Prova Intermedia del 3 Novembre 2015 – traccia B

Domanda 1. In riferimento al modello di programmazione ForkJoin con thread espliciti, discutete brevemente il modello di memoria, illustrando:

- l'accessibilità di diverse zone di memoria da parte di thread diversi;
- la modalità con cui il programmatore può condividere informazioni tra thread diversi.

Domanda 2. Rispondete alle seguenti domande, motivando la risposta.

- Un programma contiene una funzione m parallelizzabile in cui viene speso $x\%$ del tempo di esecuzione.
  - Di quanti processori abbiamo bisogno al fine di velocizzare l'esecuzione di m e ottenere uno speedup 2 per l'intero programma? Stimare il numero di processori in funzione di x.
  - È sempre possibile ottenere speedup 2? Se si, spiegare il perché. Se no, dire per quali valori di x risulta impossibile.
- Per risolvere uno stesso problema due programmatori hanno sviluppato due diversi programmi: il programma $\mathcal{A}_1$ esegue work t_1 e ha una frazione di codice sequenziale $s_1 = 70\%$, il programma $\mathcal{A}_2$ esegue work t_2 e ha una frazione di codice sequenziale $s_2 = 30\%$. Supponete che $t_1 = t_2$:
  - se $\mathcal{A}_1$ e $\mathcal{A}_2$ sono eseguiti su una stessa piattaforma con P processori, quale esecuzione sarà più veloce?
  - se $\mathcal{A}_1$ è eseguito su una piattaforma con P processori e $\mathcal{A}_2$ su una piattaforma con $\frac{P}{2}$ processori, quale esecuzione sarà più veloce?

  Se la risposta dipende dal numero dei processori, spiegate sotto quali ipotesi l'esecuzione del programma $\mathcal{A}_1$ è più / meno veloce dell'esecuzione del programma $\mathcal{A}_2$.

Domanda 3. Considerate il codice per la somma degli elementi di un array visto a lezione, modificando il metodo `run` come segue:

```
public void run() {
    for (int i=lo; i<hi; i++)
        ans += processValue(arr[i]);
}
```

Anziché aggiungere direttamente al campo `ans` il valore `arr[i]`, il metodo `run` somma un valore calcolato a partire da `arr[i]` tramite il metodo `processValue`. Il metodo `processValue` ha però tempi di esecuzione diversi nelle varie chiamate: in questo esercizio, per ogni $i \geq 0$, la chiamata `processValue(arr[i])` richiede tempo 2^i. Immaginate ora di suddividere la computazione tra due soli thread, come mostrato di seguito:

```
static int sum(int[] arr) throws java.lang.InterruptedException {
  int M = ... ;
  SumThread leftThread  = new SumThread(arr,0,M);
  SumThread rightThread = new SumThread(arr,M,arr.length);

  leftThread.start();
  rightThread.start();
```

```
    leftThread.join();
    rightThread.join();

    return leftThread.ans+rightThread.ans;
  }
```

Rispondete alle seguenti domande:

- Cosa comporta scegliere `M=arr.length/2`, come abbiamo fatto nel caso della somma? I tempi di esecuzione dei due thread sono bilanciati?
- Come sceglierestе `M` per bilanciare meglio il carico? Date un'intuizione e calcolate una stima del valore ottimale per `M`.

  Suggerimento. Assumete che il tempo di esecuzione di un thread sia dato dalla somma dei tempi di esecuzione delle chiamate al metodo `processValue` effettuate dal thread stesso e ricordate la serie geometrica:

  $$\sum_{x=0}^{t} 2^x = 2^{t+1} - 1 \approx 2^{t+1}$$

  Calcolate il tempo di esecuzione di ciascuno dei due thread ed imponete che i due tempi si eguaglino.
- Se avessimo usato l'approccio basato su *divide et impera* (senza cutoff sequenziale), in questo esempio sarebbe stato altrettanto cruciale bilanciare bene il carico? Ovvero, avremmo potuto scegliere `M=arr.length/2`? Motivare la risposta.

Domanda 4. Dopo aver introdotto il problema delle somme prefisse, descrivete l'algoritmo di Ladner & Fischer (basato su alberi bilanciati) ed analizzatene work e span.