DOTTORATO DI RICERCA

in

SCIENZE COMPUTAZIONALI E INFORMATICHE

Ciclo XXI

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

SEDE AMMINISTRATIVA: UNIVERSITÀ DI NAPOLI FEDERICO II

Vania Boccia

Realizzazione di un sistema di *checkpointing/migration*, in ambiente distribuito, per la fault tolerance di applicazioni parallele basate su PETSc.

TESI DI DOTTORATO DI RICERCA

IL COORDINATORE
Prof. Luigi M. Ricciardi

Premessa

È ormai noto che alle due metodologie classiche di indagine scientifica, quella teorica e quella sperimentale, si è aggiunta quella *computazionale*.

Le simulazioni al calcolatore consentono, infatti, di analizzare fenomeni e sistemi complessi, troppo costosi o pericolosi per essere realizzati sperimentalmente, e il loro utilizzo sta pervadendo ogni campo scientifico e tecnologico. La necessità di livelli di dettaglio sempre maggiori e tempi di risposta "adeguati", richiede la realizzazione e l'utilizzo efficace di strumenti di calcolo (hardware e software) sempre più potenti. L'effettivo utilizzo delle prestazioni dei sistemi di calcolo, disponibili per alcune classi di problemi, è in continua evoluzione: le prestazioni massime di oggi possono diventare prestazioni standard nel futuro; ciò che è importante è che, in qualsiasi istante nel tempo, esisterà sempre un insieme di problemi per cui è necessario fornire soluzioni nel più breve tempo possibile, e, a tale scopo, bisogna individuare nuovi strumenti di calcolo (hardware e software) adeguati. Il metro di giudizio non è più la riduzione del tempo ma ciò che viene denominato *time-to-solution*, che include i tempi per la trasformazione del problema in algoritmi opportuni, per la realizzazione del software basato su tali algoritmi, per l'ottimizzazione di tale software nell'ambiente computazionale di riferimento, per l'esecuzione del software, per l'analisi dei risultati e per l'avanzamento della conoscenza a partire da tale analisi.

Negli ultimi decenni l'attenzione si è estesa dai sistemi di calcolo parallelo a quelli di calcolo distribuito ad alte prestazioni, costituiti da insiemi di cluster geograficamente dislocati, in grado di fornire un sistemi scalabili di potenza sempre crescente. Tali sistemi possono essere utilizzati per risolvere i cosiddetti problemi complessi.

La realizzazione di software scientifico che sia affidabile ed efficiente, in ambienti di calcolo distribuito, richiede l'individuazione e l'analisi dei problemi connessi alla progettazione e all'implementazione di algoritmi per architetture ad alte prestazioni e la loro integrazione in un'architettura software distribuita. Ovviamente, durante l'implementazione, nascono difficoltà legate all'eterogeneità delle risorse coinvolte nel calcolo e alla dinamicità di un ambiente in cui in maniera inattesa, possono variare l'efficienza e la raggiungibilità delle risorse (sovraccarichi o guasti dei nodi di calcolo, della rete di interconnessione, ...)

Nello scenario descritto, l'esigenza di prevedere, da un lato, meccanismi che abilitino il software a "sopravvivere" a tali eventi inattesi garantendo dall'altro sistemi che consentano di usufruire, in maniera efficiente, dell'ambiente di calcolo. (ad es. migrando il software su risorse alternative, rispetto a quelle guaste, garantendo comunque il mantenimento di un certo livello di prestazioni).

Un software per ambiente distribuito deve essere progettato in modo da **adattarsi** alle continue variazioni dell'ambiente che lo esegue. Per adattarsi deve essere in grado di **capire** che qualcosa è cambiato durante la sua esecuzione. Inoltre, quando si scrive un'applicazione distribuita, devono essere prese in considerazione sia le caratteristiche dell'ambiente di esecuzione, sia le richieste dell'applicazione stessa. In generale però devono essere garantiti sia il successo dell'esecuzione sia le sue prestazioni.

Da molti anni ormai si lavora, in contesti di ricerca internazionali, al fine di individuare le metodologie standard per risolvere il problema della tolleranza ai guasti e dell'efficienza dei software progettati per ambienti distribuiti.

Di qui l'idea di intraprendere il presente lavoro di ricerca con il duplice obiettivo di:

- individuare una metodologia per l'implementazione robusta ed efficiente di meccanismi per rilevare e gestire i suddetti guasti all'interno di algoritmi e librerie per il calcolo scientifico, sviluppati per ambienti ad alte prestazioni;
- implementare tale metodologia mediante un casus studii.

"Un giorno le macchine
riusciranno a risolvere tutti i problemi,
ma mai nessuna di esse potrà porne uno."

Albert Einstein

Indice

Introduzione

La realizzazione di algoritmi e software scientifici ad alte prestazioni, che risultino affidabili ed efficienti in contesti come il calcolo distribuito, richiede l'individuazione e la risoluzione di alcuni problemi, alcuni dei quali ereditati dagli ambienti di calcolo parallelo massivo. Questi ultimi, infatti, sono caratterizzati dall'utilizzo di un elevato numero di nodi di calcolo e quindi da un'alta percentuale che si verifichino guasti durante l'esecuzione delle applicazioni. Altri problemi nascono, invece, dalla dinamicità e dall'eterogeneità delle risorse dei sistemi distribuiti, la cui disponibilità ed efficienza può variare nel tempo, sia a causa di guasti della rete, di alcuni nodi di calcolo o di interi siti, sia a causa di variazioni del carico di lavoro. Nello scenario descritto, nasce l'esigenza di prevedere, da un lato, meccanismi che abilitino il software a "sopravvivere" ad eventuali guasti (rilevando e gestendo il guasto) e dall'altro sistemi che consentano di usufruire, in maniera efficiente, dell'ambiente di calcolo (ad es., garantendo il mantenimento di un certo livello di prestazioni). In generale, un software si dice fault tolerant, in grado cioè di "sopravvivere ai guasti", se è in grado di rilevare il guasto, gestire il guasto (terminando la sua esecuzione in modo corretto) e riprendere "automaticamente" la sua esecuzione dal punto in cui questa si era interrotta a causa del guasto. Quanto appena affermato evidenzia due livelli di gestione dei fault: uno a carico dell'ambiente e l'altro a carico dell'applicazione. La tolleranza ai guasti garantisce la terminazione corretta dell'applicazione anche in caso di situazioni inattese; tuttavia da sola non basta a garantire la non perdita del lavoro svolto fino al verificarsi del guasto. Per garantire che il lavoro già effettuato non venga perso è necessario combinare ai meccanismi di fault tolerance, anche tecniche di checkpointing, che consentono all'applicazione di "congelare" periodicamente il suo stato di esecuzione e, a seguito di un guasto, di riprendere l'applicazione da dove era stata interrotta. Applicazioni progettate per essere eseguite su sistemi paralleli massivi sono, per loro natura, particolarmente esigenti in termini di prestazioni. In generale ci si aspetta di portare a termine su una certa risorsa (con precise caratteristiche HW/SW) il problema dato in un dato tempo. Tuttavia, l'ambiente di esecuzione che si considera in questo lavoro (sistemi distribuiti) è per sua natura soggetto a cambiamenti abbastanza inattesi: può accadere per esempio che l'esecuzione di un'applicazione venga compromessa, non da un guasto vero e proprio ma da un improvviso sovraccarico di una o più risorse utilizzate, e quindi da un relativo calo prestazionale.

Il caso ottimale è rappresentato, dunque, da un sistema in cui l'ambiente di esecuzione è in grado di rilevare un guasto o un calo di prestazioni, mentre l'applicazione, salvando periodicamente il suo stato, è in grado di fermare la sua esecuzione, e riprenderla (eventualmente su risorse di calcolo alternative) senza perdere il lavoro svolto fino al momento del fault.

In generale, le applicazioni scientifiche per ambienti distribuiti si possono pensare strutturate in 3 livelli:

- applicazione: è il nucleo computazionale più esterno
- librerie: è lo strato delle routine richiamate dall'applicazione, costituito da librerie numeriche consolidate che costituiscono lo standard de-facto per varie classi di problemi dell'analisi numerica (es. PETSc, NAG, Scalapack, ...)

- protocolli: è lo strato costituito dai protocolli e i software di basso livello che servono alla comunicazione e la cooperazione delle risorse.

L'idea è, quindi, di individuare per il livello più basso (protocolli) gli standard che consentano di rilevare e gestire i guasti, a livello intermedio, librerie che siano fault tolerant e lasciare al livello delle applicazioni solo il "coordinamento" delle routine delle librerie fault tolerant. Tuttavia spesso nessuna o una sola parte del sistema suddetto (applicazione/ ambiente di esecuzione) è capace di fronteggiare situazioni inattese durante l'esecuzione ed è quindi necessario assolvere a tale carenza. L'implementazione di meccanismi che abilitino la tolleranza ai guasti vanno, dunque, ricercati a livello di sistema e a livello di applicazione. I primi dovranno essere basati su standard durevoli, mentre i secondi dovranno essere realizzati in maniera efficiente e portabile.

Il presente lavoro di ricerca nasce con il duplice l'obiettivo di

- costruire e implementare in maniera robusta ed efficiente una metodologia ibrida di checkpointing all'interno dell'algoritmo del Gradiente Coniugato (CG) della libreria per il calcolo scientifico PETSc, al fine di produrne una nuova versione fault tolerant;

- realizzare un meccanismo automatico che, a seguito di uno o più fault dei nodi di calcolo, esegua una migrazione dell'applicazione, che utilizza il CG fault tolerant, sulle risorse alternative dell'ambiente distribuito.

Dopo una breve introduzione al concetto di fault tolerance, nel *Capitolo 1* si pone particolare attenzione su cosa significhi, per un'applicazione scientifica "essere fault tolerant", passando poi a descrivere alcune classi di applicazioni "naturalmente fault tolerant" ed altre che non lo sono, soffermando poi l'attenzione su queste ultime. Nel *Capitolo 2* si descrivono le principali strategie per implementare la fault tolerance in algoritmi paralleli mediante tecniche di checkpointing, soffermando l'attenzione su vantaggi e svantaggi di ciascuna tecnica. Nel *Capitolo 3* è descritto l'ambiente di calcolo parallelo e distribuito in cui è stato svolto il lavoro di ricerca, la libreria PETSc e l'algoritmo del Gradiente Coniugato utilizzato come caso di studio. Nel *Capitolo 4* è riportato il lavoro svolto per la costruzione della strategia di checkpoining ibrida e le modifiche apportate all'interno del modulo del CG di PETSc per implementare la fault tolerance algorithm-based, mediante i meccanismi di checkpointing basati sulla strategia ibrida realizzata. Nel *Capitolo 5* si descrive l'attività che ha portato alla realizzazione del sistema di *checkpointing/migration* che, dall'esterno, osserva e gestisce il flusso esecutivo dell'applicazione. Nel *Capitolo 6* vengono illustrati alcuni esperimenti, svolti nell'infrastruttura distribuita di Ateneo, con il fine di:

- valutare la validità del sistema di checkpointing realizzato, riportando alcune considerazioni sull'overhead introdotto nelle fasi di salvataggio dei dati in esperimenti con fault e failure-free, comparando i comportamenti del software al variare del supporto utilizzato per la memorizzazione dei dati (disco locale al nodo, area condivisa tra i nodi mediante protocollo NFS, area condivisa mediante filesystem distribuito Lustre);
- validare il sistema di migrazione locale e remota dell'applicazione che fa uso della implementazione fault tolerant del CG

In *Allegato*, ulteriori dettagli sul codice sviluppato.

Capitolo 1

La fault tolerance

1.1 Sistemi di calcolo, fault tolerance e affidabilità

La tolleranza ai guasti, *fault tolerance*, è la capacità di un sistema di evitare interruzioni di servizio anche in presenza di guasti.

La capacità di tollerare i guasti aumenta il livello di affidabilità di un sistema e spesso si ottiene mediante il ricorso alla ridondanza dei componenti (aggiungendo più componenti dello stesso tipo per svolgere la medesima funzione, in modo che l'eventuale avaria di uno sia compensata dagli altri).

Una misura dell'affidabilità di un componente hardware è il tempo medio di guasto MTBF (**M**ean **T**ime **B**etween **F**ailures), un parametro di qualità applicabile a dispositivi meccanici, elettrici ed elettronici e ad applicazioni software.

In particolare:

$$MTBF = MTTF + MTTR$$

dove, MTTF e il MTTR sono, rispettivamente, il **M**ean **T**ime **T**o **F**ailure e il **M**ean **T**ime **T**o **R**epair.

Supponendo che il valore di MTTF, per un unico processore, sia

10^5 ore (circa 11 anni), in un sistema di calcolo costituito da 50 processori, ci si deve aspettare che un nodo subisca un fault con maggior frequenza (circa ogni 5 anni) [30].

Con l'aumento della grandezza del sistema di calcolo, poi, gli esempi diventano ancor più significativi. Consideriamo le risorse collocate ai primi cinque posti (dati di Giugno 2009) della Top500 Supercomputers[1]: esse hanno un numero di processori compreso tra 51200 [2] e 294912 [3].

Il MTTF per uno qualsiasi dei 294912 processori del sistema si può stimare come segue:

$$MTTF = MTTF_{1proc}/numProc = 10^5/294912$$

cioè, mediamente, ogni 20 minuti un processore del sistema può guastarsi.

Inoltre, un sistema di calcolo è costituito non solo da componenti hardware, ma anche dal software applicativo e da qualsiasi middleware[4] da cui dipende l'esecuzione del software stesso.

Se consideriamo un sistema di calcolo parallelo a memoria distribuita, costituito da n nodi, la sopravvivenza delle applicazioni dipende, in genere sia dal funzionamento dei livelli software che utilizza, sia dall'integrità del sistema di calcolo, durante l'intera esecuzione.

L'occorrenza di guasti hardware non deve perciò essere gestita solo a livello hardware ma anche a livello del middleware utilizzato dalle applicazioni durante l'esecuzione, e talvolta a livello delle applicazioni stesse.

[1] La Top500 è la lista dei più potenti 500 supercomputer internazionali. Viene stilata due volte all'anno sulla base dei valori di prestazioni ottenuti mediante il benchmarch parallelo di algebra lineare HPL.

[2] Quarto posto della Top500; nome della risorsa:Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI; sito:NASA/Ames Research Center/NAS, USA

[3] Terzo posto della Top500; nome della risorsa: JUGENE; tipologia/vendor: Blue Gene/P Solution 2009 IBM; Sito: Forschungszentrum Juelich (FZJ) Germany

[4] Strato software costituito da ogni package, libreria o servizio di più alto livello, venga utilizzato dall'applicazione

1.2 Applicazioni e fault tolerance

Le applicazioni per il calcolo scientifico richiedono sempre più risorse per risolvere i cosiddetti *problemi-sfida* e tali risorse possono essere organizzate secondo i classici o più moderni paradigmi di calcolo, dal parallelo al distribuito, dal grid al cloud computing.

Più complesso è il contesto di calcolo più diventa vera l'affermazione precedente secondo cui la tolleranza ai guasti va gestita a più livelli.

Inoltre, anche a parità di contesto di calcolo, le applicazioni per il calcolo scientifico possono avere modelli di esecuzione molto variabili che comprendono algoritmi caratterizzati da task debolmente accoppiati e altri che richiedono una forte interazione e cooperazione tra i vari task.

Alcune di esse eseguono algoritmi banali basati su modelli di esecuzione concorrente, con task indipendenti tra loro o al più con un paio di punti di comunicazione (di solito all'inizio e alla fine della computazione, non durante). In tal caso, un task fallito, per un qualsiasi motivo, viene "automaticamente" riallocato dal sistema su altre risorse disponibili, rendendo, di fatto, l'applicazione fault tolerant.

Altre utilizzano, durante l'esecuzione, meccanismi di replica del task e verifica dei risultati [31]: è il caso ad esempio di SETI@home[5], che esegue in un ambiente distribuito in cui non si hanno informazioni certe sulla disponibilità di risorse di calcolo durante l'esecuzione.

A parte le applicazioni studiate per gli ambienti distribuiti e quelle *embarassing parallel* per le quali la fault tolerance è semplice da

[5]SETI@home (SETI at home) è un progetto di calcolo distribuito per Personal Computer, portato avanti dall'Università di Berkeley. SETI è un acronimo per Search for Extraterrestrial Intelligence (Ricerca di Intelligenza Extraterrestre). Lo scopo di SETI@home è quello di analizzare i dati provenienti dal Radiotelescopio di Arecibo alla ricerca di eventuali prove di trasmissioni radio provenienti da intelligenza extraterrestre.

gestire, numerose ricerche, condotte in ambito internazionale hanno portato, ad individuare speciali classi di applicazioni che, tra l'altro hanno la capacità di "ben tollerare" i guasti

Nei prossimi paragrafi sono riportati alcuni esempi di classi di applicazioni che naturalmente tollerano i guasti, altre in cui è possibile implementare la fault tolerance mediante un insieme di "linee guida".

1.2.1 Applicazioni "naturalmente fault tolerant"

Geist and Engelmann in [10] definiscono un'applicazione "naturalmente fault tolerant" quando essa è in grado di fornire il risultato corretto nonostante il fallimento di alcuni task durante l'esecuzione. Ciò è dovuto non a meccanismi di sostituzione della risorsa guasta con una nuova che ne svolgerà il compito, ma piuttosto ad una proprietà dell'algoritmo che, anche in presenza di dati mancanti, è in grado di costruire una buona soluzione del problema.
Geist and Engelmann in [16] individuano una classe di algoritmi che definiscono *super-scalabili* che, oltre ad essere *scale-invariant* [6] relativamente al numero di task con i quali ogni task comunica, sono anche naturalmente fault tolerant.

Tra gli algoritmi che godono di tale proprietà sono, ad esempio, alcune implementazione dei metodi alle differenze finite e gli algoritmi di comunicazione ad albero binario [10]. Tuttavia la proprietà di essere *scale-invariant* non è sufficiente a garantire, in caso di fallimento di uno o più task, il raggiungimento di risultati corretti. L'algoritmo deve essere in grado, sfruttando le sue proprietà matematiche, di compensare la perdita di informazioni.

[6] Scale invariant significa che ogni singolo task, in un'applicazione parallela, ha un numero fissato di task con cui comunicare per il calcolo, indipendentemente dal numero totale di task dell'applicazione.

Esistono almeno due classi di applicazioni, non banali, che possono godere della proprietà di super scalabilità:

- quelle in cui, nell'algoritmo, ogni singolo task lavora su informazioni locali (ad esempio alcune implementazioni dei metodi alle diferenze finite o agli elementi finiti) e
- quelle in cui, nell'algoritmo, ogni task richiede informazioni globali (ad esempio i metodi iterativi che si sincronizzano mediante operazioni collettive).

Oltre agli algoritmi *scale-invariant*, godono della proprietà di essere naturalmente fault tolerant anche gli algoritmi asincroni. Ne è un esempio ad esempio l'algoritmo APPS (**A**syncronous **P**arallel **P**attern **S**earch) per la risoluzione di problemi di ottimizzazione non lineare [17]

Tuttavia la tolleranza ai guasti naturale è un requisito molto stringente per un algoritmo ed è in genere posseduto da poche applicazioni.

1.2.2 Applicazioni non "naturalmente fault tolerant"

Nel caso di applicazioni di calcolo parallelo che non godono delle proprietà definite nel paragrafo precedente, si può dire che l'implementazione efficiente della tolleranza ai guasti sia ancora un problema aperto, e notevole è l'attenzione per tutti i meccanismi che abilitino la fault tolerance introducendo un minimo overhead.

Il fatto di dover introdurre la fault tolerance in modo da preservare le prestazioni delle applicazioni per ambiente HPC[7], rende questo problema di difficile trattazione da momento che tali applicazioni sono costituite da task fortmente accoppiati e alti livelli di sincronizzazione.

[7] **H**igh **P**erformance **C**omputing

L'obiettivo principale, per queste applicazioni, è quello di tollerare gli errori più comuni (guasto di uno o più nodi, problemi alla rete di interconnessione, ...), introducendo un sovraccarico minimo in caso esecuzione priva di fault.

Recentemente, per tali applicazioni, si sta cercando di adattare gli storici meccanismi di *detect/notify/recovery* [22] ai contesti di supercalcolo distribuito descritti nel paragrafo 1.1.

Le strategie utilizzate, per l'implementazione di tali meccanismi, si basano storicamente su tre fasi:

1. rilevamento del fault
2. notifica all'applicazione
3. recovery dal fault

Mentre il rilevamento del fault e la fase di notifica all'applicazione sono, in genere, a carico dell'ambiente di esecuzione, la fase di salvataggio e recupero dei dati devono essere gestiti dall'applicazione utilizzando, in taluni casi, tecniche fatte su misura per gli algoritmi che l'applicazione utilizza.

Esistono infatti algoritmi, come alcuni metodi iterativi per l'algebra lineare, che non possono fare a meno di meccanismi di salvataggio periodico dei dati (o *checkpointing*) per sopravvivere ai guasti, scegliendo soluzioni che riducano l'overhead sia nei casi di esecuzione *failure-free*, sia nelle fasi di salvataggio/recupero dei dati.
Nel prossimo capitolo saranno illustrate le principali tecniche per il checkpointing di solito utilizzate per tali applicazioni e alcune considerazioni sugli ambienti software fault tolerant. Infine sono presentati alcuni casi di strategie di checkpointing ibride, nate cioè dalla combinazione di due o più strategie di base.

Capitolo 2

La fault tolerance mediante tecniche di checkpointing

Per quanto detto nel *Capitolo 1*, la frequenza con cui un sistema parallelo subisce un fault sta aumentando con l'aumentare delle dimensioni dei sistemi di calcolo.

D'ora in avanti considereremo sempre il caso di un'applicazione parallela non naturalmente fault tolerant in esecuzione su architettura parallela a memoria distribuita costituita da p processori.

Perchè l'applicazione possa continuare la propria esecuzione, anche a seguito di un fault sia temporaneo sia duraturo, devono esistere meccanismi che rilevino e notifichino la presenza di eventuali anomalie, ed altri che, conservino lo stato di esecuzione corrente, consentendo all'applicazione di riprendere il calcolo un secondo momento, da dove era stata interrotta a causa del fault.

In generale, compito dell'ambiente sarà quello di "preservare" (o essere in grado di ricostruire) un contesto di esecuzione coerente per l'applicazione che ha subito il fault, mentre saranno a carico dell'applicazione le operazioni di salvataggio/recupero dei dati necessari a riprendere l'esecuzione senza perdere il lavoro svolto.

Nel caso di applicazioni di grande durata, infatti, è molto importante non perdere ore di calcolo già spese ed è, dunque, necessario abilitare meccanismi in grado di “fotografare” lo stato dell'applicazione al termine di ogni passo significativo dell'esecuzione; tali meccanismi sono detti di *checkpointing*.

Nel *Capitolo 1* abbiamo affermato che le fasi di *detect/notify* sono a carico dell'ambiente di esecuzione, mentre la fase relativa al *recovery* spetta all'applicazione. Vedremo che è possibile dare in carico all'ambiente anche la fase di *recovery* dell'applicazione, ma questo approccio in alcuni contesti risulta inefficiente.

2.1 Metodologie di checkpointing

Il checkpointing è una procedura che consiste nel salvare un insieme di dati che, in caso di fault, permettano di riavviare il processo dal punto di esecuzione in cui si è eseguito l'ultimo salvataggio (o checkpoint).

Memorizzando periodicamente i dati relativi allo stato corrente di esecuzione è possibile effettuare il recovery dell'applicazione dal *checkpoint* più recente, effettuando su di esso l'operazione che va sotto il nome di *rolling back*.

È possibile effettuare il checkpointing utilizzando come supporto la memoria di massa, e si parla di checkpointing *disk-based*, oppure la memoria principale, e si parla di checkpointing *diskless* [25].

2.1.1 Integrità, latenza ed overhead di checkpointing

Ogni checkpoint deve rimanere valido finché il successivo checkpointing non è stato completato: solo in questo modo, infatti, i meccanismi di checkpointing sono robusti e non falliscono anche se il fault si verifica durante le fasi di salvataggio dei dati. Se il checkpoint attuale risulta coerente, è possibile eliminare, dalla memoria, quello precedente: questa procedura garantisce l'**integrità del checkpointing**.

Nell'implementare una metodologia di checkpointing (o combinare due o più metodologie), è necessario tenere sotto controllo due parametri: la latenza e l'overhead [34]

L'**overhead del checkpointing** è l'aumento del tempo totale di esecuzione dell'applicazione dovuto ai meccanismi di checkpointing. (overhead dell'esecuzione failure-free + overhead della fase di rolling back)

La **latenza del checkpointing** è il tempo necessario a salvare i dati di checkpointing.

In generale, un meccanismo di checkpointing dovrà possedere caratteristiche di robustezza e di efficienza (tenendo sotto controllo parametri come la latenza e l'overhead).

2.2 Checkpointing disk-based e diskless

I meccanismi di checkpointing/recovery possono essere implementati nel codice delle applicazioni, come sarà descritto nel paragrafo 2.3, oppure realizzati dall'esterno mediante ambienti software fault tolerant che gestiscono le fasi di salvataggio/recupero dell'applicazione in maniera completamente trasparente. In ogni caso è possibile utilizzare, per le fasi di checkpointing e rolling back, sia approcci di tipo disk-based, sia diskless [20].

Nel disk-based checkpointing, in fase di salvataggio dati, un processore effettua il checkpoint del processo salvando l'intero spazio di indirizzi di quest'ultimo sul disco; in caso di fault, nella fase di rolling back, i processori "sopravvissuti" recuperano il loro stato utilizzando i dati di checkpoint salvati localmente, mentre uno o più processori prenderanno il posto dei processori guasti mediante meccanismi automatici che dipendono dall'implementazione.

In letteratura sono presenti 2 approcci di base e uno ibrido:

- **approccio indipendente:** [37, 28]
  - In fase di checkpointing ogni processore, senza sincronizzarsi prima con gli altri processori, salva su uno "stable storage"[1] un insieme di checkpoint locali: poichè non ha informazioni sul momento in cui gli altri processori effettuano il checkpointing, ogni processore salva non uno, ma un insieme di checkpoint locali alla volta. Ogni processore, indipendentemente, fa lo stesso.
  - In fase di rolling back, al fine di costruire un checkpoint globale e coerente dell'applicazione, i processori cercano tra i gruppi di checkpoint locali salvati da ogni processore, quelli "allineati". Se un checkpoint globale e coerente non può essere ripristinato, si può incorrere in un "effetto domino" di rolling back che può condurre addirittura a riportare i processori all'inizio dell'esecuzione.
  - a parte lo svantaggio dell'effetto domino, questo approccio ha un vantaggio: non c'è overhead di comunicazione.
- **approccio coordinato:** [28]
  - In fase di checkpointing, i processori prima si coordinano e poi, contestualmente, salvano su uno "stable storage" i propri checkpoint.
  - In fase di rolling back, i processori accedono allo stato coerente dell'applicazione e proseguono nel calcolo. Esiste sempre un checkpoint globale coerente e quindi non può aver luogo l'"effetto domino"

[1] Per stable storage, si intende una qualsiasi tipologia di memoria non volatile: un disco locale, una SAN - Storage Area Network,...

  - questo approccio ha uno svantaggio: c'è overhead di comunicazione e di I/O.

- **approccio semi-coordinato:**
  - La strategia si basa sull'alternanza dell'approccio indipendente e di quello coordinato
  - In questo modo riesce a moderare l'overhead di comunicazione (mediante la diminuzione della frequenza delle sincronizzazioni dovute all'approccio coordinato) e blocca l'effetto domino poichè garantisce l'esistenza di un checkpoint globale coerente.

Riguardo agli approcci disk-based si può affermare che, in generale, sono caratterizzati da semplicità di implementazione, un alto livello di affidabilità e una buona copertura dei guasti, nonchè uno scarso overhead di comunicazione.

Tuttavia, poichè i dati da salvare durante la fase di checkpointing possono arrivare a dimensioni considerevoli per processore, la fase di scrittura su memoria secondaria rappresenta la principale fonte di degradazione di prestazioni delle applicazioni. Per migliorare le prestazioni si possono utilizzare approcci diskless.

Nel checkpointing diskless, infatti, il processo salva il suo spazio di indirizzi nella memoria principale invece che sul disco; è un'evoluzione del cosiddetto "checkpointing coordinato". Nello schema del checkpointing coordinato i processori a memoria distribuita, che eseguono l'applicazione, salvano un checkpoint globale dello stato di esecuzione, costituito dai singoli checkpoint relativi ai singoli processori e da un resoconto (log file) dei messaggi in transito durante l'operazione di checkpointing. Il checkpointing diskless si differenzia da quello coordinato perchè evita l'utilizzo del log file appena citato, riconducendo la fase di checkpointing al salvataggio dei singoli checkpoint relativi ad ogni processore. La memorizzazione dei dati che costituiscono il checkpoint dell'applicazione nella memoria principale del calcolatore è una strategia particolarmente indicata per tutte le applicazioni che richiedono lunghi tempi di calcolo poichè, in tal caso, è possibile aumentare la frequenza con la quale si effettuano i checkpoint senza degradare eccessivamente le prestazioni dell'applicazione, con l'indubbio vantaggio di dover ripetere pochi calcoli, per ripristinare lo stato di esecuzione.

In letteratura sono presenti 2 approcci:

- **neighbor-based checkpointing:** [27]
  - In fase di checkpointing, ogni processore salva i checkpoint nella sua memoria principale e in una parte della memoria principale del processore "vicino"
  - In caso di fault il processore che sostituisce quello guasto effettua la fase di rolling back mediante leggendo il checkpoint del processore guasto dal processore vicino.
  - l'approccio tollera sempre il fault di un nodo alla volta, di più nodi simultaneamente solo se questi non sono adiacenti, non tollera il crash dell'intero sistema, per cui va sempre abbinato a checkpointing disk-based non frequenti.
  - è caratterizzato grande overhead di memoria (il doppio di quanta ne servirebbe)
- **parity-based checkpointing:** [18]
  - L'applicazione necessita di n processori per il calcolo (*application processor*) ed m processori extra che non prendono parte al calcolo (*checkpoint processor*). Questi ultimi possono essere utilizzati per rimpiazzare fino ad m application processor interessati da fault.
  - L'approccio nasce dalla combinazione di due componenti: *local checkpointing* (salvataggio locale dei dati di check-

point nella memoria principale degli application processor) e *checkpoint enconding* (tecniche di codifica dei checkpoint)

- Gli application processor effettuano il *local checkpointing* utilizzando uno dei seguenti metodi [18, 25]: simple checkpointing, incremental checkpointing, forked checkpointing
- I checkpointing processor codificano i checkpoint degli application processor mediante una delle seguenti tecniche [20, 7]: Parity (o RAID level 5), Mirroring, 1-dimensional parity, 2-dimensional parity, Reed-Solomon.
- In caso di fault, i processori superstiti effettuano il rolling back dal proprio checkpoint locale; i processori sostitutivi inizializzano il proprio stato a partire dai checkpoint dei superstiti e dai dati codificati.
- è caratterizzato da overhead di comunicazione verso i checkpoint processor e da un ingente utilizzo della memoria principale. Inoltre la fase di codifica di una grande quantità di dati richiede molto tempo, comportando un considerevole aumento dei tempi di esecuzione.

In entrambi gli approcci disk-based e diskless si rileva un uso oneroso della memoria: ogni nodo salva una copia dello spazio degli indirizzi dell'applicazione e non c'è alcuna garanzia che tutti i dati salvati siano veramente indispensabili per il recupero dell'applicazione parallela. Ciò è dovuto al fatto che le suddette metodologie salvano l'intero spazio di esecuzione dell'applicazione. In genere, infatti gli schemi suddetti sono implementati in maniera trasparente, sfruttando il vantaggio che è l'ambiente di calcolo che si occupa di tutto, non c'è bisogno di intervenire sul codice.

Per tale ragione, negli anni numerosissime sono state le implementazioni delle metodologie trasparenti, con la realizzazione di ambienti anche all'interno delle stesse librerie per il *message passing*[2] le cui implementazioni sono basate sullo standard MPI (*M*essage *P*assing *I*nterface)[3] [3]

Tra di esse le più diffuse sono: LAM [36], OpenMPI [19, 11], e MPICH-V [5]. Praticamente tutte si basano su meccanismi trasparenti di *stop/recovery/restart* dell'applicazione.

2.3 Checkpointing algorithm-based

Il principale obiettivo dell'utilizzo di metodologie trasparenti per la fault tolerance è realizzare un'applicazione fault tolerant senza introdurre modifiche ai codici scritti; tuttavia questa metodologia, che di solito si basa su un *dump binario* della memoria, non è portabile. Per tale motivo, questo tipo di soluzione non è particolarmente adatta ad essere adottata in ambienti di tipo distribuito, di natura altamente dinamica ed eterogenea. L'eterogeneità delle risorse di calcolo, con questo tipo di approccio, a seguito di un fault la possibilità di proseguire l'esecuzione dell'applicazione su una risorsa differente da quella iniziale (ad edempio con un tipo di processore diverso).

Inoltre, anche se si utilizzano gli approcci diskless, i dati di chec-

[2]Il message passing è un paradigma usato ampiamente su una specifica tipologia di macchine parallele, specialmente quelle a memoria distribuita. Anche se ci sono molte varianti, il concetto di base della comunicazione mediante messaggi è facilmente comprensibile. Il message passing si basa sull'utilizzo di più processori i quali possono accedere solo alla propria memoria (memoria distribuita), il programma viene eseguito parallelamente su ogni processori e lo scambio di informazioni da parte dei processori avviene solo attraverso lo scambio di messaggi.

[3]MPI nasce dall'esigenza di definere uno standard unico per le libreria di message massing per l'Europa e gli Stati Uniti. Nei primi anni '90 negli Stati Uniti erano presenti sul mercato diversi calcolatori a memoria distribuita, furono sviluppate sviluppate librerie di message passing che avevano l'obiettivo di fornire la portabilità su architetture eterogenee. Jack Dongarra e Tony Hey furono tra i primi a riconoscere l'importanza di definire uno standard unico per il message passing per evitare la definizione di standard diversi per la comunità europea e americana. Nell'Ottobre 1992 fu rilasciata la prima versione dello standard MPI-0, successivamente fu istituito l'MPI Forum dove furono create le sottocommissioni per l'ampliamento dello standard. Nel 1995 è stata rilasciata la versione 1.1 dello standard, l'ultima versione la 2.1 risale al 2008.

kpointing sono rilevanti e questo comporta comunque un utilizzo inefficiente della memoria.

La soluzione ai suddetti problemi può essere fornita dagli approcci *algorithm-based* che, agendo dall'interno del codice dell'applicazione, selezionano quali dati salvare, le modalità e la frequenza della fase di salvataggio, in modo da implementare strategie più efficienti.

Tuttavia l'approccio algorithm-based, richiede la conoscenza dei codici degli algoritmi che si vuole rendere fault tolerant, ed un conseguente notevole lavoro per il programmatore. Lo studio dell'algoritmo è finalizzato all'individuazione delle strutture dati da salvare durante il checkpointing, cioè tutti e soli i dati necessari all'applicazione, in fase di ripristino dopo il fault, per riprendere l'esecuzione dall'ultimo checkpoint effettuato.

Riassumendo, le fasi di *detect/notify* rimangono a carico dell'ambiente di esecuzione dell'applicazione, mentre la fase di *recovery* è demandata a funzionalità da implementare all'interno degli algoritmi.

Anche gli approcci algorithm-based possono avere implementazioni disk-based e diskless. L'implementazione disk-based si può vedere come ottimizzazione dell'approccio coordinato discusso nel paragrafo precedente: la differenza è che i dati che vengono salvati sono tutti e soli quelli necessari alla ripresa dell'applicazione in fase di rolling back.

Un approccio di tipo algorithm-based è strettamente dipendente dal tipo di algoritmo: Nel caso di algoritmi iterativi, ad esempio, si tende a considerare, per la fase di salvataggio, le variabili che compongono il ciclo di calcolo, mentre le variabili relative alla fase di inizializzazione non vengono salvate e l'inizializzazione deve essere rieseguita in fase di rolling back. In generale la fase di salvataggio dei dati e quella di ripristino sono poste rispettivamente alla fine del ciclo di calcolo e prima di esso. Come appare ovvio, infatti, i dati vengono salvati solo quando sono già terminate tutte le operazioni contenute nel ciclo, mentre il recupero avviene prima dell'esecuzione del ciclo di calcolo che deve essere riavviato dall'iterazione nella quale è stato effettuato l'ultimo salvataggio dei dati.

Due esempi molto diffusi di librerie di funzioni, utili per l'implementare il checkpointing algorithm-based, sono SRS (**S**top **R**estart **S**oftware) [32] e FT_MPI (**F**ault **T**olerant MPI) [12]

La prima è una libreria che, utilizzando MPI e la libreria IBP[4] (**I**nternet **B**ackplane **P**rotocol) [38], implementa le fasi di detect/-notify fermando l'esecuzione dell'applicazione, dopo aver salvato i checkpoint su risorse di storage denominate IBP_DEPOT.

Gli IBP_DEPOT possono essere locali ai nodi (i dischi dei nodi stessi che effettuano il calcolo), oppure remoti: ogni nodo deve avere un IBP_DEPOT. In entrambe i casi la fase di checkpointing è affetta da un overhead più o meno consistente dovuto alla scrittura su disco. Nel caso remoto, oltre all'overhead aumenta la latenza del checkpointing in quanto la fase di scrittura è bloccante per l'applicazione. Analoghe considerazioni si possono fare nella fase di rolling back che segue il restart manuale dell'applicazione [33].

FT_MPI, invece è un'implementazione del paradigma di message passing che ha aggiunto funzionalità fuori dall'attuale versione dello standard. Tali funzionalità consentono all'applicazione di non dover fermare e riprendere l'esecuzione in un secondo momento, ma di "risolvere il fault" durante l'esecuzione.

Tali funzionalità, possedute solo da questa libreria, consentono di implementare la fault tolerance algorithm-based mediante tecniche di checkpointing diskless [8].

Tuttavia, la sua ultima implementazione risale all'anno 2003 e,

[4] Internet Backplane Protocol (IBP) è un middleware per la gestione e l'utilizzo di risorse di storage remote.

sebbene possa ancora essere utilizzata, presenta alcuni problemi che la rendono di fatto instabile su alcune architetture, e questo è sconveniente in ambienti distribuiti, tipicamente eterogenei. [9]

Per le ragiorni appena enunciate, nello svolgimento del presente lavoro, si è scelto di utilizzare le sole implementazioni standard di MPI, come vedremo nel *Capitolo 4*

Nel prossimo Capitolo, invece, saranno introdotti sia l'ambiente di sviluppo sia il caso di studio del presente lavoro di ricerca.

Capitolo 3

Il contesto di lavoro e il casus studii

Consideriamo un sistema distribuito costituito da un certo numero di risorse di storage e di *cluster*(architetture MIMD[1] [24] a memoria distrbuita), organizzate secondo il paradigma del GRID [13, 14] computing.

L'utilizzo di questo tipo di infrastrutture presenta i seguenti vantaggi:

- localmente al cluster, è possibile ottenere elevate prestazioni (grazie al calcolo parallelo);
- globalmente, considerando tutta cioè tutta l'architettura distribuita, c'è una grande disponibilità di risorse di calcolo e storage, di solito non reperibili in un unico centro di calcolo.

È in tale contesto che si svolge l'attività di ricerca descritta nei prossimi capitoli.

[1]Nell'architettura MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata) multi computer ogni processore ha una memoria propria. I vari processori comunicano tra loro mediante uno scambio di messaggi (paradigma del message passing). La comunicazione tra i vari processori è affidata a una rete di interconnessione (dedicata e in genere a bassa latenza) che consente a ciascun processore di trasmettere dati a qualunque altro processore presente nella stessa rete.

Nella prossima sezione saranno descritte le caratteristiche generali dell'architettura distribuita, basata su paradigma GRID, utilizzata per lo sviluppo del sistema di *checkpointing/migration*.

Nelle rimanenti sezioni di questo capitolo saranno, invece, riportate le caratteristiche della libreria di message passing scelta e la libreria per il calcolo scientifico PETSc su cui si è lavorato, con particolare riguardo al modulo della libreria che è stato scelto come casus studii, per l'introduzione dei meccanismi per la tolleranza ai guasti.

Nel *Capitolo 5*, invece, saranno descritti, più in dettaglio, alcuni strumenti del GRID middleware, che sono stati utilizzati per l'implementazione dei meccanismi di migrazione automatica dell'applicazione.

3.1 L'architettura distribuita

L'architettura considerata, in questo lavoro di ricerca, basata sul paradigma del Grid Computing, è costituita da un punto di accesso all'ambiente per gli utenti, da risorse di calcolo di tipolgia cluster, da risorse di storage e servizi di gestione. Il middleware di griglia utilizzato è gLite [21] [2], realizzato nell'ambito del progetto europeo EGEE [1]. [3]

In fig. 3.1 è mostrato il layout dell'architettura basata sul middleware gLite che, dall'alto verso il basso, comprende:

- Un punto di accesso alla GRID, che è una User Interface (UI) [6] che dispone di tutti i comandi client per l'autenticazione e l'autorizzazione, la ricerca di risorse, la sottomissione e il monitoraggio di job, il trasferimento dei dati,

[2] Lightweight Middleware for Grid Computing
[3] Enabling Grid for E-sciencE.

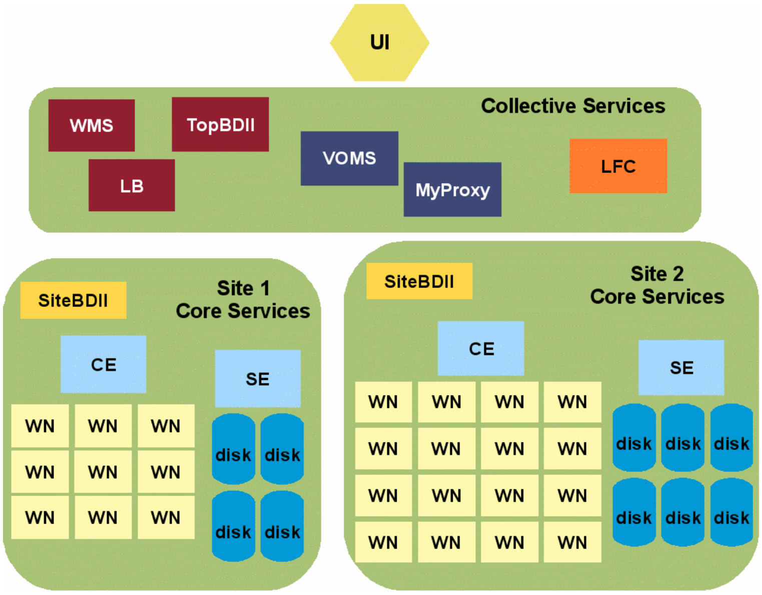

Figura 3.1: L'architettura GRID utilizzata

- I servizi *collective* [6] per la gestione delle interazioni tra tutte le risorse dell'ambiente distribuito, che comprendono i servizi per la gestione della sicurezza [4], i servizi informativi [5], i servizi per la sottomissione e il monitoraggio dei job [6], i cataloghi di file, [7]

- I servizi *core* [6] per il calcolo [8] e per la memorizzazione dei dati.[9].

[4]VOMS server, MyProxy server
[5]Top_ e Site_ BDII - Berkeley Database Information Index di primo e secondo livello
[6]WMS - Workload Management System e LB - Logging Bookeeping
[7]LFC - Logical File Catalogue
[8]Computing Element e Worker Nodes - CE e WN
[9]Storage Element -SE

3.2 La libreria di Message Passing

Come anticipato, nel *Capitolo 2*, in questo lavoro si deciso di utilizzare una delle implementazioni standard di MPI. Tra le tante, la scelta è ricaduta su MVAPICH poichè è un'implementazione ad alte prestazioni di MPI *over infiniband*[10].

MVAPICH supporta varie interfacce:

- L'interfaccia OpenFabrics/Gen2 offre alte prestazioni e scalabilità.
- Memory shared only channel: utile per esecuzione di job MPI su sistemi multi-processore senza utilizzare una rete ad alte prestazioni.
- QLogic InfiniPath: fornisce un supporto nativo per gli adapter Qlogic. Offre alte prestazioni nelle comunicazioni punto a punto ed in quelle collettive.

Per questa implementazione di MPI è stato sviluppato un sistema di *job startup*, `mpirun`, più scalabile e robusto che in altre implementazioni poichè è stato progettato per l'avvio sincronizzato di applicazioni che coinvolgono migliaia di nodi, garantendo comunque alte prestazioni e un utilizzo minimale della memoria.

MVAPICH, inoltre, gestisce tre tipi di fault tolerance:

- garantisce l'affidabilità dei dati trasferiti nelle comunicazioni *mem-to-mem* nel caso di utilizzo della memoria condivisa, vengono rilevati gli errori del bus I/O con il CRC a 32 bit, i dati vengono ritrasmessi in caso di errore.
- implementa la fault tolerance a livello della rete InfiniBand, in

[10]tecnologia di rete ad alta velocità e bassa latenza

modo analogo ad un'altra implementazione LA-MPI [2], mediante l'uso dell'APM (Automatic Path Migration).

- fornisce un meccanismo di checkpoint start/restart analogo a quello realizzato da LAM/MPI e OpenMPI.

3.3 La libreria PETSc

PETSc (**P**ortable, **E**xstensible **T**ookit for **S**cientific **c**omputation) è una libreria di strutture dati e metodi per il calcolo scientifico sviluppata dal Dipartimento di Matematica e Computational Science dell'Argonne National Laboratory [4].

Essa è utilizzata per l'implementazione di un gran numero di applicazioni in ambiente di calcolo ad alte prestazioni (su architetture MIMD a memoria distribuita) per la risoluzione di problemi in svariati campi scientifici (dalla medicina, alla geofisica, alla biologia, ...) modellati da equazioni differenziali alle derivate parziali PETSc è una libreria flessibile, i cui moduli sono stati sviluppati seguendo la filosofia del paradigma Object Oriented e possono essere utilizzati in codici scritti in *Fortran*, C o C^{++}.

La libreria è caratterizzata da una una struttura gerarchica (fig. 3.2) che, dal basso verso l'alto, evidenzia diversi livelli di astrazione per la scelta del solutore da utilizzare per la risoluzione dei problemi.

Il livello più basso della gerarchia è costituito dalla libreria di Message Passing (MPI)[11] e dalle ormai consolidate librerie di algebra lineare BLAS e LAPACK.

Al di sopra del blocco di librerie di base, in particolare, si collocano:

[11] È possibile installarla utilizzando una qualsiasi delle implementazioni di MPI, in questo lavoro si è scelto MVAPICH, cfr.3.2

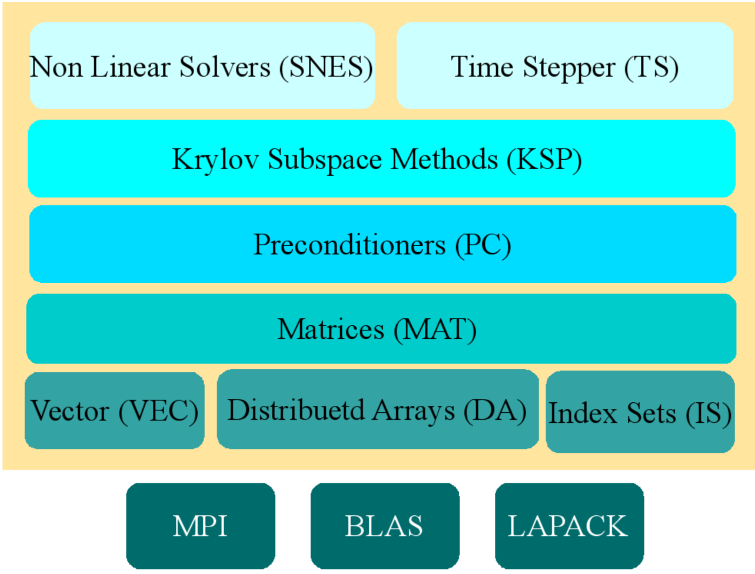

Figura 3.2: Struttura Gerarchica di **PETSc**

- i più diffusi solutori lineari iterativi basati su metodi di proiezione di Krylov (*KSP* - Krylov Subspace Methods [15, 29]) (GMRES, CG, CGS, Bi-CG-Stab, ...) e, ad un livello di astrazione sotto, i principali precondizionatori (*PC*)(Additive Schwartz, Block Jacobi, ILU, ICC,...) per tali metodi.
- i principali solutori non lineari, (*SNES*) con metodi Newton-like.

Per ogni elemento da considerare, nel problema assegnato, ad esempio un vettore o una matrice nel prodotto matrice-vettore, PETSc costruisce il corrispondente oggetto *Vec* e *Mat*[12].

Con gli oggetti collocati a un livello più di astrazione più elevato (e quindi all'aumentare della complessità del problema assegnato), PETSc si comporta in modo analogo; ad esempio:

[12]Ciascun oggetto PETSc è stato definito in relazione all'operazione che deve eseguire e non ai dati che deve contenere. Ad esempio un vettore *VEC* non è considerato un array di valori, ma un oggetto astratto su cui sono definite operazioni di addizione e moltiplicazione scalari.

- per la risoluzione di un sistema lineare costruisce l'oggetto *KSP*, che può includere oppure no l'oggetto precondizionatore *PC*, a seconda che il metodo risolutivo scelto lo si voglia precondizionare o meno;
- per la risoluzione di sistemi non linari costruisce l'oggetto *SNES* che include in sé un oggetto *KSP* per la risoluzione del sistema linearizzato con il metodo che caratterizza lo *SNES*.

Come già è stato anticipato, la libreria è stata sviluppata seguendo i principi della programmazione ad oggetti: ogni oggetto PETSc ha un suo stato, e fornisce una serie di operazioni ("metodi" e "variabili di istanza") che lo caratterizzano. Nella definizione di ciascun oggetto, l'attenzione non è rivolta solo ai dati che deve contenere, ma anche alle operazioni che devono essere eseguite sull'oggetto stesso.

È noto che, tra le caratteristiche più importanti di un linguaggio OO, c'è l'ereditarietà, ovvero la capacità di definire sottoclassi di una classe, che non solo ereditano le caratteristiche di quest'ultima ma le espandono, per costruire classi più specializzate.

Molti oggetti di PETSc hanno sottoclassi che raffinano le superclassi. A esempio, il KSP può essere visto come una classe astratta che è specializzata da diverse sottoclassi che implementano alcuni metodi di risoluzione di sistemi lineari. Ad esempio una sottoclasse dell'oggetto KSP è l'oggetto KSPCG che esegue il metodo del gradiente coniugato.

Il KSP può essere visto come una classe astratta che è specializzata dalle diverse sottoclassi che implementano i metodi di risoluzione di sistemi lineari (CG, GMRES, ...) che possono essere, a loro volta, meglio caratterizzati mediante la scelta di precondizionatori (PC) e tipologie di matrici (MAT).

Gli sviluppatori della libreria hanno realizzato un modo ingegnoso per simulare l'ereditarietà, il conseguente polimorfismo e il relativo binding dinamico: l'utilizzatore della libreria, nel caso dell'oggetto KSP, seleziona, grazie alla funzione KSPSetType, il tipo di sottoclasse che vuole utilizzare, dopodichè ogni funzione dell'oggetto che viene richiamata, automaticamente esegue la parte di codice relativa alla sottoclasse dell'oggetto KSP.

3.4 Il caso di studio: il gradiente coniugato parallelo di PETSc

Come descritto nella sezione 3.3, la libreria PETSc dispone di numerosi risolutori, per i sistemi lineari.

In questo lavoro si è preso in considerazione il Il gradiente coniugato (CG) un metodo iterativo [26] utilizzato per la risoluzione di sistemi lineari nella forma

$$Ax = b$$

dove A è una matrice simmetrica definita positiva. Si tratta di una soluzione efficiente particolarmente adatta nel caso di matrici sparse con determinate caratteristiche, che difatti viene utilizzato per la risoluzione di sistemi associati all'approssimazione numerica di equazioni alle derivate parziali ma risulta un buon algoritmo anche per la risoluzione di sistemi lineari densi poichè ha complessità computazionale $O(n^2)$.

Osservando lo schema dell'algoritmo iterativo del CG (Algoritmo 1), ogni iterazione è composta dalle seguenti operazioni di algebra lineare di base:

- due prodotti scalari con complessità O(n);

- tre operazioni di tipo saxpy[13] con complessità $O(n)$;
- un prodotto matrice vettore con complessità $O(\alpha n)$.

Si osserva, attraverso i dati appena forniti, che il nucleo computazionale più costoso è l'aggiornamento del vettore $q(k)$ e che tale algoritmo converge al più in n passi, quindi la complessità totale è $O(n^2)$.

La necessità di ridurre il tempo al minimo possibile ha condotto quindi verso lo sviluppo del software in ambiente di calcolo parallelo.

L'introduzione del parallelismo, all'interno della libreria PETSc, è stata rivolta principalmente a ridurre i costi computazionali delle operazioni più onerose dell'algoritmo illustrato, preservando la filosofia modulare che lo caratterizza in maniera da operare modifiche trasparenti all'utente.

La versione parallela del CG, in PETSc, viene in tal modo ad essere un altro oggetto della libreria caratterizzabile in modo semplice da parte dell'utente così come lo sono gli altri oggetti più semplici.

L'algoritmo 2 mostra un frammento di codice di un programma che risolve un sistema lineare utilizzando l'algoritmo del Gradiente Coniugato di PETSc:

- mediante le linee 11 e 14 il programmatore crea l'oggetto KSP e lo caratterizza con la scelta del risolutore (*"cg"*)
- con la linea 17 vengono fissati il tipo di matrice e il precondizionatore del metodo
- alla linea 21 viene risolto il sistema lineare (di matrice kspmg e vettore dei termini noti b), mediante il metodo KSPSolve

L'algoritmo 2, inoltre, evidenzia l'assenza di istruzioni relative

[13]operazione che consiste nell'aggiornamento di un vettore, mediante somma con un altro vettore moltiplicato per uno scalare α, cioè $y = y + \alpha \star x$

Algoritmo 1 Template di Netlib del Gradiente Coniugato

```
Initial r^(0) = b − Ax^(0)
for do i = 1, 2, ...
    solve Mz^(i−1) = r^(i−1)
    ϱ_{i−1} = r^(i−1)^T z^(i−1)
    if i = 1 then
        p^(1) = z^(0)
    else
        β_(i−1) = ϱ_{i−1}/ϱ_{i−2}
        p^(i) = z^(i−1) + β_(i−1) p^(i−1)
    end if
    q^(i) = Ap^(i)
    α_i = ϱ_{i−1}/p^(i)^T q^(i)
    x^(i) = x^(i−1) + α_i p^(i)
    r^(i) = r^(i−1) − α_i q^(i)
    check convergence;
end for
```

Algoritmo 2 Utilizzo del Gradiente Coniugato della libreria PETSc.

```
/* Inizializzazione dell'ambiente PETSc
PetscInitialize(&Argc, &Args, (char*)0, help);
...
...
/* Inizializzazione della matrice dei coefficienti
...
...
/* Inizializzazione del vettore dei termini noti
...
/* Creazione dell'oggetto KSP */
KSPCreate(PETSC_COMM_WORLD, &ksp);
...
/* Scelta dell'implementazione del Gradiente Coniugato */
KSPSetType(ksp, "cg");
...
/* Configurazione della matrice e del precondizionatore del metodo */
KSPSetOperators(kspmg, cmat, cmat,
                        DIFFERENT_NONZERO_PATTERN);
...
/* Risuluzione del sistema lineare */
KSPSolve(kspmg, b, x);
...
...
```

alla libreria di Message Passing, poichè tutte le direttive di comunicazione sono nascoste all'utente della libreria.

3.5 L'obiettivo della ricerca

Il caso di studio di questo lavoro di ricerca è stato il metodo iterativo del gradiente coniugato della libreria PETSc; il fine ultimo è quello di realizzare ed implementare una metodologia di checkpointing che, attraverso l'utilizzo di tecnologie e strumenti standard, renda fault tolerant l'algoritmo del Gradiente Coniugato della libreria PETSc.

Il fatto di lavorare all'interno delle librerie scientifiche fornisce un grande vantaggio: la realizzazione dei meccanismi per la fault tolernace all'interno degli algoritmi di base fa si che ogni applicazione, che faccia uso di tali algoritmi, erediti automaticamente le caratteristiche di tolleranza ai guasti implementate a livello più basso.

Questo, se da un lato richiede uno sforzo maggiore nella comprensione dei moduli di calcolo di più basso livello, dall'altro invece si traduce, alla lunga, in un significativo risparmio di tempo.

Le applicazioni che utilizzano algoritmi di librerie fault tolerant infatti, senza ulteriori modifiche, sono in grado di superare i fault e proseguire l'esecuzione migrando automaticamente su risorse alternative locali o remote (se non ce ne sono più localmente).

Nel prossimo capitolo saranno illustrate le strategie adoperate per abilitare la *fault tolerance* all'interno di un modulo della libreria PETSc, mediante meccanismi di checkpointing *algorthm-based.*

Oltre al fatto che un checkpointing algorithm-based comporta la conoscenza del codice dell'algoritmo, un'altra difficoltà è senz'altro dovuta al fatto di dover garantire che l'implementazione di tali meccanismi non comporti problemi di instabilità per la libreria stessa.

La struttura della libreria PETSc è molto complessa, come è stato descritto precedentemente, la simulazione di alcuni meccanismi della programmazione Object Oriented comporta alcune difficoltà legate alla stretta relazione esistente tra oggetti che appartengono i livelli di astrazione differenti [14].

[14] L'oggetto KSP, in particolare, è collocato agli alti vertici di tale gerarchia ed è, quindi, in stretta relazione con tutti i moduli sottostanti.

Capitolo 4

Progettazione e realizzazione della strategia ibrida di checkpointing

4.1 Alcune considerazioni preliminari

Da quanto affermato durante la panoramica sulle metodologie di checkpointing del *Capitolo 2*, risulta chiaro che la progettazione di una nuova strategia di checkpointing e la modifica di strategie già esistenti mirano a ridurre l'overhead introdotto dalle fasi di salvataggio/rolling back.

L'overhead delle fasi di recovery è legato non solo alla velocità di accesso in scrittura/lettura del supporto di memorizzazione (disco, memoria ram, storage remoto, ...) ma anche alla quantità di dati di checkpoint.

Relativamente ai metodi di Krylov, in [23], sono riportate alcune sperimentazioni, sull'utilizzo dell'approccio denominato *partial* *checkpointing*[1], finalizzate alla riduzione dei dati da salvare ad ogni iterazione.

Nel suddetto lavoro, il *partial checkpointing* è stato implementato per gli algoritmi CG, GMRES e le loro versioni "flessibili" [2]. Dai test effettuati su tutti i metodi, il residuo [3], in fase di rolling back, è risultato meno vicino alla soluzione di quanto non fosse prima del fault portando ad un rallentamento nella convergenza del metodo e rendendo, di fatto, addirittura più conveniente la ripartenza dell'esecuzione da principio piuttosto che dall'ultimo checkpoint salvato.

L'utilizzo di una strategia algorithm-based, sebbene onerosa da implementare, rimane il metodo più sicuro per selezionare e, di conseguenza ridurre, i dati di checkpointing.

Inoltre, già da qualche anno, la comunità scientifica si è interessata ai vantaggi, in termini di robustezza, che possono derivare dall'utilizzo di più tecniche di checkpointing combinate (strategie *ibride*) ed un esempio di implementazione di tale strategia è riportato in [35].

È quest'ultima la strada che si è scelto di seguire durante questo lavoro di ricerca.

Nei prossimi paragrafi saranno illustrate le modalità con le quali si è progettata la strategia ibrida per il CG di PETSc, soffermando l'attenzione su quali siano state le scelte strategiche e quali quelle obbligate.

[1] cenni alla strategia

[2] in grado di modificare durante l'esecuzione il precondizionatore utilizzato

[3] Per un sistema lineare del tipo $Ax = b$, al passo k del metodo iterativo del CG, la soluzione calcolata è $x_{(k)}$ e il residuo è: $r_{(k)} = b - Ax_{(k)}$

4.2 Descrizione della strategia ibrida

Sulla base delle considerazione appena riportate e non potendo implementare il checkpointing algorithm-based in maniera diskless, a causa dell'assenza di una libreria di Message Passing fault tolerant stabile (come riportato nella sezione 2.3), il checkpointing è stato implementato, all'interno del Gradiente Coniugato di PETSc, utilizzando un approccio algorithm-based e disk-based.

Nella prossima sezione sarà illustrata la prima delle due strategie che confluiranno nella strategia ibrida che è stata implementata.

4.2.1 Il checkpointing disk-based codificato

Il checkpointing disk-based codificato nasce dall'idea di codifica che, solitamente, è alla base degli approcci diskless *Parity-based* (paragrafo 2.2). Non potendo implementare il checkpointing algorithm-based in maniera diskless[4], si è pensato di ottimizzare le prestazioni dell'implementazione disk-based, utilizzando lo schema RAID level 5[5] dell'implementazione diskless.

L'approccio disk-based, notoriamente più inefficiente di quello diskless, migliora le sue prestazioni grazie alla metodologia algorithm-

[4]Alla fine del *Capitolo 2* sono state riportate le motivazioni che hanno portato ad escludere la libreria FT_MPI come implementazione per lo svolgimento del presente lavoro.

[5]Lo schema di parità o *RAID Level 5* fa uso di un checkpoint processor che codifica la *bitwise parity.* La codifica è realizzata mediante l'utilizzo dell'operatore XOR dei dati di ogni checkpoint processor.

In altri termini, si indichi con b_i^j il j-esimo byte dell'i-esimo application processor. Il j-esimo byte dei dati del checkpoint processor sarà:

$$b_{ckp}^j = b_1^j \oplus b_2^j \oplus \ldots \oplus b_n^j$$

In caso di fault di un qualsiasi application processor, un processo sostituto prende il posto del processo fallito. Il nuovo processore calcola i dati di checkpoint del processo fallito utilizzando i checkpoint degli application processor "sopravvissuti" e del checkpoint processor. Se il fault è subito dal i-esimo processo, il checkpoint è ricostruito nel modo seguente:

$$b_i^j = b_{ckp}^j \oplus b_1^j \oplus b_2^j \oplus \ldots \oplus b_{i-1}^j \oplus b_{i+1}^j \ldots \oplus b_n^j$$

Una volta che il nuovo processore ha calcolato il checkpoint del processore che ha rimpiazzato, gli altri application processor eseguono un rolling back dell'ultimo checkpoint, a questo punto l'esecuzione può ricominciare.

based, che richiede l'individuazione, nell'algoritmo, tutti e soli i dati necessari all'applicazione per essere ripristinata a seguito di un fault.

Algoritmo 3 Il Gradiente Coniugato

```
CG(A, b, X)
/*operazioni di inizializzazione */
R = b; /*copia di B in R */
repeat
  beta = Z * R;
  if beta == 0.0 then
    printf(Convergenza dovuta a beta =0);
    break;
  else if beta < 0.0 then
    printf(Divergenza dovuta ad un precondizionatore indefinito);
    break;
  end if
  P = Z + b * P;
  b = beta/betaold;
  betaold = beta;
  Z = A * P;
  dpi = Z * P;
  if dpi <= 0.0 then
    printf(Divergenza dovuta a matrice indefinita o definita negativa);
    break;
  end if
  a = beta/dpi;
  X = X + a * P;
  R = R - a * Z;
  dp = Norma(Z);
  i = i + 1;
until (i < max_it && r > tol)
```

Nel caso in esame, riprendendo il metodo del Gradiente Coniugato, e focalizzando l'attenzione sul ciclo di calcolo *do-while* (cfr. Algoritmo 3) si osserva che i dati necessari per il checkpointing sono costituiti da quattro vettori distribuiti: *X, Z, R, P* e da quattro scalari: *i, betaold, a e reason.*

La fase di checkpoint può essere divisa in due sottofasi (fig. 4.1):

1. Ogni application processor salva localmente i propri dati di checkpoint e li invia al checkpoint processor.

2. Il checkpoint processor calcola lo XOR-bitwise dei dati ricevuti dagli application processor e li salva nella propria memoria di massa[6].

La fase di recupero dell'applicazione (rolling back) si articola anch'essa in due sottofasi (fig. 4.2):

1. I processori sopravvissuti recuperano i propri dati localamente.
2. Il processore che ha preso il posto del processore guasto, grazie ai dati locali degli altri processori e ai dati codificati del checkpoint processor, ricostruisce la parte di dati che gli spetta.

4.2.2 Osservazioni

La strategia appena descritta è robusta in quanto garantisce l'integrità del checkpointing (il checkpoint relativo all'iterazione k non viene eliminato finchè non è completata la fase di salvataggio dati relativa all'iterazione $k + 1$); inoltre l'applicazione è in grado di sopravvivere sia al fault di un application processor, sia a quello del checkpoint processor.

In generale, l'utilizzo di una codifica offre un vantaggio: la quantità di dati che il checkpoint processor deve immagazzinare è drasticamente ridotta: il checkpoint processor non salva i vettori globali, ricostruendoli a partire dai vettori parziali che ciascun application processor gli invia (come invece accade per la strategia coordinata), ma effettua la codifica XOR-bitwise dei vettori degli application processor, che hanno dimensione N/p [7], ottenendo dati codificati di lunghezza N/p.

[6] Poichè l'approccio è disk-based, i checkpoint sono salvati sul disco locale degli application processor nella directory *workname*

[7] con N dimensione del problema e p numero di processori.

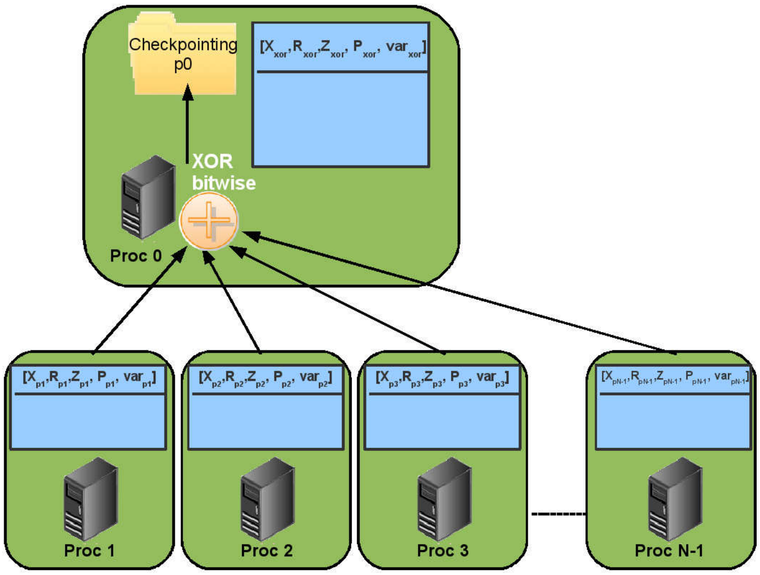

Figura 4.1: Prima strategia di checkpointing: codificato, fase di salvataggio dei dati

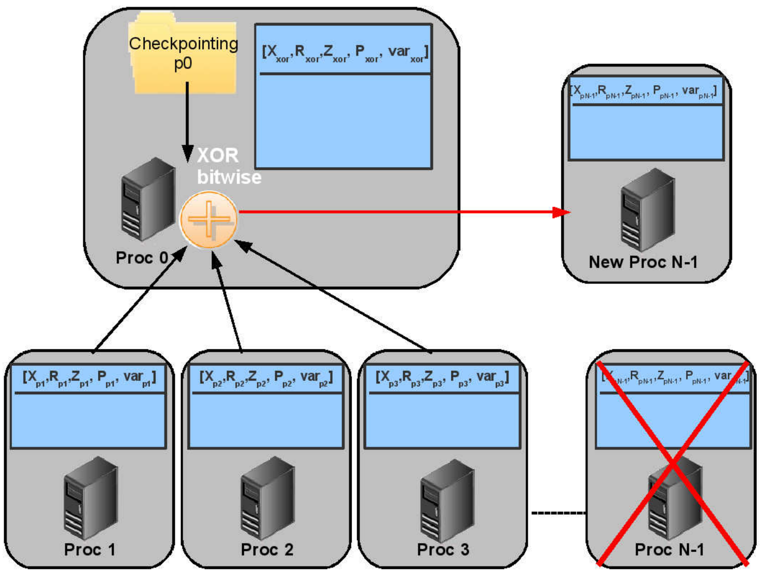

Figura 4.2: Prima strategia di checkpointing: codificato, fase di recupero dei dati

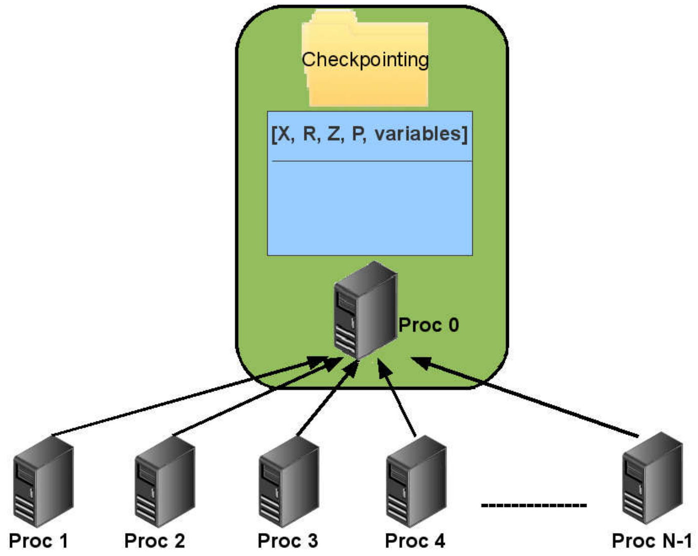

Figura 4.3: Seconda strategia di checkpointing: coordinato, fase di salvataggio dei dati

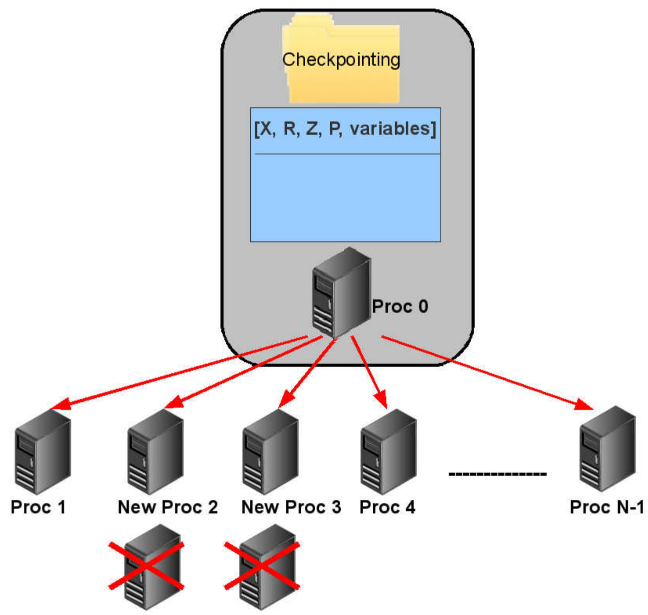

Figura 4.4: Seconda strategia di checkpointing: coordinato, fase di recovery dei dati a seguito di due fault contemporanei.

È chiaro che questa strategia è ottimizzata, in termini di overhead di I/O, rispetto a quelle senza codifica, ma è in grado di tollerare un solo fault per volta. Per questa ragione si è pensato di aggiungere alla strategia descritta nella precedente sezione, anche sporadiche fasi di checkpointing coordinato, senza per questo aumentare eccessivamente l'overhead di checkpointing.

In figg. 4.3 e 4.4 sono mostrate rispettivamente le fasi di checkpointing e di rolling back della strategia coordinata di checkpointing.

4.2.3 La versione ibrida di checkpointing

Due strategie di base, completamente indipendenti e adottabili singolarmente, possono, insieme, dar luogo ad un'unica, nuova implementazione che le contenga: la strategia ibrida. Nel combinare le strategie di checkpointing, è necessario tuttavia bilanciare la necessità di robustezza con l'aumento di overhead.

La frequenza con la quale vengono salvati i dati di checkpointing su disco, non può essere eccessiva e, in ogni caso i tempi di salvataggio dei dati, globalmente, non devono essere rilevanti rispetto al tempo totale di esecuzione dell'applicazione che usa il metodo iterativo.

Per tale ragione, la strategia ibrida è stata implementata utilizzando un meccanismo automatico di scelta della frequenza di checkpointing rispetto ai tempi medi di esecuzione delle iterazioni del CG.

L'applicazione che utilizza il CG, dapprima viene eseguita utilizzando le frequenze (per il checkponting codificato e per quello coordinato) scelte dall'utente (ad esempio ogni iterazione esegue il checkpointing codificato ed ogni “k” quello coordinato); dopo la prima iterazione, a runtime, l'algoritmo richiama un modulo che si occupa di rettificare la frequenza dei checkpointing sulla base del tempo di esecuzione delle iterazioni.

Per quanto riguarda la fase di ripristino, invece, l'algoritmo determina e sceglie, tra le due tipologie di checkpointing, quella che permette di ripristinare l'applicazione dall'iterazione più alta.

E' importante notare che oltre a controllare quale dei due checkpoint è il più recente, l'algoritmo verifica quale sia il checkpoint più conveniente da utilizzare nella fase di rolling back.

Possiamo infatti immaginare il seguente scenario di esecuzione: supponiamo che l'applicazione utilizzi p processori ed effettui un checkpoint coordinato ogni 10 iterazioni e un checkpointing di tipo codificato ad ogni iterazione; supponiamo che si verifichino due fault contemporanei, alla 45-sima iterazione.

Nella fase di rolling back, successiva al fault, sono presenti due checkpoint: uno corrispondente alla 40-esima iterazione (relativo all'approccio coordinato) e l'altro relativo alla 44-sima iterazione (approccio codificato).

Poichè il fault riguarda due nodi, il checkpoint codificato non è utilizzabile, quindi l'applicazione sarà ripristinata automaticamente a partire dalla 40-sima iterazione, utilizzando i dati relativi al checkpoint coordinato.

Nel caso di singoli fault, invece, l'applicazione sceglierà automaticamente l'ultimo checkpoint effettuato (coordinato o codificato che sia).

Il vantaggio introdotto con la strategia ibrida è che l'applicazione può tollerare fino ad $p-1$ fault contemporanei, purchè uno dei fault non riguardi il nodo di checkpoint (coordinato).

Nella prossima sezione viene descritto il lavoro svolto, all'interno del codice del gradiente coniugato, al fine di implementare, utilizzando un approccio algorithm-based, la strategia di checkpointing ibrida appena esposta.

Nel prossimo capitolo saranno, invece, illustrate alcune ulteriori funzionalità, implementate a supporto delle applicazioni CG-based, con l'obiettivo di:

1. irrobustire ulteriormente la strategia ibrida in grado di tollerare il fault dell'intero Cluster che esegue l'applicazione (mediante l'utilizzo di quella che sarà definita "variante distribuita asincrona del checkpointing ibrido")

2. rendere possibile, in caso di fault dell'intero Cluster di esecuzione, la "migrazione" dell'applicazione CG-based, su risorse di calcolo alternative, fornite dal sistema informativo dell'ambiente distribuito.

4.3 La nuova versione del Gradiente Coniugato di PETSc.

Al fine di implementare i meccanismi descritti nelle precedenti sezioni del presente Capitolo, è stato dapprima creato un clone dell'oggetto KSP_CG (che è stato chiamato KSP_CGFT) e, nel codice di quest'ultimo, è stata modificata la routine **KSPSolve_CGFT**, mediante l'aggiunta delle funzionalità di Rolling back e Checkpointing. Nell'Algoritmo 4, sono riportate le prime modifiche apportate al codice dell'algoritmo del CG di PETSc per abilitare le suddette funzionalità:

- Per i meccanismi di checkpointing:
  - (riga 31) alle iterazioni fissate per eseguire il checkpoint di tipo coordinato viene richiamata la funzione PetscCheckpointingCoord)...)
  - (riga 34) alle iterazioni fissate per eseguire il checkpoint di tipo codificato viene richiamata la funzione PetscCheckpointingCodif(...)

Algoritmo 4 Frammento di codice del Gradiente Coniugato di PETSc: implementazione fault tolerant

```
PetscErrorCode KSPSolve_CGFT(KSP ksp)
PetscFunctionBegin;
/* operazioni di inizializzazione */
...
maxit = ksp->max_it;
if (restart) then
  rt = CheckCheckpoint(ksp, &restart, &start, workname, idg,
          check_proc);
  if (rt == 1) then
    ierr = PetscRollbackCoord(ksp, Z, R, X, P, &a, &betaold, &i,
            workname, ksp->num_call);
  else if (rt == 0) then
    ierr = PetscRollbackCodif(ksp, Z, R, X, P, &a, &betaold, &i,
            &ksp->reason, &Zck, &Rck, &Xck, &Pck, &ack,
            &betaoldck, &ick, &reasonck, workname, DB);
  else
    printf(Non è possibile recuperare dal fault);
  end if
end if
repeat
  ...
  /* istruzioni del ciclo repeat-until dell'Algoritmo 3 */
  ...
  if (chkenable) then
    /* ck_coord è l'iterazione a cui viene
    effettuato il checkpointing coordinato*/
    /* ck_codif è l'iterazione a cui viene
    effettuato il checkpointing codificato */
    if (i % ck_coord == 0) then
      ierr = PetscCheckpointingCoord(ksp, Z, R, X, P, a,
              betaold, i, workname, ksp->num_call);
    else/* caso i % ck_codif == 0 */
      ierr = PetscCheckpointingCodif(ksp, Z, R, X, P, a,
              betaold, i, ksp->reason, &Zck, &Rck, &Xck,
              &Pck, &ack, &betaoldck, &ick, &reasonck,
              workname, DB);
    end if
  end if

until (i < max_it && r > tol)
ierr = PetscCheckFreq(iteration_Time, checkCoord_Time,
        checkCodif_Time, &ck_coord, &ck_codif)
/* operazioni di finalizzazione */
PetscFunctionReturn(0);
```

  - (riga 41) dopo ogni checkpoint, ne viene rettificata la frequenza (sulla base del tempo di esecuzione del salvataggio dei dati di checkpointing rispetto al tempo di esecuzione delle iterazioni).

- Per i meccanismi di rolling back:

  - (riga 8) viene richiamata la funzione che controlla quale tipo di checkpoint è possibile utilizzare per la fase di recovery da un fault
  - (riga 11) se il checkpoint più recente è di tipo coordinato allora viene richiamata la funzione PetscRollbackCoord(...)
  - (riga 14) se il checkpoint più recente è di tipo codificato allora viene richiamata la funzione PetscRollbackCodif(...)

Le funzioni PetscRollbackCoord(...) PetscRollbackCodif(...) seguono gli schemi descritti nelle figure 4.4, 4.2, mentre le funzioni PetscCheckpointingCoord(...) e PetscCheckpointingCodif(...) seguono gli schemi descritti nelle figure 4.3, 4.1.

Capitolo 5

Il sistema di checkpointing/migration in ambiente GRID

L'ambiente distribuito, con la grande quantità di risorse di calcolo e di servizi replicati, può essere di supporto all'ambiente HPC.

In particolare i meccanismi descritti nel *Capitolo 4* abilitano l'applicazione a sopravvivere al fault di uno o più nodi di calcolo, ma non al guasto (o indisponibilità momentanea) dell'intero cluster su cui essa è in esecuzione.

Nelle prossime sezioni, dapprima è descritta una variante della metodologia di checkpointing ibrida realizzata, poi sono illustrati i servizi di monitoraggio e migrazione che supportano l'esecuzione dell'applicazione basata sul CG e ne predispongono l'eventuale fase di migrazione a seguito dei fault.

Il sistema completo di checkpointing/migration è stato implementato per garantire, all'applicazione, la sopravvivenza ai guasti in differenti scenari di esecuzione:

- **in ambiente di calcolo parallelo:** dovrà garantire la sopravvivenza delle applicazioni ai guasti di un certo numero di nodi:
  - continuando l'esecuzione sui rimanenti nodi del contesto della macchina parallela
  - salvando i dati localmente e terminando l'esecuzione nel caso non siano più disponibili nodi per continuare.
- **in ambiente distribuito:** dovrà garantire la sopravvivenza delle applicazioni ai guasti di tutti i nodi del cluster:
  - salvando i dati in remoto, terminando l'esecuzione se non ci sono nodi per continuare o
  - migrando l'applicazione su risorse alternative per proseguire l'esecuzione.

5.1 La variante distribuita del checkpointing ibrido

Nel `Capitolo 4` si è discusso dei benefici derivanti dall'utilizzo di una strategia ibrida di checkpointing che consente di sfruttare i vantaggi delle strategie componenti:

- si paga solo un piccolo prezzo in termini di overhead totale (a causa della strategia coordinata che è meno efficiente di quella codificata)
- si acquista in termini di robustezza del checkpointing, per cui l'applicazione è in grado di tollerare fino a *n-1* fault contemporanei.

Tuttavia, può accadere che l'intero cluster, su cui l'applicazione è in esecuzione, diventi non disponibile. In tale circostanza, la strategia ibrida descritta non è in grado di recuperare lo stato dell'applicazione, non essendo raggiungibili i dati di checkpointing (nè quelli relativi alla strategia coordinata, nè quelli relativi alla strategia codificata).

Inoltre, nel caso di cluster con i quali è possibile interagire solo mediante scheduler per sottomettere i job, non è detto che, al momento del rolling back, il set di nodi selezionati dallo scheduler del cluster, per continuare l'esecuzione dell'applicazione, sia lo stesso (a parte i nodi che hanno subito il fault).

Se il cluster ha un file system *shared* allora il problema non si presenta in quanto, per il salvataggio dei dati di checkpointing, può essere scelta proprio una directory dell'area condivisa. In caso contrario, i dati di checkpointing codificato non sono utilizzabili.

Di qui l'idea che ha portato alla realizzazione di una variante distribuita *asincrona* della strategia ibrida che consiste nel:

- salvatare i dati di checkpointing coordinato su una risorsa di storage esterna al cluster
- raccogliere e salvare i dati di checkpointing codificato su una risorsa di storage esterna al cluster

I dati di checkpointing relativi all'iterazione $i - esima$ vengono rimossi non più quando è completato il checkpointing locale successivo, come descritto nel precedente Capitolo, ma solo dopo che i dati di checkpointing sono stati trasferiti sullo storage remoto.

Per la realizzazione della variante distribuita del checkpointing ibrido, sono stati utilizzati 4 *thread* (uno per ogni vettore dei dati di checkpointing) in modo da non aumentare l'overhead dell'applicazione.

La versione definitiva della strategia ibrida realizzata opera dunque così:

- dopo la prima iterazione, e dopo l'esecuzione di un checkpointing (codificato o coordinato) viene attivato il modulo che rettifica periodicamente la frequenza dei checkpointing sulla base delle prestazioni di esecuzione dell'applicazione [1].

- a partire dall'iterazione successiva a quella designata per il checkpointing coordinato, mediante l'utilizzo della funzione *PetscStartCopyThreads* (cfr. riga 26 dell'algoritmo 5)vengono abilitati 4 thread che si occupano di effettuare la copia dei 4 vettori (dati di checkpointing coordinato) sullo storage esterno al cluster di esecuzione.

- a partire dall'iterazione successiva a quella designata per il checkpointing codificato, mediante l'utilizzo della funzione **PetscStartCollectThreads** (cfr. riga 29 dell'algoritmo 5) vengono raccolti i dati di checkpointing codificato di tutti i nodi che concorrono all'esecuzione e copiati sullo storage esterno al cluster di esecuzione. Questa funzione deve essere esplicitamente abilitata al momento dell'esecuzione sulla base delle informazioni relative alla configurazione *shared* o meno dell'area disco del cluster.

- in caso di rolling back, procede come la strategia ibrida di base, poichè prevede di avere già i dati in locale, delegando al modulo di migrazione il compito di compiere eventuali trasferimenti di dati da risorse remote ai nodi di calcolo.

La funzione **PetscStartCopyThreads** utilizza i comandi di copia e replica dei dati messi a disposizione dal middleware di GRID per spostare e rendere successivamente accessibili, mediante nomi simbolici, i dati di checkpointing in fase di rolling back remoto.

[1] il modulo di rettifica di basa sul tempo di esecuzione di un checkpointing coordinato e codificato confrontato con il tempo medio per l'esecuzione di un'iterazione del CG

Algoritmo 5 Frammento di codice del Gradiente Coniugato fault tolerant con checkpointing ibrido adattivo distribuito

```
PetscErrorCode KSPSolve_CGFT(KSP ksp)
PetscFunctionBegin;
/* operazioni di inizializzazione */
...
if (restart) then
   rt = CheckCheckpoint(...);
   if (rt == 1) then
      ierr = PetscRollbackCoord(...);
   else if (rt == 0) then
      ierr = PetscRollbackCodif(...);
   else
      printf(Non è possibile recuperare dal fault);
   end if
end if
repeat
   ...
   /* istruzioni del ciclo repeat-until dell'Algoritmo 3 */
   ...
   if (chkenable) then
      /* ck_coord è l'iterazione a cui viene
      effettuato il checkpointing coordinato*/
      /* ck_codif è l'iterazione a cui viene
      effettuato il checkpointing codificato */
      if (i % ck_coord == 0) then
         ierr = PetscCheckpointingCoord(...);
         ierr = PetscStartCopyThreads(...);
      else/* caso i % ck_codif == 0 */
         ierr = PetscCheckpointingCodif(...);
         ierr = PetscStartCollectThreads(...);
      end if
   end if

until (i < max_it && r > tol)
ierr = PetscCheckFreq(...);
/* operazioni di finalizzazione */
PetscFunctionReturn(0);
```

Mentre il lavoro descritto nel *Capitolo 4* ha portato alla realizzazione di un sistema di checkpointing/migrazion locale al cluster su cui l'applicazione è in esecuzione, la variante distribuita consente di tollerare anche il guasto dell'intero cluster e di effettuare la migrazione dell'esecuzione su risorse di calcolo remote fornite dall'ambiente distribuito.

5.2 I servizi di monitoring/migration

Rispetto agli scenari di esecuzione citati l'applicazione, che utilizza l'algoritmo del Gradiente Coniugato di PETSc, viene supportata da due moduli software: il *Monitor* e il *Migrator*.

In corso di esecuzione:

- il Monitor preleva le informazioni circa eventuali fault e i dati di esecuzione dell'applicazione.
- in caso di fault, di uno o più nodi di calcolo, si attiva il meccanismo di migrazione locale o remota:
  - Caso locale:
    - il Monitor verifica la possibilità di continuare l'esecuzione dell'applicazione e contatta il Migrator
    - Il Migrator ferma l'applicazione salvandone lo stato, recluta, sullo stesso cluster, il certo numero di nodi sostitutivi e rilancia l'applicazione dal punto un cui si era fermata, restituendo lo *status* al Monitor.
  - Caso remoto:
    - il Monitor chiede, al servizio informativo dell'ambiente distribuito, i dati relativi alla presenza di altre risorse, in grado di proseguire l'applicazione corrente.
    - il Migrator effettua il processo di migrazione remota e restituisce lo status al Monitor.

Nel seguito di questa sezione saranno riportati dettagli sulle componenti del sistema distribuito appena citate.

5.2.1 Il monitor

Nella sezione 3.1 abbiamo discusso dell'architettura a livelli dell'ambiente distribuito distinguendo tra punto di accesso alla GRID, servizi di gestione e servizi di calcolo e storage. Nella presente sezione e in quella successiva, verrà descritto come questi tre livelli interagiscono tra loro per dar vita al sistema di monitoraggio dell'applicazione basata sul CG.

Il coordinamento dell'applicazione avviene dall'esterno del Cluster mediante lo scambio di informazioni via socket [2] tra un client (in esecuzione sul Cluster insieme all'applicazione) e un server in esecuzione sulla UI (il monitor). Il monitor si avvia in concomitanza della richiesta di esecuzione dell'applicazione su uno dei cluster dell'ambiente GRID e rimane in ascolto di uno (o più) client che gli invieranno informazioni relative all'esecuzione dell'applicazione monitorata.

Ad ogni iterazione vengono inviate, dal client al monitor, le seguenti informazioni:

- numero di iterazione corrente e tempo di esecuzione dell'iterazione passata

[2] È il punto in cui il codice applicativo di un processo accede al canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate. Dal punto di vista di un programmatore, un socket è un particolare oggetto sul quale leggere e scrivere i dati da trasmettere o ricevere. Ci sono due tipi fondamentali di socket:

- i socket tradizionali su protocollo IP, usati in molti sistemi operativi per le comunicazioni attraverso un protocollo di trasporto (TCP o UDP);
- gli Unix domain socket, usati nei sistemi operativi POSIX per le comunicazioni tra processi residenti sullo stesso computer.

- tipo di checkpointing effettuato (codificato, coordinato, no-checkpointing) e
- eventuale messaggio di fault

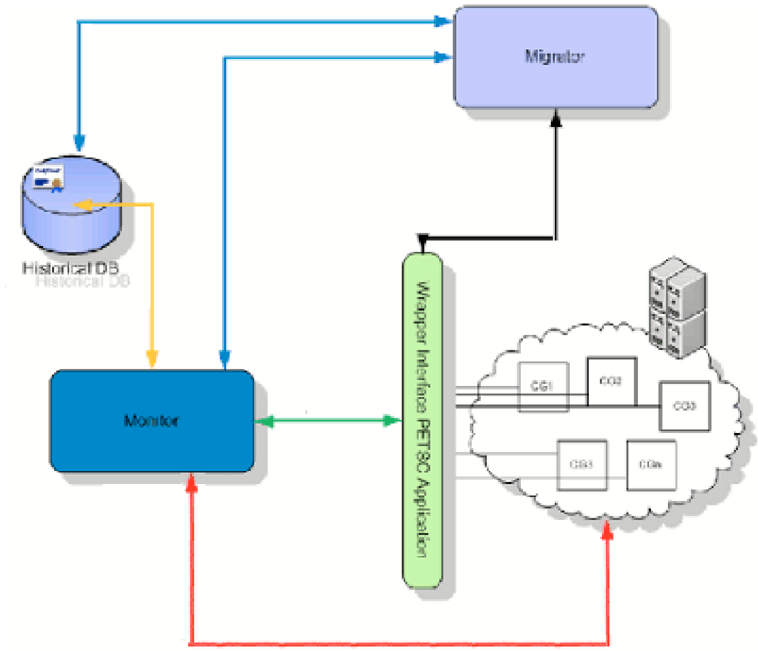

Figura 5.1: Schema di esecuzione del sistema di checkpointing/migration

In figura 5.1 è mostrato lo schema di interazione tra i componenti del sistema di checkpointing/migration.

5.2.2 Il migrator: esecuzione dell'applicazione basata sul CG in ambiente GRID.

Come discusso nella precedente sezione, il monitor rimane in ascolto per tutta la durata dell'applicazione, osservandone il comportamento e, in caso di eventi inattesi, occupandosi della comunicazione dell'evento al modulo software preposto alla *decision policy*: il migrator, il cui schema esecutivo prevede:

- alla prima esecuzione:
  - genera uno script descrittore del job utilizzando la sintassi JDL[3] del tipo riportato (cfr. codice Algoritmo 6)
  - esegue il monitor
  - manda in esecuzione l'applicazione utilizzando le funzioni della libreria di gLite
  - rimane in ascolto di messaggi da parte del monitor.
- in caso di fault dell'applicazione:
  - controlla, mediante il sistema informativo, la disponibilità di risorse alternative per la ripresa dell'applicazione dal punto in cui si è verificato il fault
  - se c'è disponibilità, provvede a generare un nuovo file JDL che indichi, tra l'altro, la risorsa di storage che possiede i dati di checkpointing.
  - riprende l'esecuzione dell'applicazione sul nuovo set di risorse.

[3] JDL è l'acronimo di Job Description Language; è la sintassi utilizzata dal middleware gLite per la caratterizzazione dei job da sottomettere sulla GRID

Algoritmo 6 Esempio di file JDL per l'esecuzione dell'applicazione in ambiente GRID

```
Type =" Job";
JobType =" MPICH";
CpuNumber = 80;
Executable =" cgft.sh";
Arguments =" MVAPICH 15000 10 100 40 1 30 1 local";
StdOutput =" 15000 − 10 − local.out";
StdError =" 15000 − 10 − local.err";
InputSandbox = "cgft.sh"," mpi − hooks.sh";
OutputSandbox = "15000 − 10 − local.err"," 15000 − 10 − local.out";
Requirements = other.GlueCEInfoTotalCPUs > 600
        &&Member("MPI − START",
        other.GlueHostApplicationSoftwareRunTimeEnvironment)
        &&Member("MVAPICH",
        other.GlueHostApplicationSoftwareRunTimeEnvironment);
RetryCount = 1;
```

Algoritmo 7 Esempio di script per l'esecuzione dell'applicazione

```
#/bin/bash
# Esempio di script cgft.sh
SERVER =" ui01.scope.unina.it"
PORT = 1745
work = `pwd`/tmp
EXE = /opt/exp_soft/unina.it/TESTs/PETSC_FT/TEST/cgft
MPI_FLAVOR = $1
DIM =$2
NP =$3
MAXIT = $4
STOP = $5
MAX_FAULT = $6
FR_COORD = $7
FR_CODIF = $8
FS =$9
HOST_NODEFILE = `sort − u $PBS_NODEFILE`
outfile = $DIM − $NP − $FS.out
eval MPI_PATH = `printenv MPI_${MPI_FLAVOR}_PATH`
$MPI_PATH/bin/mpirun − np $NP − machinefile
        $HOST_NODEFILE $EXE − ck_coo_fr $FR_COORD
        −ck_cod_fr $FR_CODIF − m $DIM − stop $STOP
        −ksp_monitor − $flag $work − ksp_max_it $MAXIT
        −socketS $SERVER − socketP $PORT
```

Capitolo 6

Risultati, test ed osservazioni

Il lavoro di ricerca ha riguardato la costruzione di una metodologia di checkpointing disk-based che è stata definita "ibrida asincrona distribuita". Tale medotologia è stata implementata all'interno del metodo del Gradiente Coniugato della libreria PETSc, dando vita ad una versione fault tolerant del modulo del CG (CGFT). La realizzazione del sistema di checkpointing/migration ha, poi, abilitato tutte le applicazioni basate sul CGFT a sopravvivere ai guasti (di uno, più nodi del cluster o dell'intero cluster di esecuzione) mediante la migrazione (locale o remota) dell'esecuzione su risorse alternative del contesto distribuito.

Nelle prossime sezioni, dopo una breve descrizione dell'ambiente distribuito nel quale sono state effettuate le prove, sono riportati i risultati dei test funzionali e prestazionali del sistema di checkpointing/migration, effettuati su un'applicazione che fa uso del CGFT.

6.1 Descrizione dell'ambiente

I test sono stati svolti sull'infrastruttura, costituita da risorse di calcolo e di storage , basata sul paradigma del Grid Computing e realizzata dall'Unversità Federico II nell'ambito del progetto PON S.Co.P.E. dell'Avviso 1575.

La suddetta infrastruttura è stata utilizzata come descritto di seguito:

- per il lancio e il monitoraggio dell'esecuzione dell'applicazione, è stata utilizzata la User Interface, che fornisce gli strumenti per l'accesso trasparente all'architettura GRID,
- per l'esecuzione dell'applicazione sono stati utilizzati due cluster:
  - uno, sito al Dipartimento di Matematica ed Applicazioni R. Caccioppoli, denominato *scope-dma*
  - l'altro, sito nella nuova struttura che ospita il centro di calcolo di Ateneo
- per la memorizzazione dei dati, utili alla migrazione dell'applicazione a seguito dei fault, è stata utilizzata la risorsa di storage sita nella nuova struttura che ospita il centro di calcolo di Ateneo
- per il coordinamento dei ruoli coinvolti sono stati utilizzati i seguenti servizi di tipo "collective" introdotti nel *Capitolo 3*:
  - il servizio di Autenticazione/Autorizzazione per l'accesso sicuro all'infrastruttura,
  - il Metascheduler per la sottomissione dell'applicazione sulle risorse di calcolo,
  - il Logical File Catalogue per la registrazione dei dati, mediante l'utilizzo di nomi simbolici e

– il Sistema Informativo per il reclutamento di risorse.

Il sistema di calcolo *scope-dma* è costituito da un Blade DELL m1000 di 16 nodi (DELL m600) con le seguenti caratteristiche:

hardware:

- due processori quad core Intel Xeon E5410@2.33GHz (Architettura a 64 bit),
- 16 Gb di RAM,
- due dischi SATA da 80GB in configurazione RAID1,
- due schede Gigabit Ethernet
- una scheda Infiniband (Mellanox Technologies MT25418 ConnectX IB DDR) per la connettività ad alta banda e bassa latenza.

software:

- Sistema operativo: Scientific Linux 4.6.
- File system disponibili: NFS[1] e area di scratch locale ai nodi[2]

Del centro di calcolo di Ateneo sono stati, invece, utilizzati:

- 3 Blade DELL m1000 identici, per configurazione hardware e software, al sistema del Diparimento di Matematica ed Applicazioni
- una SAN[3] EMC^2, che esporta ai nodi di calcolo, mediante File

[1] L'NFS (Network file system) è un file system di rete. Esso è un sistema per l'accesso a file remoti attraverso LAN, o anche WAN

[2] Si indica con area locale di scratch uno spazio di memoria locale riservato.

[3] Storage Area Network

System distribuito Lustre[4] una porzione di storage condivisa da tutti i nodi e aggiuntiva rispetto al disco locale.

6.2 Caratterizzazione dei test

Al fine di validare in maniera esaustiva il lavoro svolto, sono stati effettuati numerosi test, volti sia a verificare dal punto di vista funzionale i meccanismi implementati, sia a valutarne le prestazioni.

I primi test, utilizzati per la validazione del sistema di checkpointing e migrazione locale al cluster, descritto nel *Capitolo 4*, sono stati effettuati sul cluster *scope-dma*.

I test prestazionali sono stati, invece, eseguiti su parte delle risorse di calcolo e storage del centro di calcolo universitario, dove si disponeva di un numero maggiore di nodi e diversi tipi di file system (uno locale e due di rete) su cui verificare le prestazioni dei meccanismi di checkpointing.

Su tale sistema, aumentando il numero dei nodi, è stato possibile accrescere la dimensione del sistema lineare da risolvere con il CGFT e, dunque, valutare meglio l'impatto dei meccanismi di checkpointing sul tempo di esecuzione totale dell'applicazione quando i dati da salvare diventano dell'ordine di Giga Bytes.

Un'ulteriore batteria di test ha riguardato, inoltre, la validazione dei meccanismi di migrazione dell'applicazione da un cluster all'altro: l'applicazione è stata dapprima eseguita sul sistema *scope-dma* e, al verificarsi di un fault, è avvenuta la migrazione sui nodi del cluster del centro di calcolo di Ateneo.

[4] Lustre è un filesystem di rete generalmente usato da sistemi cluster progettato da Sun Microsystem. Il nome stesso deriva da una fusione tra le parole Linux e cluster. Il progetto mira a creare un filesystem che possa supportare reti di cluster di decine di migliaia di nodi e petabytes di dati da immagazzinare, senza però compromettere la velocità e la sicurezza; è rilasciato sotto licenza GPL.

Le batterie di test, relative al sistema di calcolo con più nodi, descritte nelle prossime sezioni 6.3 e 6.4 hanno previsto la risoluzione di tre sistemi lineari con matrici sparse di dimensioni 225000000 x 225000000, 400000000 x 400000000 e 625000000 x 625000000 [5], utilizzando un numero di processori compreso tra 8 e 28.

La risoluzione di ciascun sistema lineare ha richiesto la gestione di una quantità notevole di elementi non nulli (di tipo *double*[6]), rispettivamente pari a: $2.25 * 10^9$, $4 * 10^9$ e $6.25 * 10^9$ [7].

6.3 Test di prestazioni della fase di checkpointing

In questa sezione sono presentati i test svolti con l'obiettivo di verificare il comportamento dei due meccanismi di checkpointing: coordinato e codificato.

Le dimensioni della matrice sparsa del sistema lineare, da risolvere con il CGFT, sono: $N_1 = 5{*}10^{16}$, $N_2 = 1.6{*}10^{17}$ ed $N_3 = 3.9{*}10^{17}$. Per tali dimensioni, i dati di checkpointing (cioè gli elementi, in doppia precisione, dei 4 vettori da salvare) sono, rispettivamente: $9{*}10^8$, $1.6 * 10^9$ e $2.5 * 10^9$ elementi.

Di seguito sono riportati i risultati dei test che hanno fornito i tempi delle fasi di salvataggio dei dati. Tali tempi sono relativi sia al checkpointing coordinato sia a quello codificato, e si fa uso, per la memorizzazione dei dati di checkpointing, sia dell'area scratch locale ad ogni nodo, sia ai due file system di rete Lustre ed NFS.

I dati riportati nelle tabelle 6.1 , 6.2 e 6.3 si riferiscono al tempo

[5] Test 1 - dimensione matrice: $5 * 10^{16}$, Test 2 - dimensione matrice: $1,6 * 10^{17}$ e Test 3 - dimensione matrice $3.9 * 10^{17}$.

[6] i dati in doppia precisione sono rappresentati, sulle risorse in esame, con 8 bytes

[7] Test 1 - dati da gestire: 16.8 GB; Test 2 - dati da gestire: 29.8 GB e Test 3 - dati da gestire: 46,5 GB.

NP	Area locale di scratch	Lustre FS	NFS
8	13.3 sec.	79.04 sec.	167.74 sec.
12	9.39 sec.	73.08 sec.	153.81 sec.
16	7.21 sec.	70.8 sec.	149.38 sec.
20	6.17 sec.	69.93 sec.	142.91 sec.
24	5.48 sec.	67.95 sec.	142.04 sec.
28	5 sec.	56 sec.	141 sec.

Tabella 6.1: Tempi di salvataggio dei dati di un'iterazione con il checkpointing codificato con i file system locale, Lustre ed NFS - dimensione globale dei dati di checkpointing: 6.7GB; dimensione locale dei dati di checkpointing $6.7GB/NP$.

NP	Area locale di scratch	Lustre FS	NFS
8	34 sec.	124.63 sec.	259 sec.
12	16.46 sec.	116 sec.	251 sec.
16	12 sec.	105 sec.	242 sec.
20	11.24 sec.	98 sec.	240.83 sec.
24	9.07 sec.	87 sec.	243 sec.
28	7.97 sec.	85 sec.	244 sec.

Tabella 6.2: Tempi di salvataggio dei dati di un'iterazione con il checkpointing codificato con i file system locale, Lustre ed NFS - dimensione globale dei dati di checkpointing: 11.9GB; dimensione locale dei dati di checkpointing $11.9GB/NP$.

NP	Area locale di scratch	Lustre FS	NFS
8	235 sec.	362.68 sec.	639 sec.
12	39.75 sec.	206.81 sec.	306.15 sec.
16	18.95 sec.	196.19 sec.	373.41 sec.
20	17.07 sec.	192.85 sec.	374.1 sec.
24	13.97 sec.	189.72 sec.	369.27 sec.
28	13.49 sec.	188.12 sec.	363.61 sec.

Tabella 6.3: Tempi di salvataggio dei dati di un'iterazione con il checkpointing codificato con i file system locale, Lustre ed NFS - dimensione globale dei dati di checkpointing: 18GB; dimensione locale dei dati di checkpointing $18GB/NP$

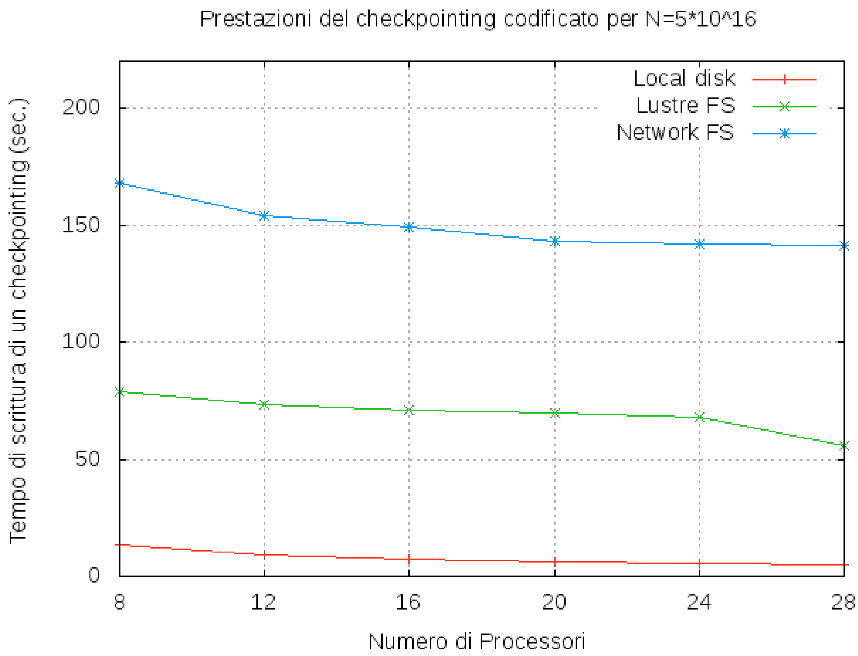

Figura 6.1: Tempi per il checkpointing codificato su area disco locale, lustre FS e NFS: matrice sparsa con $5 * 10^{16}$ elementi. Dati di checkpoint: $9 * 10^8$ elementi $= 6.7$GB

medio di salvataggio dei dati di checkpointing (facendo uso della strategia codificata).

Come si evince dai grafici delle figure 6.1, 6.2 e 6.3, tale tempo dipende dal numero di processori utilizzati: infatti se N è la dimensione globale dei dati di checkpointing, e p il numero di processori, ogni processore, durante la sua fase di checkpointing dovrà salvare N/p dati.

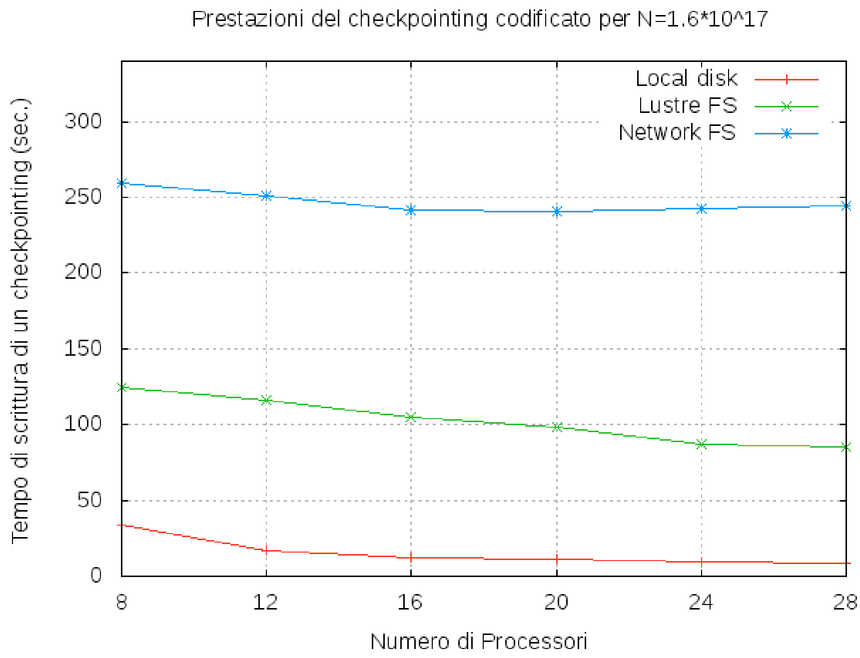

Figura 6.2: Tempi per il checkpointing codificato su area disco locale, lustre FS e NFS: matrice sparsa con $1.6 * 10^{17}$ elementi. Dati di checkpoint: $1.6 * 10^9$ elementi = 11.9GB

I grafici delle figure 6.1, 6.2 e 6.3, evidenziano, inoltre, la maggiore efficienza del salvataggio su area scratch locale al nodo, suggerendo il ricorso a file system di rete solo in casi di reale necessità (ad es. per eseguire i run di applicazioni con dati di dimensioni non gestibili con il solo disco locale oppure per favorire le fasi di migrazione locale al cluster).

Un'altra osservazione che va fatta è relativa alla regolarità del comportamento del file system Lustre rispetto ad NFS: aumentando la dimensione dei dati di checkpointing e all'aumentare del numero di processori, il tempo di salvataggio dei dati su FS Lustre diminui-

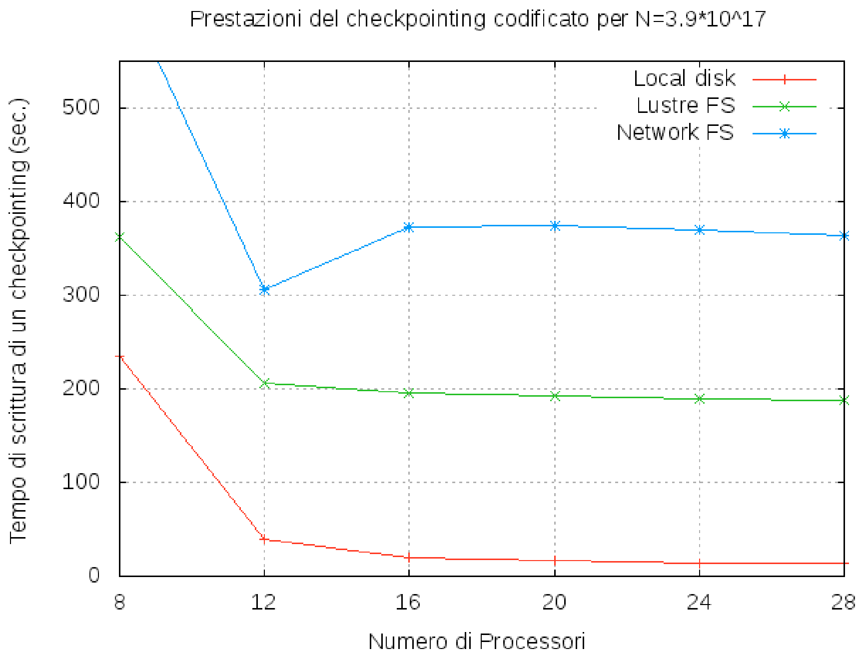

Figura 6.3: Tempi per il checkpointing codificato su area disco locale, lustre FS e NFS: matrice sparsa con $3.9 * 10^{17}$ elementi; Dati di checkpoint: $2.5 * 10^9$ elementi = 18GB

Dim.ne dati di checkpointing	Local FS	Lustre FS	NFS
$9 * 10^8$	96.88 sec.	266.94 sec.	162.41 sec.
$1.6 * 10^9$	157.52 sec.	234.98 sec.	256 sec.
$2.5 * 10^9$	266.94 sec.	417 sec.	449 sec.

Tabella 6.4: Tempi di salvataggio dei dati di un'iterazione con il checkpointing coordinato con i file system locale, Lustre ed NFS

sce (come accade per l'area scratch) mentre ciò accade per NFS solo per il problema di dimensione minore.

Il fenomeno appena evidenziato è da attribuire al fatto che Lustre è un file system distribuito e, pertanto, gestisce in maniera più

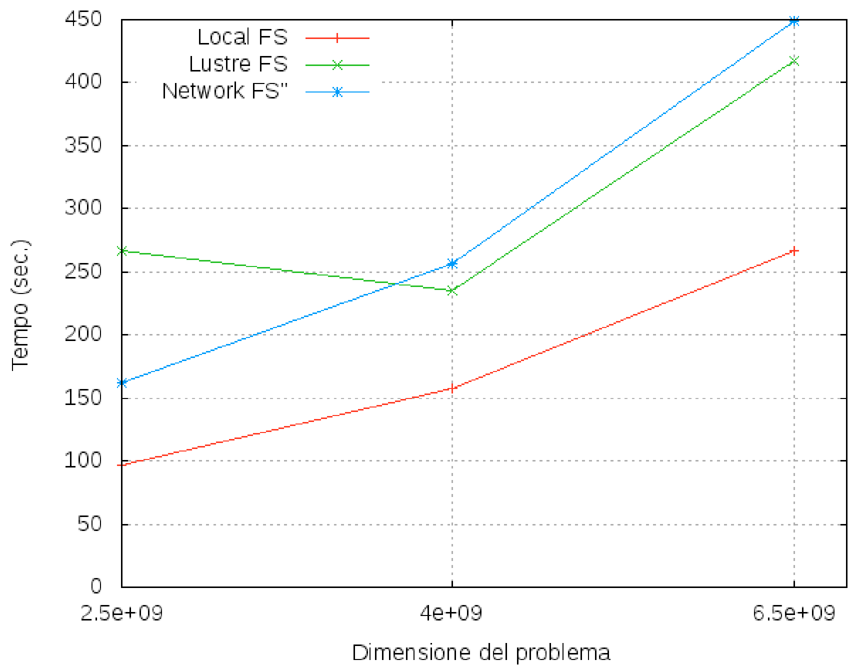

Figura 6.4: Matrici di $5 * 10^{16}$ $1.6 * 10^{17}$ e $3.9 * 10^{17}$ elementi: tempi per il checkpointing coordinato su area disco locale, lustre FS e NFS.

efficiente, rispetto ad NFS, l'accesso concorrente ad una stessa area disco, da parte di più processori.

La figura 6.4 mostra, invece, l'andamento dei tempi medi di salvataggio dei dati, riportati in tabella 6.4, ottenuti utilizzando la strategia coordinata. In questo caso, il grafico, contrariamente a quello relativo alla strategia codificata, non riporta la dipendenza del tempo di checkpointing dal numero di processori.

In realtà i tempi di scrittura del checkpointing coordinato non sono indipendenti dal numero di processori utilizzato: prima della fase di scrittura, infatti, è necessaria un'operazione collettiva nella quale tutti i processori mandano i propri dati locali di checkpointing ad un processore che, poi li scrive sul disco.

Nei test effettuati, con le dimensioni N_1, N_2 ed N_3 del sistema lineare e un numero di processori compreso tra 8 e 28, i tempi di comunicazione sono risultati trascurabili rispetto a quelli di scrittura su disco.

I tempi di scrittura del checkpointing coordinato, per un'iterazione, sono molto più rilevanti rispetto a quelli della strategia codificata (poichè un solo processo scrive tutti i dati di checkpointing).

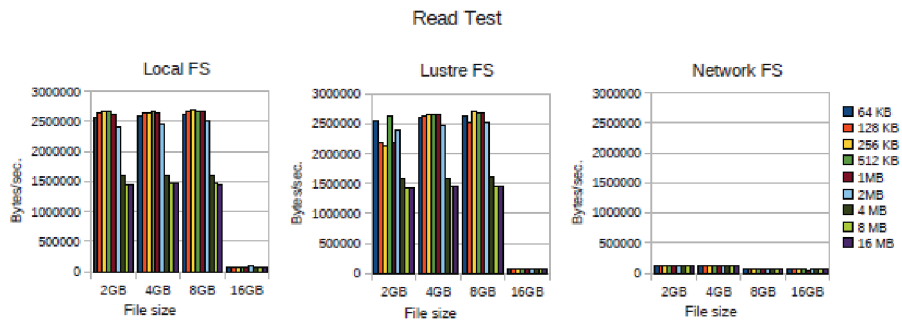

Figura 6.5: Test di lettura: LOCAL FS, LUSTRE FS e Network FS

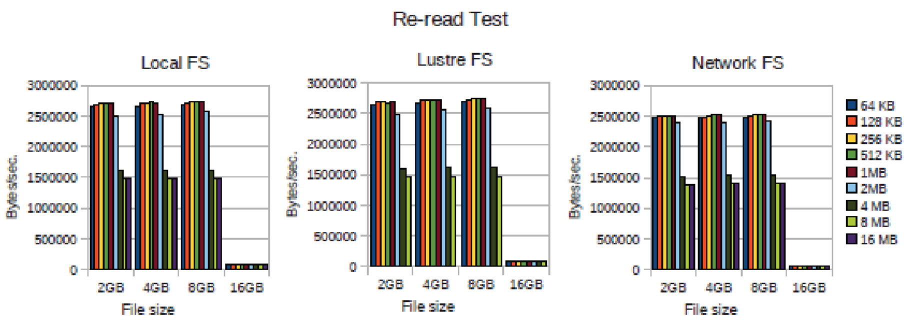

Figura 6.6: Test di rilettura: LOCAL FS, LUSTRE FS e Network FS

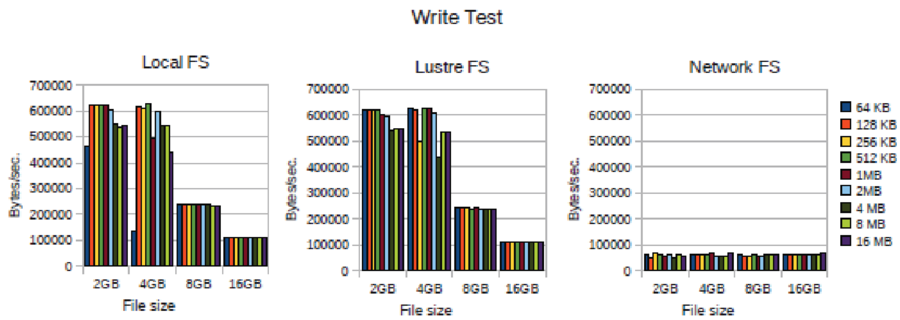

Figura 6.7: Test di scrittura: LOCAL FS, LUSTRE FS e Network FS

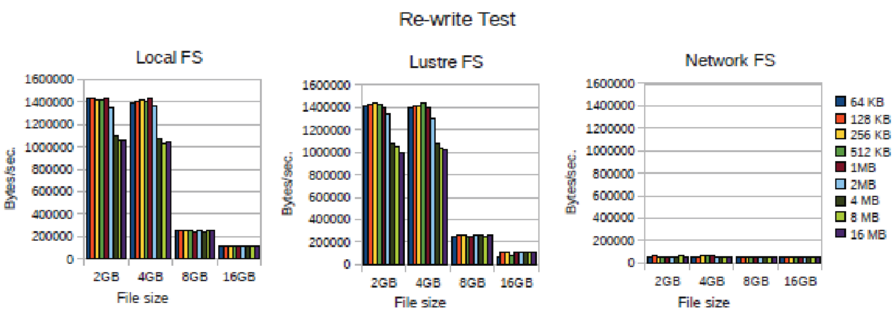

Figura 6.8: Test di riscrittura: LOCAL FS, LUSTRE FS e Network FS

Per avere un ulteriore riscontro dei fenomeni osservati durante i test di questa sezione, nelle figure 6.5, 6.7, 6.6 e 6.8 sono riportate, per dati di dimensioni confrontabili con quelle dei dati di checkpointing, le prestazioni [8], in termini di bytes/sec. per le operazioni di lettura, scrittura, rilettura e riscrittura relative ai tre file system considerati nei test: area scratch locale ai nodi, Lustre FS ed NFS.

[8] I dati riportati si riferiscono all'esecuzione del benchmarch "iozone" e mirano a fornire esclusivamente una idea qualitativa del comportamento atteso dal meccanismo di checkpointing/rolling back al variare del supporto di memorizzazione utilizzato.

Nella prossima sezione sono dapprima riportate informazioni circa il numero di iterazioni e il tempo totale di esecuzione impiegati dall'applicazione per la risoluzione dei tre sistemi lineari della batteria di test.

Alla luce di tali informazioni e, conoscendo l'overhead introdotto dalle fasi di checkpointing/rolling back, vengono fatte alcune considerazioni rispetto all'opportunità di utilizzo e all'utilità dei meccanismi di checkpointing, nel caso dei due sistemi lineari più grandi. Vengono valutati i benefici, quando presenti, dovuti ai meccanismi di checkpointing in scenari di esecuzione con uno o più fault e failure-free e riportate considerazioni sull'opportunità di migrazione dell'applicazione in caso di fault distinguendo i casi in cui è possibile una "migrazione locale" dell'applicazione, da quelli in cui è necessaria una "migrazione remota".

6.4 Test di migrazione

In tabella 6.5 sono riportati i tempi di esecuzione di un'iterazione del CG per i sistemi di dimensione N_1, N_2 ed N_3.

NP	$N_1 = 5 * 10^{16}$	$N_2 = 1.6 * 10^{17}$	$N_3 = 3.9 * 10^{17}$
8	3.18 sec.	5.51 sec.	8.76 sec.
12	2.48 sec.	3.62 sec.	5.65 sec.
16	2.05 sec.	2.86 sec.	5.46 sec.
20	1.18 sec.	2.1 sec.	4.67 sec.
24	1.06 sec.	1.74 sec.	4.16 sec.
28	0.98 sec.	1.49 sec.	2.31 sec.

Tabella 6.5: Tempi medi di esecuzione di un'iterazione del CG al variare della dimensione N del problema e del numero di processori

I test di questa sezione sono stati svolti utilizzando 16 processori e l'area scratch locale ai nodi. I sistemi lineari utilizzati sono quelli

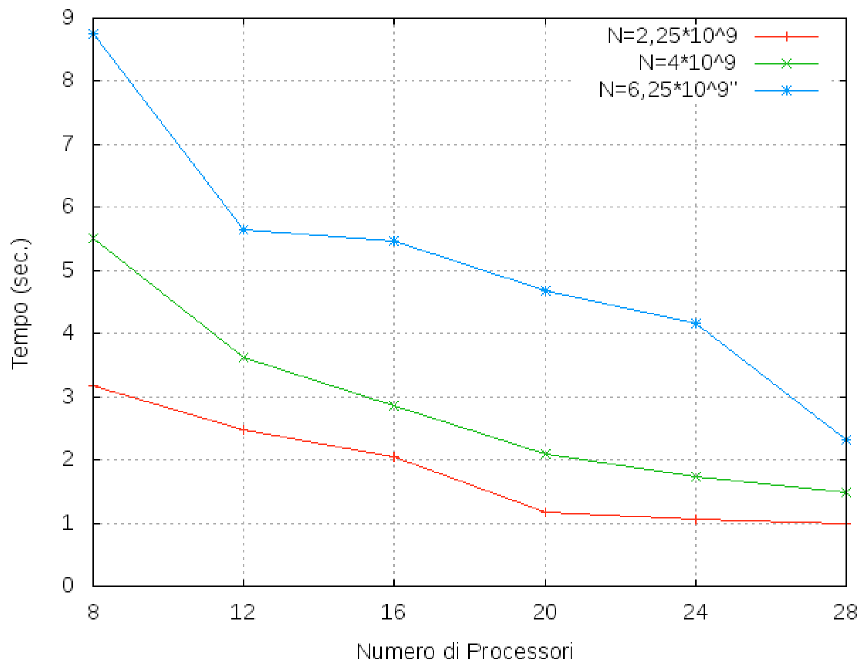

Figura 6.9: Tempi medi di esecuzione di un'iterazione del CG al variare della dimensione N del problema e del numero di processori

di dimensioni N_2 ed N_3 considerati nella precedente sezione: essi convergono dopo un elevato numero di iterazioni.

In particolare:

- **il sistema di ordine** N_2 converge dopo 5761 iterazioni,
- **il sistema di ordine** N_3 converge dopo 6892 iterazioni.

La scelta di utilizzare 16 processori è motivata dal fatto che, secondo la tabella 6.5, vi è ancora un utilizzo efficiente delle risorse di calcolo (i tempi di esecuzione descrescono in maniera meno significativa se si utilizzano da 20 a 28 processori).

Dato l'elevato numero di iterazioni e la considerevole dimensione dei dati da trattare è possibile, in questi casi, fare confronti significativi tra l'esecuzione dell'applicazione, che risolve i sistemi lineari mediante il CG, con e senza meccanismi di checkpointing.

A seconda di quando si verifica il fault, infatti, diventa più o meno rilevante, rispetto ai tempi totali di esecuzione dell'applicazione, l'overhead comunque introdotto dalle fasi di checkpointing/rolling back.

Per chiarire meglio l'affermazione appena fatta, consideriamo il caso del sistema di dimensioni N_2.

Tale sistema lineare viene risolto dopo 5761 iterazioni, ciascuna di durata 2.86 secondi (con 16 processori), secondo la tabella 6.5. Disabilitando i meccanismi di checkpointing e in assenza di fault, l'applicazione dura circa 16476 secondi (quasi 5 ore).

Tuttavia, in assenza di meccanismi di checkpointing/rolling back, se si verifica anche un solo fault durante l'esecuzione, è necessario far ripartire l'esecuzione da principio, ripetendo anche le iterazioni già eseguite.

It_{fault}	T_{it} perse	T_{tot}
1000	2860 sec.	16476+2860= 19336 sec.
2000	5720 sec.	16476+5720= 22196 sec.
3000	8580 sec.	16476+8580= 25056 sec.
4000	11440 sec.	16476+11440= 27916 sec.
5000	14300 sec.	16476+14300= 30776 sec.
no fault	——	16476 sec.

Tabella 6.6: Esecuzione dell'applicazione priva di meccanismi di checkpointing: conseguenze di un fault sul tempo totale di esecuzione se il fault avviene alle iterazioni 1000, o 2000, o 3000, o 4000, o 5000 sul tempo totale di esecuzione; la colonna relativa al parametro "T_{it} *perse*" indica i tempi di esecuzione delle iterazioni da ripetere a seguito del fault; l'ultima riga fa riferimento ai tempi totali di esecuzione in assenza di fault.

Se il fault si verifica alle prime iterazioni, le conseguenze di una ripartenza da principio dell'applicazione sono trascurabili rispetto al tempo totale di esecuzione, mentre si ha un notevole spreco di tempo (che arriva in alcuni casi a duplicare il tempo totale di esecuzione) se il fault si verifica verso la fine dell'esecuzione. (cfr. tabella 6.6)

N	T. esec.	T. checkp.	T. tot.	Overhead % checkp.
$N_2 = 1,6 * 10^{17}$	16476 sec.	8151 sec.	24614 sec.	49.4%
$N_3 = 3.9 * 10^{17}$	37630 sec.	18666 sec.	56296 sec.	49.6%

Tabella 6.7: Esecuzione dell'applicazione con i meccanismi di checkpointing abilitati: overhead introdotto in assenza di fault rispetto al tempo di esecuzione con i meccanismi di checkpointing non abilitati

Se invece si utilizzano i meccanismi di checkpointing/migration, per il sistema di dimensioni N_2 utilizzando sempre 16 processori e il file system locale, si possono presentare diversi scenari, alcuni dei quali sono stati esaminati di seguito.

Durante l'esecuzione, il sistema che regola automaticamente la frequenza dei checkpointing dell'uno e dell'altro tipo, ha programmato un checkpointing coordinato ogni 220 iterazioni ed un checkpointing codificato ogni 17 [9].

Sono stati valutati i seguenti scenari di esecuzione:

- **caso 1:** nessun fault durante l'esecuzione
- **caso 2:** un solo fault durante l'esecuzione
- **caso 3:** più di un fault durante l'esecuzione
- **caso 4:** fault dell'intero cluster di esecuzione

[9] Vengono eseguiti 26 checkpointing coordinati ciascuno di durata 158 secondi circa, e 338 checkpointing codificati ciascuno di durata 12 secondi circa

Nel **caso 1**, secondo i dati riportati nella riga 2 della tabella 6.7, in assenza di fault, l'introduzione dei meccanismi di checkpointing introduce un peggioramento nei tempi di esecuzione di circa il 49% rispetto all'applicazione priva dei meccanismi di checkpointing.

It_{fault}	T_{it} perse	T_{tot}	$Overhead_{chkp}$
1000	(14 it.) 40.04 sec.	24614+40.04 = 24654.04 sec.	27%
2000	(11 it.) 31.46 sec.	24614+31.46 = 24645.46 sec.	11%
3000	(8 it.) 22.88sec.	24614+22.88 = 24636.88 sec.	-2%
4000	(5 it.) 14.30 sec.	24614+14.30 = 24628.30 sec.	-12%
5000	(2 it.) 5.72 sec.	24614+5.72 = 24619.72 sec.	-21%

Tabella 6.8: Esecuzione dell'applicazione con i meccanismi di checkpointing abilitati: overhead introdotto in assenza di fault rispetto al tempo di esecuzione con i meccanismi di checkpointing non abilitati. It_{fault} è l'iterazione alla quale si verifica il fault; T_{it} perse si riferisce al tempo di esecuzione delle iterazioni da ripetere in fase di rolling back da checkpointing.

Nel **caso 2**, è stata considerata l'occorrenza di un solo fault (tabella 6.8);

Come si evince dai dati in tabella, all'aumentare del numero di iterazioni eseguite prima del fault, l'utilizzo delle tecniche di checkpointing diventa sempre meno oneroso fino a diventare preferibile alla esecuzione dell'applicazione da principio.

Infatti, se il fault si verifica all'iterazione 3000, accade quanto segue: gli ultimi checkpointing di tipo coordinato e codificato, prima del fault, sono stati eseguiti rispettivamente all'iterazione 2860 e all'iterazione 2992.

L'applicazione, in fase di rolling back, riprende l'esecuzione dall'iterazione 2992, relativa al checkpoint più recente (codificato) ripetendo solo 8 iterazioni, e il tempo totale è di 24636.88 secondi (riga 3, tabella 6.8).

In questo caso il guadagno, in termini percentuali, sui tempi di esecuzione in assenza di meccanismi di checkpointing, è del 2%; tale guadagno incrementa fino al 21% se il fault avviene all'iterazione 5000 (riga 5, tabella 6.8).

Nel **caso 3**, considerando un fault all'iterazione 2000 ed uno all'iterazione 5000, l'applicazione effettua rolling back 2 volte, ripetendo in totale solo 13 iterazioni. Il tempo speso è di 24614 + 37.18 secondi (per le 13 iterazioni). L'applicazione, priva dei meccanismi di checkpointing impiegherebbe 36496 secondi. In questo caso, grazie ai meccanismi di checkpointing si guadagna il 32.4% rispetto al tempo totale di esecuzione.

It_{fault}	T_{it} perse	T_{tot}
1000	5460 sec.	37630+5460 = 43090 sec.
2000	10920 sec.	37630+10920 = 48550 sec.
3000	16380 sec.	37630+16380 = 54010 sec.
4000	21840 sec.	37630+21840 = 59470 sec.
5000	27300 sec.	37630+27300 = 64930 sec.
6000	32760 sec.	37630+32760 = 70390 sec.
no fault	——	37630 sec.

Tabella 6.9: Esecuzione dell'applicazione priva di meccanismi di checkpointing: conseguenze di un fault sul tempo totale di esecuzione se il fault avviene alle iterazioni 1000, o 2000, o 3000, o 4000, o 5000, o 6000 sul tempo totale di esecuzione; la colonna relativa al parametro "T_{it} *perse*" indica i tempi di esecuzione delle iterazioni da ripetere a seguito del fault; l'ultima riga fa riferimento ai tempi totali di esecuzione in assenza di fault.

Nelle tabelle 6.9 e 6.10 sono riportati risultati analoghi a quelli delle tabelle 6.6 e 6.8, ottenuti con 16 processori sul sistema di dimensioni N_3.

It_{fault}	T_{it} perse	T_{tot}	$Overhead_{chkp}$
1000	(6 it.) 32.76 sec.	56296+32.76 = 56328.76 sec.	31%
2000	(12 it.) 65.52 sec.	56296+65.52 = 56361.52 sec.	16%
3000	(4 it.) 21.84 sec.	56296+21.84 = 56317.84 sec.	0%
4000	(10 it.) 54.60 sec.	56296+54.60 = 56350.60 sec.	-1%
5000	(2 it.) 10.92 sec.	56296+10.92 = 56306.92 sec.	-13%
6000	(8 it.) 43.68 sec.	56296+43.68 = 56339.68 sec.	-20%

Tabella 6.10: Esecuzione dell'applicazione con i meccanismi di checkpointing abilitati: overhead introdotto in assenza di fault rispetto al tempo di esecuzione con i meccanismi di checkpointing non abilitati. It_{fault} è l'iterazione alla quale si verifica il fault; T_{it} perse si riferisce al tempo di esecuzione delle iterazioni da ripetere in fase di rolling back da checkpointing.

Nei casi su riportati, è stato possibile effettuare una migrazione locale al cluster di esecuzione, semplicemente richiedendo la ripresa dell'esecuzione su un set locale di processori non guasti: i tempi di rolling back sono trascurabili rispetto a quelli di salvataggio dati[10].

N	Dati di checkpointing	T. di trasferimento
$N_1 = 5 * 10^{16}$	6.7GB	75 sec.
$N_2 = 1,6 * 10^{17}$	11.9GB	120 sec.
$N_3 = 3.9 * 10^{17}$	18GB	134 sec.

Tabella 6.11: Tempi medi di trasferimento dati dalla risorsa di storage al nodo di calcolo: vengono utilizzati 4 thread concorrenti che si occupano di trasferire i 4 vettori contenenti i dati di checkpointing.

Nel **caso 4**, invece, la fase di rolling back prevede una gestione remota dei dati di esecuzione. Relativamente all'infrastruttuta distribuita utilizzata, un'idea dei tempi in gioco, necessari allo spostamento dei dati da una risorsa remota di storage al cluster su cui

[10] sono confrontabili con i tempi di esecuzione di un'iterazione

viene ripresa l'esecuzione, è fornita dalla tabella 6.11.

E' ovvio che, in questo caso, ai tempi di rolling back, vanno aggiunti, poi, quelli di re-inizializzazione dell'applicazione (reclutamento delle nuove risorse di calcolo, tempo in coda sul nuovo cluster di esecuzione) che dipendono dallo stato dell'infrastruttura distribuita al momento della migrazione.

E' bene precisare che le considerazioni espresse, circa l'overhead introdotto dai meccanismi di checkpointing, si riferiscono ad un'applicazione che, su molti dati, effettua una esigua quantità di calcoli.

Dalle esperienze effettuate si può concludere che i vantaggi dell'utilizzo delle tecniche di checkpointing non sono sempre evidenti. Tuttavia, se si pensa ad applicazioni che gestiscono la stessa mole di dati ma utilizzano algoritmi con complessità maggiore di quella considerata, i tempi di salvataggio dei dati di checkpointing rimangono gli stessi e l'utilità dei meccanismi di checkpointing risulta molto più evidente.

La possibilità di utilizzo dei meccanismi di checkpointing, come quello realizzato in questo lavoro di ricerca, non è limitata solo alla predisposizione, nelle applicazioni, della fault tolerance; ci sono altri scenari nei quali salvare lo stato corrente dell'esecuzione e riprenderla successivamente è molto più che utile: sempre più ci si imbatte nelle policy restrittive imposte dai grandi centri di calcolo, secondo le quali è possibile utilizzare le risorse di calcolo solo per un fissato intervallo di tempo. In tali casi il discorso del checkpointing, che abilita l'applicazione a riprendere l'esecuzione in un secondo momento, senza perdere quanto già svolto, è molto utile.

Bibliografia

[1] EGEE Project Home Page. http://www.eu-egee.org/.

[2] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall, and Mitchel W. Sukalski. Architecture of LA-MPI, A Network-Fault-Tolerant MPI. *Parallel and Distributed Processing Symposium, International*, 1:15b, 2004.

[3] Vari autori. The Message Passing Interface (MPI) standard - Version 2.2. URL http://www.mcs.anl.gov/research/projects/mpi/, 2009.

[4] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp D., Kaushik M., Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. PETSc 3.0.0 Users Manual. URL http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf, 2008.

[5] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*,

pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[6] Stephen Burke, Simone Campana, Patricia Mendez Lorenzo, Christopher Nater, Roberto Santinelli, and Andrea Sciaba. GLITE 3.1 USER GUIDE. URL https://edms.cern.ch/file/722398/gLite-3-UserGuide.pdf, 2008.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26:145–185, 1994.

[8] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building Fault Survivable MPI Programs with FT_MPI Using Diskless Checkpointing. In *In Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 2005.

[9] David Dewolfs, Jan Broeckhove, Vaidy S. Sunderam, and Graham E. Fagg. FT_MPI, Fault-Tolerant Metacomputing and Generic Name Services: A Case Study. In *PVM/MPI*, pages 133–140, 2006.

[10] Christian Engelmann, Christian Engelmann, and Al Geist. Super-Scalable Algorithms for Computing on. In *Proceedings of ICCS*, pages 313–321. Springer, 2005.

[11] Graham E. Fagg and Jack J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, 2004.

[12] FT_MPI Home Page. URL http://icl.cs.utk.edu/ftmpi/overview/index.html.

[13] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, USA, first edition, 1998.

[14] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2. a. edition, December 2003.

[15] Roland Freund, Gene H. Golub, and Noel M. Nachtigal. Iterative Solution of Linear Systems. *Acta Numerica*, 1:57–100, 1992.

[16] Al Geist and Christian Engelmann. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. URL http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf, 2002.

[17] Patricia D. Hough, Tamara G. Kolda, and Virginia J. Torczon. Asynchronous Parallel Pattern Search For Nonlinear Optimization. *SIAM J. Sci. Comput*, 23:134–156, 2000.

[18] Edward Hung and M. Phil Student. Fault Tolerance and Checkpointing Schemes for Clusters of Workstations. 2008.

[19] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A Checkpoint and Restart Service Specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

[20] Najib A. Kofahi, Said Al-Bokhitan, and Ahmed Al-Nazer Journal. On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis. *Information Technology Journal*, 4:367–376, 2005.

[21] E Laure and alt. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12:33–45, 2006.

[22] Kihwal Lee and Lui Sha. Process resurrection: A fast recovery mechanism for real-time embedded systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:292–301, 2005.

[23] H.D. Simon M.A. Heroux, P. Raghavan. *Fault Tolerance in Large Scale Scientific Computing*, chapter 11, pages 203–220. Siam Press, 2006.

[24] Almerico Murli. *Lezioni di Calcolo Parallelo*. Liguori Editore, 1 edition, 2006.

[25] James S. Plank, Kai Li, and Michael A. Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[26] Yousef Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[27] Luis M. Silva and Joao Gabriel Silva. An Experimental Study about Diskless Checkpointing. *EUROMICRO Conference*, 1:10395, 1998.

[28] Luìs Moura Silva and Gabriel Joao Silva. The Performance of Coordinated and Independent Checkpointing. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 280–284, Washington, DC, USA, 1999. IEEE Computer Society.

[29] G. L. G. Sleijpen and H.A van der Vorst. Krylov Subspace Methods for Large Linear Systems of Equations. URL `http://citeseerx.ist.psu.edu/viewdoc/download`, 1993.

[30] Hertong Song, Chokchai Leangsuksun, and Raja Nassar. Availability Modeling and Analysis on High Performance Cluster Computing Systems. *First International Conference on Availability, Reliability and Security*, 0:305–313, 2006.

[31] Jason Sonnek, Mukesh Nathan, Abhishek Chandra, and Jon Weissman. Reputation-Based Scheduling on Unreliable Distributed Infrastructures. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 30, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Sathish Vadhiya, Antoine Henry, and K. Raghavendra. SRS Library documentation. URL http://garl.serc.iisc.ernet.in/SRS/srs_doc.pdf, 2008.

[33] Sathish S. Vadhiyar and Jack J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. In *In: Parallel Processing Letters. Volume*, pages 291–312, 2002.

[34] Nitin Vaidya. On Checkpoint Latency. In *In Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 60–65, 1995.

[35] Nitin H. Vaidya. A Case for Two-level Distributed Recovery Schemes. In *In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.

[36] C. Wang, F. Mueller, C. Engelmann, and S.L. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[37] F. Zambonelli. On the Effectiveness of Distributed Checkpoint Algorithms for Domino-free Recovery. *International Symposium on High-Performance Distributed Computing*, 0:124, 1998.

[38] Yong Zheng, Alessandro Bassi, Micah Beck, James S. Plank, and Rich Wolski. Internet Backplane Protocol: C API 1.4. URL http://loci.cs.utk.edu/ibp/documents/IBPClientAPI.pdf, 2004.