Corso di Programmazione Conconcorrente

Decomposizione Parallela

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- Decomposizione Parallela di Problemi
  - Una classificazione dei problemi
    - *universale vs esistenziale*
    - *statica vs dinamica*
- Problemi Statici ed Universali
  - Es. somma degli elementi di un array
- Problemi Statici ed Esistenziali
  - ***`ExecutorCompletionService`***
  - Es. generico risolutore concorrente
- Problemi Dinamici ed Universali
  - Tecniche di *work-stealing*
  - Framework Fork/Join la Decomposizione Parallela di Problemi
    - **`ForkJoinPool`**, **`ForkJoinTask`**, **`RecursiveTask`**, **`RecursiveAction`**
    - Fine-Tuning dei parametri
- Problemi Dinamici ed Esistenziali

Decomposizione Parallela (1)

- Molti problemi che richiedono notevoli risorse di calcolo si prestano a naturali decomposizioni *top-down*
  - I più indicati per sfruttare lo speed-up offerto da architetture multi-processore (cfr. *legge di Amdahl*)

- Il *Lightweight Executable Framework* semplifica la scrittura del codice concorrente necessario

- "Schema" di una tipica soluzione parallela:
  1. *Decomposizione in sotto-problemi*
  2. *Elaborazione parallela dei sotto-problemi*
  3. *Ricomposizione (seriale) dei risultati parziali in uno globale*

Decomposizione Parallela (2)

- Alcune osservazioni:
  - Supponiamo, per semplicità:
    - la piattaforma dedicata
    - il problema privo di significativo I/O (>>)

    altrimenti i ragionamenti legati allo speed-up si complicano
  - i sotto-problemi risolti nel passo 2 devono essere sufficientemente “corposi” da rendere i benefici legati alla parallelizzazione tali da giustificare l'overhead dovuto:
    - alla creazione dei thread e dei task
    - alla gestione della decomposizione parallela
  - per la legge di Amdahl il passo 3 deve essere veloce:
    - fattore “seriale” che limita lo speed-up massimo

Decomposizione Parallela (3)

- ***`ExecutorService`***: pool di thread di dimensione dell'ordine del numero di CPU fisiche disponibili
- La suddivisione in sotto-problemi del passo 2, viene resa con la creazione di task dedicati alla loro soluzione

- Le molte varianti possono essere inquadrate lungo diverse coordinate; tra queste:
  - *Decomponibilità* – la suddivisione in sotto-problemi può essere:
    - *statica,* ovvero svolta al passo 1
    - *dinamica,* ovvero svolta al passo 2
  - *Quantificazione* – la ricerca del risultato finale è:
    - *esistenziale,* conclusa quando è noto almeno uno dei parziali
    - *universale,* conclusa quando sono noti tutti i parziali

Esempio di Decomposizione Parallela

- Sommare tutti gli elementi di una array di enormi dimensioni
- Si utilizzano
  - **`Executor`** per ottenere un pool di thread
  - **`Future`** per contenere i risultati parziali
- Per questo esempio molto semplice:
  - numero di task (**`Callable`** e **`Future`**) creati
    - prevedibile a priori
    - numero di task creati == numero di CPU fisiche
    - L'overhead dovuto alla creazione dei task è ottimale
  - la parallelizzazione ha senso per array di dimensioni sufficientemente grandi da rendere questi costi relativamente trascurabili

✔ N.B.: non sempre il numero di task da creare è prevedibile a priori

Decomposizione Statica ed Universale: Somma degli Elementi di un Array (1)

```
public class ArraySumTask
  implements Callable<Integer> {

  private int[] array;

  private int low,high

  public ArraySumTask(int[] array, int lo, int hi) {
    this.array=array; this.low=lo; this.high=hi;
  }

  public Integer call() {
    int result = 0;
    for ( int i=this.low; i<this.high; i++) {
      result += this.array[i];
    }
    return result;
  }

}
```

Decomposizione Statica ed Universale: Somma degli Elementi di un Array (2)

```
public class ArraySum {

  private static int[] array = {1, 2, 3, …omissis…};

  private final static int NCPU =
     Runtime.getRuntime().availableProcessors();

  public static void main(String[] args) {
      ExecutorService pool =
            Executors.newFixedThreadPool(NCPU);
      List<Future<Integer>> pending =
                    new LinkedList<Future<Integer>>();
      int sliceSize = array.length / NCPU;
      …
```

Decomposizione Statica ed Universale: Somma degli Elementi di un Array (3)

```
…    // elaborazione parallela dei sotto-problemi
     int l = 0;
     for (int w=0; w<NCPU; w++ ) {
       int h = Math.min(lo + sliceSize,array.length);
       pending.add(pool.submit(
         new ArraySumTask(array, l, h)
       ));
       l = h + 1;
     }
     int sum = 0;
     // ricomposizione dei risultati parziali
     for ( Future<Integer> : pending ) {
         sum += f.get();
     }
     System.out.println( "The sum is "+sum);
     pool.shutdown();
   }
}
```

Decomposizione Statica ed Esistenziale (1)

- Classe di problemi risolti al primo parziale disponibile
- Ad es.:

trovare gli indici di riga e di colonna di un qualsiasi elemento non nullo in una matrice sparsa di grandi dimensioni

- I risolutori ancora attivi al momento della scoperta del primo parziale vanno interrotti
  - non interessati ad eventuali altre soluzioni

Decomposizione Statica ed Esistenziale (2)

- Soluzione generica a questo tipo di problemi:
  - sono dati un certo numero di **`Solver`** per un dato problema, ciascuno restituisce un valore di tipo **`Result`**
  - i **`Solver`** cercano la soluzione concorrentemente

- Si usano in particolare i “CompletionService”
  - ***`CompletionService`*** **`>>`**
    - **`ExecutorCompletionService >>`**
- Oltre a ..
  - ***`Executor`***
  - ***`Future`***

`java.util.concurrent.CompletionService` (1)

- Interfaccia che permette di disaccoppiare la sottomissione di nuovi *task* da eseguire, dalla consumazione dei corrispondenti risultati
- I risultati sono consumati nell'ordine temporale in cui i task sono completati, indipendentemente dall'ordine di sottomissione
- Come può essere implementata?

java.util.concurrent

Interface CompletionService<V>

All Known Implementing Classes:

ExecutorCompletionService

`CompletionService` (2)

- In effetti può essere facilmente realizzata affiancando al classico ***`ExecutorService`*** una ***`BlockingQueue`*** (detta *completion queue*)
  - i worker thread vi depositano i risultati (ancora incapsulati in un ***`Future`***) non appena completano il loro task
  - per calcolare il risultato *globale* ricomponendo i *parziali*
    - estrazioni bloccanti dalla *completion queue*
    - evase solo al momento del completamento dei parziali
- Per questo motivo i metodi di ***`CompletionService`*** sembrano "mescolare" quelli di ***`ExecutorService`*** con quelli di ***`BlockingQueue`***
  - Implementazione:

    `ExecutorCompletionService`

`CompletionService` (3)

Method Summary	
Future<V>	**poll()** Retrieves and removes the Future representing the next completed task or null if none are present.
Future<V>	**poll(**long timeout**, TimeUnit** unit) Retrieves and removes the Future representing the next completed task, waiting if necessary up to the specified wait time if none are yet present.
Future<V>	**submit(Callable**<V> task) Submits a value-returning task for execution and returns a Future representing the pending results of the task.
Future<V>	**submit(Runnable** task, V result) Submits a Runnable task for execution and returns a Future representing that task.Upon completion, this task may be taken or polled.
Future<V>	**take()** Retrieves and removes the Future representing the next completed task, waiting if none are yet present.

java.util.concurrent.ExecutorCompletionService

java.util.concurrent

Class ExecutorCompletionService<V>

java.lang.Object
  ↳ java.util.concurrent.ExecutorCompletionService<V>

All Implemented Interfaces:

CompletionService<V>

Constructor Summary

ExecutorCompletionService(Executor executor) Creates an ExecutorCompletionService using the supplied executor for base task execution and a LinkedBlockingQueue as a completion queue.

ExecutorCompletionService(Executor executor, BlockingQueue<Future<V>> completionQueue) Creates an ExecutorCompletionService using the supplied executor for base task execution and the supplied queue as its completion queue.

Generico Risolutore Concorrente (1)

```
void solve(Executor e,
           Collection<Callable<Result>> solvers)
                        throws InterruptedException {
  CompletionService<Result> ecs =
      new ExecutorCompletionService<Result>(e);
  int n = solvers.size();
  List<Future<Result>> futures = new
                     ArrayList<Future<Result>>(n);
  Result result = null;
   /*
      Esec. concorrente dei solver e calcolo di Result a seguire...
   */
   if (result != null)
       use(result); // utilizzo del risultato
}
```

Generico Risolutore Concorrente (2)

```
… try { // sottomissione di tutti i task

    for (Callable<Result> s : solvers)
      futures.add(ecs.submit(s));
    // raccolta dei risultati nell'ordine di completamento
    for (int i = 0; i < n; ++i) {
      try { Result r = ecs.take().get();
            if (r != null) {
               result = r; //al primo disponibile esci
               break;
            }
      } catch(ExecutionException ignore) {}
    }
  }
  // prova a cancellare i rimanenti
  // (eventualmente ancora in esecuzione)
  finally {
    for (Future<Result> f : futures)
      f.cancel(true);
  }
```

Framework per la Decomposizione Parallela

- I problemi decomponibili dinamicamente:
  - per la frequenza con la quale si presentano
  - visto i benefici ottenibili con scheduler dedicati che tengano esplicitamente conto della natura e delle relazioni intercorrenti tra i task sottomessi

  hanno meritato la progettazione di un intero Framework "Fork/Join" per la decomposizione parallela
- Semplificano ulteriormente la decomposizione parallela rispetto ai precedenti *executor service* in **`java.util.concurrent`**
- Introdotto in JDK 7.0

Motivazioni dei Fork/Join Framework

- La decomposizione in sottoproblemi non sempre è realizzabile *staticamente*
- Il numero ottimale di task da creare può essere
  - non banalmente prevedibile
  - dipendente dalla dimensione *N* del problema trattato
    - N.B. per problemi di dimensione troppo piccola la decomposizione risulta addirittura controproducente!
- Tipica situazione degli algoritmi *ricorsivi* operanti su strutture *dinamiche*
- I task di questi problemi
  - seguono una organizzazione regolare e strutturata
  - ammettono scheduling molto più efficaci di quelli ottenibili con uno scheduler *general-purpose*

Suddivisione Dinamica in Task

- I task devono ammettere, ricorsivamente, una decomposizione in sotto-task di dimensioni più piccole
- In sostanza la decomposizione in task viene rimandata a tempo dinamico
  - all'inizio task a grana grossa
  - quindi a grana via via più fine
- I task di dimensioni sufficientemente piccole possono più convenientemente essere risolti serialmente
- Serve una soglia: `SEQUENTIAL_THRESHOLD`
  - task più piccoli: soluzione seriale
  - task più grandi: decomposizione parallela
- Si minimizza il numero di task creati
  - $N <$ `SEQUENTIAL_THRESHOLD`: un solo task
- Si basa sulle tecniche già viste + tecniche di *work stealing*

Struttura dei Task

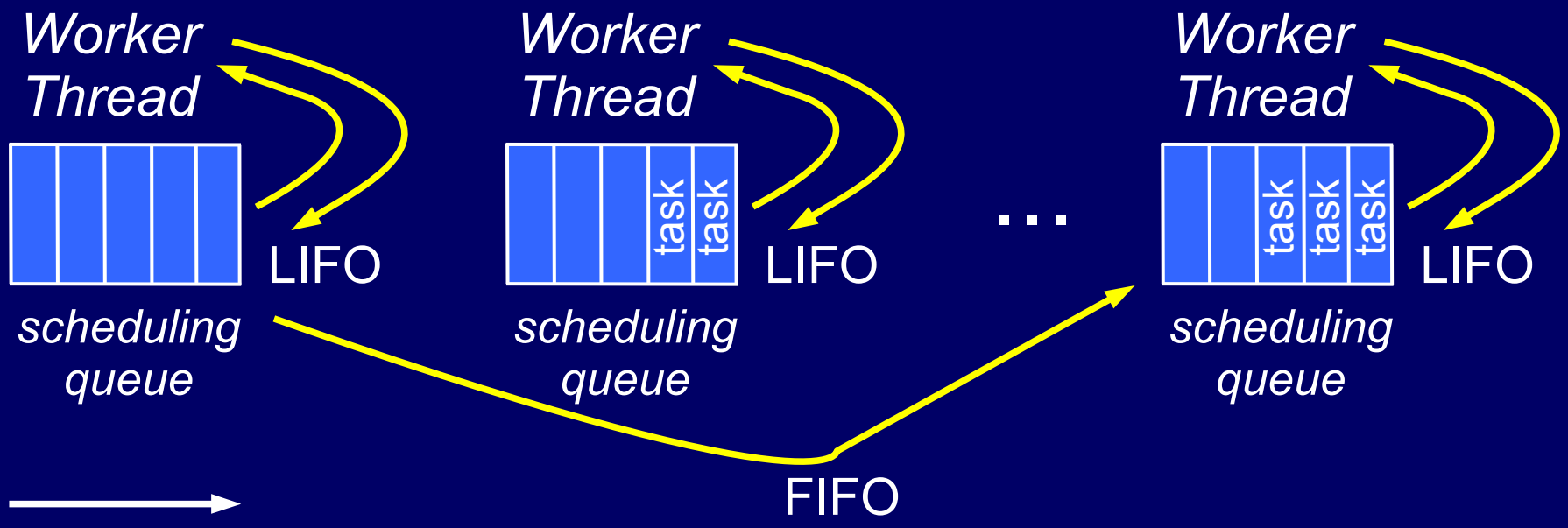
```
if ( N < SEQUENTIAL_THRESHOLD ) {
    // più conveniente una soluzione seriale

   << soluzione diretta (e seriale) del problema >>

} else {
    // più conveniente una decomposizione parallela

    << decomposizione in sotto-task >>
    << fork di tutti i sotto-task >>
    << join di tutti i sotto-task >>
    << ricomposizione dei risultati parziali >>
}
```

- Osservare che
  - Il numero di task sottomessi cresce esponenzialmente
  - La dimensione dei task sottomessi decresce esponenzialmente

Work Stealing (1)

Ogni worker thread mantiene una propria *scheduling queue*

- La scheduling queue è acceduta
  - LIFO: per il processamento “ordinario” dei propri task da parte dei worker thread
  - FIFO: per il reperimento di task degli altri worker thread da parte dei worker thread altrimenti inattivi



Al solito: n. di worker thread == n. CPU disponibili

fork() & join(): Terminologia

- Stiamo per usare nuovamente una coppia di primitive
- Si chiamano ancora una volta **`fork()`** e **`join()`**
- Sono i nomi più diffusi per una coppia di operazioni supportate dai framework di decomposizione parallela
  - il framework Fork & Join di Java 7+ è solo un esemplare

- E' tuttavia opportuno chiarire che sebbene correlate alle omonime primitive astratte viste nella prima parte del corso, sono concettualmente ben diverse
- ✔ In particolare lavorano a livello di *task* e servono ad informare lo scheduler delle relazioni tra gli stessi
  - **`fork()`**: crea sotto-task
  - **`join()`**: attende la terminazione di un task

Work Stealing (2)

- I sotto-task generati (con **`fork()`**) da un worker thread sono inseriti nella sua coda
- I worker thread processano LIFO i propri task
  - si tratta dei task più “piccoli”
- Se una *scheduling queue* si svuota, il worker thread cerca di *rubare* task da un altro worker thread scelto casualmente con estr. FIFO
  - Si tratta dei task più “grossi”
- Se non riesce ad ottenere il task, ripete il tentativo dopo un'attesa casuale (*backoff*)
- La **`join()`** viene eseguita da un worker thread con cicli di attesa semi-attiva (si usa il metodo **`isDone()`**) intervallati dall'esecuzione di altri task se disponibili
  - >> *Eliminazione delle attese passive e dei context-switch*

Work Stealing (3)

Alcune osservazioni:

- Le attese (di ogni tipo, attese e passive) sono minimizzate
- Il lavoro ordinario ed il *work stealing* insistono su parti opposte della ***`Deque`*** limitando la competizione
  - si tratta di ***`Deque`*** che consentono ins./estr. concorrenti alle estremità opposte
- L'attività di work stealing reperisce i task di dimensioni maggiori
  - dovrebbero fornire più lavoro
  - minimizza la necessità di ripetere l'operazione nell'immediato futuro e quindi minimizza la competizione
- Al diminuire della granularità minima dei task (`SEQUENTIAL_THRESHOLD`)
  - è più probabile che il *work stealing* consenta di distribuire uniformemente il carico
  - è più probabile che si finisca per decomporre anche task che conveniva risolvere sequenzialmente

Fine-Tuning dei Parametri

- Due principali parametri semplici da configurare
  - Numero di worker-thread
    - la scelta di default di **`ForkJoinPool`** (uguale al numero delle CPU fisiche disponibili) è la più sensata
    - ma solo per task *CPU-bound* >>
  - **`SEQUENTIAL_THRESHOLD`**: soglia sulla dimensione dei problemi oltre la quale avviare la decomposizione parallela
    - la documentazione consiglia di avere task di base consistenti almeno in un *numero di istruzioni* compreso tra 100 e 10.000
    - solo sperimentalmente si può determinare una buona soglia
      - *robustezza del setting:* la migliore distribuzione del carico ottenibile con task a grana troppo fine compensa il costo della creazione dei task in eccesso
      - *ogni scelta ragionevole non si discosta eccessivamente dall'ottimo*

Java Fork/Join Framework (1)

- Principali 4 classi di interesse in **`java.util.concurrent`**:
  - Per gli executor service:

    `ForkJoinPool`: specializza ***`ExecutorService`***

  - Per i task:

    `ForkJoinTask`: classe base astratta
    - Per i task che restituiscono un valore

      `RecursiveTask<V>`: specializza (ma non implementa) ***`Callable`*****`<V>`**
    - Per i task che *non* restituiscono un valore

      `RecursiveAction`: specializza (ma non implementa) ***`Runnable`***

ForkJoinTask

jsr166y

Class ForkJoinTask<V>

java.lang.Object
  └ jsr166y.ForkJoinTask<V>

All Implemented Interfaces:
Serializable, Future<V>

Direct Known Subclasses:
RecursiveAction, RecursiveTask

ForkJoinTask<V>	**fork**() Arranges to asynchronously execute this task.
boolean	**isDone**()
V	**join**() Returns the result of the computation when it is done.
V	**invoke**() Commences performing this task, awaits its completion if necessary, and returns its result, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.

- Classe base astratta per modellare *Task*

RecursiveTask<V>

- Modella task che restituiscono risultati

Constructor Summary

RecursiveTask()

Method Summary

protected abstract V	compute() The main computation performed by this task.

RecursiveAction

- Modella task che non restituiscono risultati

Constructor Summary

RecursiveAction()

Method Summary

protected abstract void	compute() The main computation performed by this task.

ForkJoinPool

- Realizza l'*executor service*
  - Per default il numero di *worker thread* è pari al numero di CPU fisiche disponibili

Constructor Summary	
ForkJoinPool() Creates a ForkJoinPool with parallelism equal to Runtime.availableProcessors(), using the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode.	
ForkJoinPool(int parallelism) Creates a ForkJoinPool with the indicated parallelism level, the default thread factory, no UncaughtExceptionHandler, and non-async LIFO processing mode.	
<T> T	**invoke**(ForkJoinTask<T> task) Performs the given task, returning its result upon completion.

Framework Fork/Join: Calcolo della Dimensione di una Directory (1)

- Problema di decomposizione parallela dinamica ed universale
- Calcolo della dim. directory, interpretata come somma delle dimensioni di tutti i file contenuti, anche indirettamente
- ✔ Attenzione, comporta I/O

```
import […omissis…]
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class DirSizer {
    private static class SizerTask extends RecursiveTask<Long> {
        @Override
        protected Long compute() { /*… codice nella successiva slide… */ }
    }

    public static void main(String[] args) {
        final ForkJoinPool pool = new ForkJoinPool();
        try { System.out.println("Total size (including rec. subdirs): "+
            pool.invoke(new SizerTask(new File(args[0]))));
        } finally { pool.shutdown(); }
    }
}
```

Framework Fork/Join: Calcolo della Dimensione di una Directory (2)

```
private static class SizerTask extends RecursiveTask<Long> {
  private long accumulator;
  private final File dir;

  public SizerTask(File dir) { this.accumulator = 0; }

  @Override
  protected Long compute() {
      final List<SizerTask> tasks = new ArrayList<>();
      final File[] children = this.dir.listFiles();
      if (children!=null) {
          for (File child : children) {
          if (child.isFile()) { this.accumulator += file.length(); }
          else { /* it is a child directory */
              final SizerTask task = new SizerTask(child);
              task.fork();
              tasks.add(task);
          }
      }
      for (final SizerTask task : tasks) /* sum-up */
         this.accumulator += task.join();
      return this.accumulator;
  }}
```

Framework Fork/Join: Somma degli Elementi di un Array (1)

- Anche se i benefici (rispetto all'uso diretto del LEF) sono più evidenti con riferimento ad un problema di decomposizione dinamica, il framework fork/join può agevolmente essere usato anche per problemi statici (ed universali)

```
static final ForkJoinPool fjPool = new ForkJoinPool();
static final int SEQUENTIAL_THRESHOLD = 4096;

class SumArray extends RecursiveTask<Long> {
   private int low;
   private int high;
   private int[] array;

   SumArray(int[] arr, int lo, int hi) {
      array = arr; low = lo; high = hi;
   }

   protected Long compute() {/* … calcolo slide successiva … */ }
}

long sumArray(int[] array) {
   return fjPool.invoke(new SumArray(array,0,array.length));
}
```

Framework Fork/Join: Somma degli Elementi di un Array (2)

```
class SumArray extends RecursiveTask<Long> {
   …@Override
   protected Long compute() {
      If (high-low <= SEQUENTIAL_THRESHOLD) {
         long sum = 0;
         for(int i=low; i < high; ++i)
            sum += array[i];
         return sum;
      } else {
          int mid = low + (high - low) / 2;
          SumArray left  = new SumArray(array, low, mid);
          SumArray right = new SumArray(array, mid, high);
          left.fork();
          long rightAns = right.compute();
          long leftAns  = left.join();
          return leftAns + rightAns;
      }
    }…
}
```

Framework Fork/Join: Dec. Parallela Dinamica ed Esistenziale

- Es. *trovare un nodo con una certa proprietà in un albero di enormi dimensioni*
  - Al primo trovato si può interrompere la ricerca
- Supporto diretto dal framework aggiunto in Java 8
  - Nuova sottoclasse di **`ForkJoinTask`**
  - **`java.util.concurrent.CountedCompleter`**
- In alternativa, è possibile:
  - centralizzare il risultato utilizzando una classe a cui accedere concorrentemente
    - ✔ introduce punti di sincronizzazione: riduce lo speed-up
  - far terminare immediatamente i task non appena rilevano che il risultato è stato già ottenuto

Approfondimenti

- *http://gee.cs.oswego.edu/dl/papers/fj.pdf*