

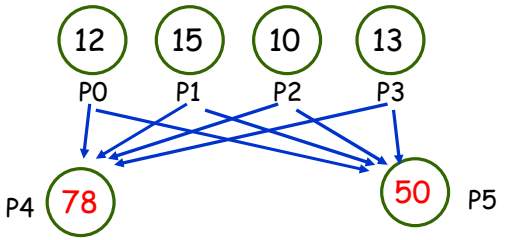
Il problema e' mal posto perche' la matrice del sistema e' singolare

Introdurre dei coefficienti ai dati nella fase di checkpoint (matrice di checkpoint)

Esempio 4: codifica pesata - checkpoint

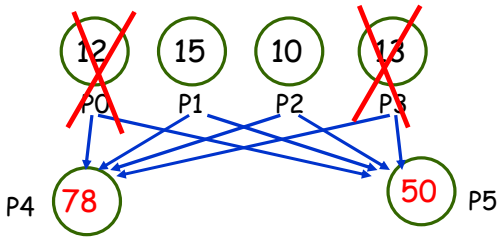
$$A=\begin{pmatrix} 1 & 2 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} \Rightarrow \begin{cases} 1\cdot 12+2\cdot 15+1\cdot 10+2\cdot 13=78 \\ 1\cdot 12+1\cdot 15+1\cdot 10+1\cdot 13=50 \end{cases}$$

Matrice di checkpoint *Dati codificati*



Esempio 4: codifica pesata - ripristino

Se 2 proc cadono



$$\begin{cases} 1 \cdot X0 + 2 \cdot X3 = 78 - 2 \cdot 15 - 1 \cdot 10 = 38 \\ 1 \cdot X0 + 1 \cdot X3 = 50 - 1 \cdot 15 - 1 \cdot 10 = 25 \end{cases}$$

Sistema compatibile di 2 equazioni in 2 incognite

$X0 = 12 \qquad X3 = 13$

Problema:

Se invece dei processi P0 e P3 cadono i processi P0 e P2 che succede?

$$\begin{cases} 1 \cdot X0 + 1 \cdot X2 = 78 - 2 \cdot 15 - 2 \cdot 13 = 22 \\ 1 \cdot X0 + 1 \cdot X2 = 50 - 1 \cdot 15 - 1 \cdot 10 = 25 \end{cases}$$

Problema mal posto

Quindi, per n=4 e m=2

La matrice di checkpoint deve essere tale che tutte le sottomatrici di ordine 2 devono essere non singolari

ESEMPIO

$$A = \begin{pmatrix} 1 & 2 & 1 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

NO !!

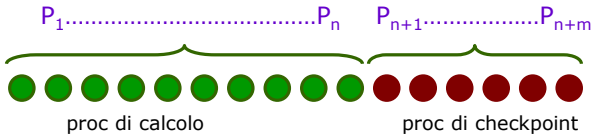
$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

SI !!

Codifica pesata - in generale

Siano
- n processi di calcolo con dati $D_1,\ldots, D_n$
- m processi di checkpoint

n >> m



Codifica pesata - checkpoint

$$A = \begin{pmatrix} a_{11} & \cdots & \cdots & a_{1n} \\ \vdots & & & \vdots \\ a_{m1} & \cdots & \cdots & a_{mn} \end{pmatrix}$$

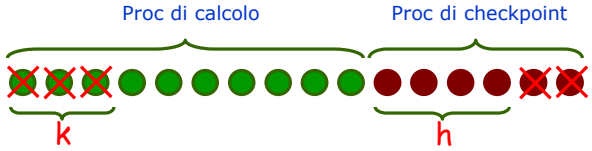
Matrice di checkpoint con m righe e n colonne

I valori dei checkpoint $C_1,..,C_m$ sono calcolati come

$$\begin{cases} a_{11}D_1 + \cdots + a_{1n}D_n = C_1 \\ a_{m1}D_1 + \cdots + a_{mn}D_n = C_m \end{cases}$$

Supponiamo che (per semplicita')

- cadono (i primi) k processi di calcolo
- sopravvivono (i primi) h processi di checkpoint



Nel precedente prodotto
$D_1,..,D_k$ e $C_{h+1},..,C_n$
diventano incognite

Il sistema diventa

$$\begin{cases} a_{11}D_1 + \cdots + a_{1k}D_k = C_1 - \sum_{i=k+1}^{n} a_{1i}D_i \\ a_{h1}D_1 + \cdots + a_{hk}D_k = C_h - \sum_{i=k+1}^{n} a_{hi}D_i \end{cases}$$

Se k > h il sistema ha piu' incognite che equazioni
(non e' possibile ripristinare i dati)

Codifica pesata - ripristino

$$\begin{cases} a_{11}D_1 + \cdots + a_{1k}D_k = C_1 - \sum_{i=k+1}^{n} a_{1i}D_i \\ a_{h1}D_1 + \cdots + a_{hk}D_k = C_h - \sum_{i=k+1}^{n} a_{hi}D_i \end{cases}$$

Se $h \geq k$ il sistema ha piu' equazioni che incognite
(scegliendo le prime k equazioni si puo'
risolvere il sistema e ripristinare i dati)

Sistema di ripristino

Come scegliere le equazioni

Per garantire compatibilita' al sistema,
la matrice dei coefficienti del sistema ottenuto deve essere non singolare

Il minore fondamentale A_k di ordine k della matrice di checkpoint A deve essere non singolare

$A=$ [A_k]

Osservazione

Nell'esempio abbiamo supposto che i primi k processi di calcolo cadono e i primi h processi di checkpoint sopravvivono

In generale bisogna supporre che un gruppo di k processi possono cadere e che qualsiasi gruppo di h processi di checkpoint sopravvivono

⇓

Qualsiasi minore di ordine k
deve essere non singolare

Inoltre, il ripristino dei dati,
con lo schema della codifica pesata,
richiede la risoluzione di un sistema di equazioni

⇓

errore di r.o. elevato se qualche minore ha indice di condizionamento elevato

Riassumendo..

Siamo alla ricerca di una
matrice di checkpoint A tale che:

- tutti i minori di ordine k
  siano non singolari (per ogni k)
- i relativi sistemi di ripristino
  siano ben condizionati

Matrici casuali gaussiane:

Una matrice di numeri reali casuali A di m righe e n colonne, si dice Gaussiana se gli elementi sono variabili casuali indipendenti normalmente distribuite
con media=0 e dev.st.=1

⇓

(numeri con valore assoluto piccolo sono piu' probabili)

Esempio: m=4, n=5

$$A = \begin{pmatrix} -0.04 & -1.87 & 0.57 & -0.25 & -0.23 \\ 0 & 0.42 & 0.04 & -0.37 & 0.11 \\ -0.31 & 0.89 & 0.67 & -0.29 & 0.31 \\ 1.09 & 0.73 & 0.56 & -1.47 & 1.44 \end{pmatrix}$$

60

Si dimostra che:

Data una matrice casuale Gaussiana A di m righe e n colonne, si ha:

1. la matrice A ha (in probabilita') un basso indice di condizionamento (e quindi e' non singolare)
2. ogni minore di A di ordine k e' ancora una matrice casuale Gaussiana (per ogni k)

Ogni minore di A di ordine k e'
una matrice non singolare
ben condizionata

61

Esempio 5: codifica pesata a gruppi

I due schemi di
- codifica a gruppi
- codifica pesata

possono essere combinati insieme

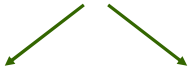
Unire i vantaggi di

- codifica a gruppi (parallelismo)
- codifica pesata (flessibilita')

62

Riassumendo...

diskless
checkpoint



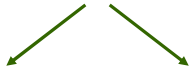
neighbor based
(conserva i dati di checkpoint)
- mirroring
- neighbor ring
- neighbor pair

checksum based
(codifica i dati di checkpoint)
- checksum di base
- checksum a gruppi
- checksum pesato
- checksum pesato a gruppi

63

Riassumendo...

diskless
checkpoint



neighbor based
(conserva i dati di checkpoint)
- molta memoria
- assenza di errori di r.o.
- piu' rigido
- richiede nuovi processi

checksum based
(codifica i dati di checkpoint)
- poca memoria
- errori di r.o.
- piu' flessibile
- processi pronti

64