Università degli Studi di Roma Tre

Calcolo Parallelo e Distribuito

Parallelizzare Quicksort in Python

Claudio Pisa Salvatore Tranquilli

4 dicembre 2006

Sommario

Implementiamo in Python l'algoritmo di ordinamento Quicksort utilizzando il paradigma di programmazione message-passing (implementazione MPI) e la programmazione multithread. Dopo di che confrontiamo i tempi di esecuzione con quelli dell'implementazione seriale.

Indice

1 Introduzione **2**

2 Scelte implementative **2**

3 Quicksort Parallelo **4**
- 3.1 Tecniche di decomposizione 4
- 3.2 Quicksort parallelo (decomposizione ricorsiva) 5
- 3.3 Quicksort parallelo (decomposizione dell'input) 7
  - 3.3.1 Quicksort parallelo (decomposizione dell'input) per architetture distribuite 8
  - 3.3.2 Quicksort parallelo (decomposizione dell'input) per shared address space 10

4 Prove sperimentali **12**
- 4.1 Metriche e considerazioni 12
- 4.2 Risultati 13
- 4.3 Conclusioni 15

1 Introduzione

Grazie alla possibilità di implementazioni molto efficienti, il Quicksort può essere considerato l'algoritmo di ordinamento più utilizzato nel Mondo [QW-06] . Nella sua formulazione seriale permette di ordinare una sequenza di lunghezza n in un tempo che nel caso peggiore è $\Theta\left(n^2\right)$ ma è mediamente $\Theta\left(n \cdot \log n\right)$.

L'algoritmo, ideato da *Charles Anthony Richard Hoare*, data una sequenza da ordinare, ricorsivamente sceglie un elemento *pivot* e divide la sequenza scambiando la posizione degli elementi. Il principale svantaggio del Quicksort rispetto ad altri algoritmi di ordinamento è che se il Quicksort non viene implementato attentamente, può creare dei problemi non riscontrabili in fase di debug, ovvero funzionare solo con alcune classi di sequenze.

In pseudocodice possiamo scrivere [IPC-03]:

```
1  procedure QUICKSORT(A,q,r)
   begin
     if q<r then
     begin
       x := A[q];
6      s := q;
       for i := q+1 to r do
         if A[i]<=x then
         begin
           s := s+1;
11         swap(A[s],A[i]);
         end if
       swap(A[q],A[s]);
       QUICKSORT(A,q,s);
       QUICKSORT(A,s+1,r);
16   end if
   end QUICKSORT
```

2 Scelte implementative

Per l'implementazione è stato scelto il linguaggio di scripting Python[1] . Sono diversi i motivi che hanno portato a questa scelta:

- Python è indipendente dall'architettura (come Java),
- è un linguaggio che permette la scrittura di codice più velocemente rispetto a linguaggi di più basso livello (ad esempio rispetto al C),
- permette la scrittura di codice multi-thread ed è compatibile con le librerie MPI esistenti.

[1] http://www.python.org/

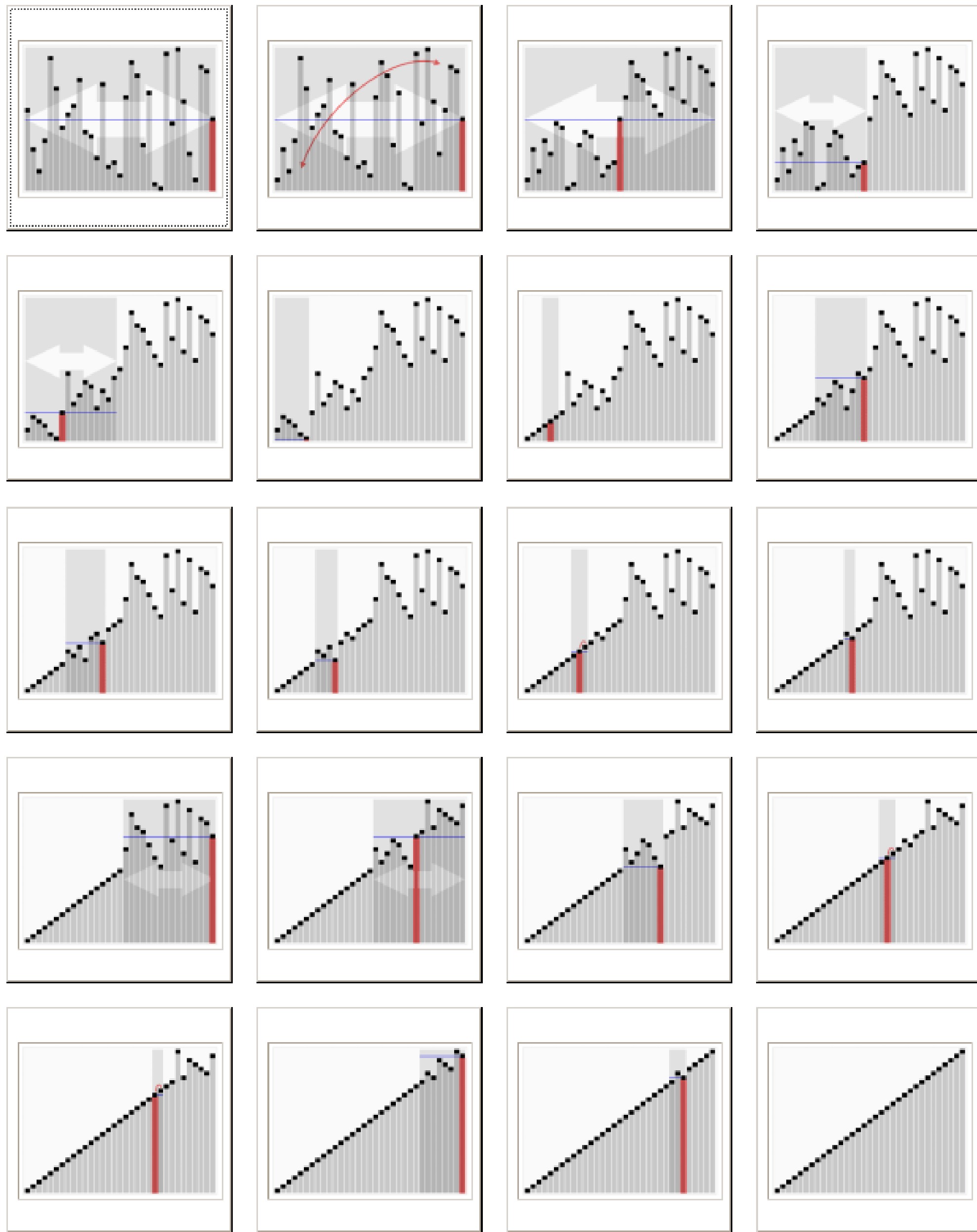

Figura 1: Schematizzazione grafica del funzionamento dell'algoritmo Quicksort [QW-06]

Per poter scrivere codice MPI in Python sono disponibili diverse soluzioni:

- l'uso di un interprete Python modificato per potersi interfacciare con la sottostruttura MPI,
- l'uso di un modulo Python che permette di utilizzare le più comuni istruzioni MPI.

La scelta è ricaduta sulla seconda possibilità[2], in quanto rende il codice più facilmente portabile da un sistema all'altro ed alcune librerie hanno l'API molto simile all'API MPI C vista durante il corso.

Per poter scrivere codice multithread in Python sono disponibili (nella distribuzione standard dell'interprete Python) le seguenti librerie:

- `thread`, che fornisce le API base per lo sviluppo multithread,
- `threading`, che amplia le API `thread` con uno stile simile a Java, velocizzando di fatto lo sviluppo thread safe.

La scelta è ricaduta sulla seconda possibilità, in quanto permette di scrivere codice probabilmente più sicuro.

Nel presente documento abbiamo cercato di riportare i frammenti di codice più significativi. Per la versione completa degli script rimandiamo ai file allegati.

3 Quicksort Parallelo

3.1 Tecniche di decomposizione

Per parallelizzare l'algoritmo Quicksort si può pensare di applicare una tecnica di decomposizione tra quelle viste nella teoria, ovvero [IPC-03]:

- decomposizione ricorsiva
- decomposizione esplorativa
- decomposizione dei dati
- decomposizione speculativa

Per quanto riguarda le decomposizioni esplorativa e speculativa, queste non possono essere applicate all'algoritmo Quicksort, perché sarebbe privo di senso.

La decomposizione ricorsiva sembra la decomposizione più naturale per questo algoritmo: basta associare un nuovo task ad ogni chiamata ricorsiva.

Per quanto riguarda la decomposizione dei dati, la decomposizione dell'input è applicabile ed è infatti utilizzata dalle implementazioni più efficienti del Quicksort.

[2] Abbiamo utilizzato la libreria MyMPI v.1.1: http://peloton.sdsc.edu/~tkaiser/mympi/

3.2 Quicksort parallelo (decomposizione ricorsiva)

Utilizzando la decomposizione ricorsiva, otteniamo la seguente implementazione:

```
from threading import Thread

class qsorter(Thread):
        L=[] #list
        p=None  #pivot
        R=[] #result
        #
        def __init__(self,A,pivot=None):
                try:
                        self.L=A[:]
                except TypeError: #the list is empty
                        self.L=[]
                self.p=pivot
                Thread.__init__(self)
        #__init__
        #
        def run(self):
                #do the sorting, returning an
                #ordered list (in self.R) and
                #creating new threads
                r=len(self.L)
                if r<=1:        #list too small
                        self.R=self.L
                else:           #quicksort
                        x=self.p
                        LA=[]
                        RA=[]
                        for i in range(r):
                                e=self.L[i]
                                if e<=x:
                                        LA.append(e)
                                else:
                                        RA.append(e)

                        #recursive calls
                        nqsl=qsorter(LA,pivot_selection(LA))
                        nqsr=qsorter(RA,pivot_selection(RA))

                        nqsl.start()
                        nqsr.start()
                        nqsl.join()
                        nqsr.join()

                        #set the result
                        self.R=nqsl.getresult()
                                +nqsr.getresult()
        #run
        def getresult(self):
                return self.R
```

```
        #getresult
#qsorter

def quicksortP(L):
        nq=qsorter(L)
        nq.run()
        return nq.getresult()
#quicksortP
```

Dove la funzione `pivot_selection` viene scelta tra le funzioni `pivot_selection0`, `pivot_selection1` e `pivot_selection2`, le quali selezionano come elemento *pivot* rispettivamente il primo elemento della sequenza utilizzata come argomento, un elemento pseudo-casuale e la media degli elementi.

I test su questo script hanno dimostrato che nel caso di liste sufficientemente lunghe, la memoria del sistema si esaurisce. Questo succede perché il programma crea un albero di chiamate ricorsive che si espande in ampiezza, le cui foglie corrispondono al passo base della ricorsione. Una volta create le foglie, l'albero smette di espandersi ed inizia a contrarsi utilizzando una contrazione in ampiezza (vedere Figura 2). L'occupazione in memoria di questo albero è $O\left(\sum_{i=0}^{\log n} 2^i\right)$, dove n è la lunghezza della sequenza da ordinare.

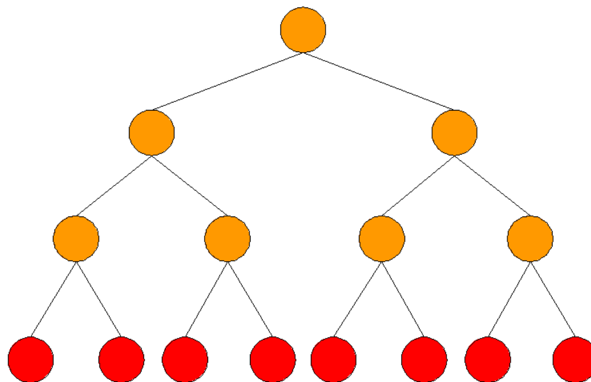

Figura 2: Albero delle chiamate ricorsive nel caso di chiamate ricorsive parallele

Abbiamo quindi modificato il programma cambiando l'ordine delle chiamate ricorsive.

Con questa modifica, l'albero delle chiamate ricorsive veniva espanso e contratto in profondità, riducendo l'occupazione di memoria, pari alla profondità dell'albero $l \simeq \log_2 n$. In questo modo, però, abbiamo serializzato le chiamate ricorsive, e quindi poteva essere eseguito *un solo task per volta* (vedere Figura 3), contro un massimo di n task eseguibili parallelamente del programma precedente (vedere Figura 2).

```
class qsorter(Thread):
        [...]
        def run(self):
                [...]
                        #recursive calls
                        nqsl=qsorter(LA, pivot_selection(LA))
                        nqsr=qsorter(RA, pivot_selection(RA))

                        nqsl.start()
                        nqsl.join()
                        nqsr.start()
                        nqsr.join()
                [...]
```

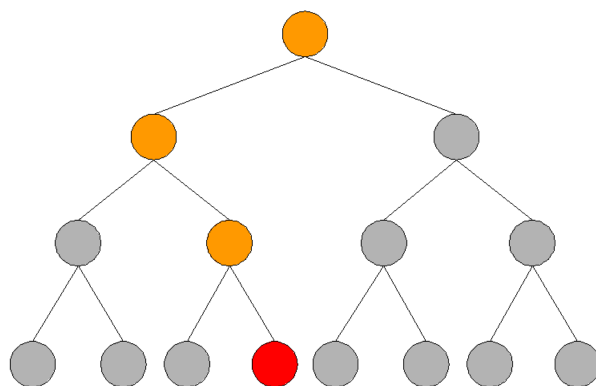

Figura 3: Albero delle chiamate ricorsive nel caso di chiamate ricorsive serializzate

3.3 Quicksort parallelo (decomposizione dell'input)

Applicando la decomposizione dell'input, otteniamo un algoritmo che permette un grado di parallelismo più alto di quanto visto con la decomposizione ricorsiva.

Questo algoritmo lavora come segue. Sia L una lista di n elementi e p sia il numero di elementi di elaborazione (EL da ora in poi). Ad ogni EL viene assegnato una sottolista ELL di elementi contigui di L. Le etichette degli EL definiscono l'ordinamento globale della sequenza ordinata.

Dopo la scelta di un elemento pivot (comune a tutti gli EL) ogni processo compie un ordinamento locale dividendo la propria sottolista in due liste ELL_s ed ELL_l,contenenti rispettivamente gli elementi di ELL minori del pivot e gli elementi di ELL maggiori del pivot.

A questo punto avviene un riordinamento globale: ogni EL scrive in una lista globale detta $smaller$ la sua ELL_s e in una lista globale detta $larger$ la sua ELL_l; questa scrittura avviene secondo l'etichettatura degli EL.

Ora si hanno due liste parzialmente ordinate. Una parte degli EL ordinerà la lista $smaller$, il resto ordinerà la lista $larger$. Il processo continua ricorsivamente e termina quando la lista $smaller/larger$ viene assegnata ad un solo EL che la ordina con il normale Quicksort seriale.

3.3.1 Quicksort parallelo (decomposizione dell'input) per architetture distribuite

La formulazione parallela del Quicksort con passaggio di messaggi è stata implementata attraverso l'uso di MPI. Questo paradigma permette la realizzazione di programmi paralleli attraverso l'uso contemporaneo di più workstation.

Il vantaggio di questa soluzione va cercato nell'economicità della stessa: possiamo utilizzare più macchine generiche per raggiungere capacità di calcolo parallelo che eguagliano (o superano!) una macchina shared space, notoriamente più costosa.

Lo svantaggio è legato all'uso di una rete di calcolatori tra le workstation, che introduce un overhead nelle comunicazioni che porta ad un degrado delle prestazioni.

```
from mpi import *

#split pivot in two parts, smaller contains elements smaller
    than the pivot, larger contains elements larger or equal
    than the pivot
def split(subVector,smaller,larger,pivot):
  for i in subVector:
    if i>=pivot:
      larger.append(i)
    else:
      smaller.append(i)

#flat transform list of lists l1 in a flat list l2
def flat(l1,l2):
  for i in l1:
    for j in range(0,len(i)):
      l2.append(i[j])

def sort(subVector,subGroup,myId,subProcessNumber):
  smaller=[]
  larger=[]
  leng=0
  localSmaller=range(0,mpi_comm_size(subGroup))
  localLarger=range(0,mpi_comm_size(subGroup))
  globalSmaller=[]
  globalLarger=[]

  #recursive case
  if mpi_comm_size(subGroup)>1:
    #if subVector is empty (more processes than necessary),
        the process do not collaborate for pivot selection
    if len(subVector)==0:
```

```
      pivot_selection(subGroup)
    else:
32    #split pivot in two parts, smaller contains elements
          smaller than pivot, larger contains elements larger
           than pivot or equal to pivot
      split(subVector,smaller,larger,pivot_selection(
          subVector,subGroup))
    #broadcast smaller and larger to other process in the
        group
    for i in range(0,mpi_comm_size(subGroup)):
      leng=mpi_bcast(len(smaller),1,MPI_INT,i,subGroup)
37    localSmaller[i]=mpi_bcast(smaller,leng[0],MPI_INT,i,
          subGroup)
      leng=mpi_bcast(len(larger),1,MPI_INT,i,subGroup)
      localLarger[i]=mpi_bcast(larger,leng[0],MPI_INT,i,
          subGroup)
    #execution of a global rearrangement
    smaller=[]
42  larger=[]
    for i in range(0,len(localSmaller)):
      smaller.append(localSmaller[i].tolist())
    for i in range(0,len(localLarger)):
      larger.append(localLarger[i].tolist())
47  #partition of process to sort smaller and larger parts
        separately, recursive call of sort function
    flat(smaller,globalSmaller)
    flat(larger,globalLarger)
    #if a empty vector is assigned to more than one process
        it is useless continue the sorting
    if len(globalSmaller)+len(globalLarger)==0:
52    return []
      S=(len(globalSmaller)*(float(subProcessNumber)/(len(
          globalSmaller)+len(globalLarger))))+0.5
    if int(S)==0:
      S=1
    if(myId<int(S)):
57    smallerGroup=mpi_comm_split(subGroup,0,myId)
      leng=mpi_comm_size(smallerGroup)
      if myId<leng-1:
      #if the process is the last in the group must sort last
           elements
        return sort(globalSmaller[myId*int(len(globalSmaller)
            /leng):(myId+1)*int(len(globalSmaller)/leng)],
            smallerGroup,myId,leng)
62    else:
        return sort(globalSmaller[myId*int(len(globalSmaller)
            /leng):len(globalSmaller)],smallerGroup,myId,leng
            )
    else:
      largerGroup=mpi_comm_split(subGroup,1,myId-int(S+1))
      myId=myId-int(S)
67    leng=mpi_comm_size(largerGroup)
      if myId<leng-1:
        #if the process is the last in the group must sort
```

```
            last elements
          return sort(globalLarger[myId*int(len(globalLarger)/
            leng):(myId+1)*int(len(globalLarger)/leng)],
            largerGroup,myId,leng)
        else: #base case
          return sort(globalLarger[myId*int(len(globalLarger)/
            leng):len(globalLarger)],largerGroup,myId,leng)

  else:
    #the entire larger/smaller part is assigned to only one
        process, sort by serial quicksort and print the final
        result
    return QuickSort(subVector)
```

3.3.2 Quicksort parallelo (decomposizione dell'input) per shared address space

La formulazione shared space del Quicksort è stata implementata utilizzando la libreria `threading` di Python.

Il seguente script segue una logica simile alla versione MPI. La differenza sta nel fatto che ora non c'è bisogno di scambio di messaggi, in quanto la memoria di lavoro è accessibile da parte di tutti gli elementi di eleborazione. Lo svantaggio è che bisogna gestire attraverso locking l'accesso alla risorse.

```
from threading import *

#recursive sort function
def sort(myId,subGroup,subVector,subGroupVector):
  if myId==subGroup[0]:
    eventList[myId].set()
    globalSmaller[myId]=[]
    globalLarger[myId]=[]
  if myId==subGroup[len(subGroup)-1]:
    eventList2[myId].set()
  smaller=[]
  larger=[]
  #if a empty vector is assigned to subGroup it is useless to
      continue the sorting
  if len(subGroupVector)==0:
    return []
  if len(subGroup)>1:
    split(subVector,smaller,larger,pivot_selection(
        subGroupVector))
    #global reorganization of subGroupVector in smaller and
        larger vectors smaller
    eventList[myId].wait()
    globalSmaller[subGroup[0]][len(globalSmaller[subGroup
        [0]]):len(globalSmaller[subGroup[0]])  len(smaller)]=
        smaller
    eventList[myId].clear()
    if myId==subGroup[len(subGroup)-1]:
```

```
                        eventList[subGroup[0]].set()
    else:
      eventList[myId+1].set()
                   #larger
    eventList[myId].wait()
                  globalLarger[subGroup[0]][len(globalLarger[
                       subGroup[0]]):len(globalLarger[subGroup
                       [0]])+len(larger)]=larger
                  eventList[myId].clear()
    if myId==subGroup[len(subGroup)-1]:
      eventList[subGroup[0]].set()
    else:
                              eventList[myId+1].set()
    #assign thread to sorting of smaller or larger vectors
    eventList[myId].wait()
    S=(len(globalSmaller[subGroup[0]])*(float(len(subGroup))
         /(len(globalSmaller[subGroup[0]])+len(globalLarger[
         subGroup[0]]))))+0.5
    if int(S)==0:
      S=1
    else:
      S=int(S)
    smallerGroup=subGroup[0:S]
    largerGroup=subGroup[S:len(subGroup)]
    larger=globalLarger[subGroup[0]]
    smaller=globalSmaller[subGroup[0]]
    eventList[myId].clear()
                  if myId==subGroup[len(subGroup)-1]:
                    eventList[subGroup[0]].set()
                  else:
                    eventList[myId+1].set()
    #barrier: synchronizing threads
    eventList2[myId].wait()
    eventList2[myId].clear()
    if myId!=subGroup[0]:
      eventList2[myId-1].set()
      #recursive call of sort
    if myId in smallerGroup:
      if myId==smallerGroup[len(smallerGroup)-1]:
        return sort(myId,smallerGroup,smaller[(myId-
             smallerGroup[0])*int(len(smaller)/len(
             smallerGroup)):len(smaller)],smaller)
      else:
        return sort(myId,smallerGroup,smaller[(myId-
             smallerGroup[0])*int(len(smaller)/len(
             smallerGroup)):(myId+1-smallerGroup[0])*int(len(
             smaller)/len(smallerGroup))],smaller)
    else:
      if myId==largerGroup[len(largerGroup)-1]:
        return sort(myId,largerGroup,larger[(myId-largerGroup
             [0])*int(len(larger)/len(largerGroup)):len(larger
             )],larger)
      else:
        return sort(myId,largerGroup,larger[(myId-largerGroup
```

```
            [0])*int(len(larger)/len(largerGroup)):(myId+1-
            largerGroup[0])*int(len(larger)/len(largerGroup))
            ],larger)
    else:
      #sort base case
      return QuickSort(subVector)

#thread sorter definition
class qsorter(Thread):
  myId=0
  result=[]
  #
  def __init__(self,myId):
    self.myId=myId
    Thread.__init__(self)
  #__init__          #

  #run routine call sort for first time
  def run(self):
    if self.myId==int(sys.argv[1])-1:
      self.result=sort(self.myId,range(0,int(sys.argv[1])),
          vector[self.myId*int(len(vector)/int(sys.argv[1])):
          len(vector)],vector)
    else:
      self.result=sort(self.myId,range(0,int(sys.argv[1])),
          vector[self.myId*int(len(vector)/int(sys.argv[1]))
          :(self.myId+1)*int(len(vector)/int(sys.argv[1]))],
          vector)

  def getResult(self):
    return self.result
```

4 Prove sperimentali

4.1 Metriche e considerazioni

Il nostro obiettivo è stato quello di implementare l'algoritmo Quicksort in versione parallela e compararne la velocità rispetto al Quicksort sequenziale. Per fare ciò abbiamo preso in esame le seguenti metriche [IPC-03]:

- **tempo di esecuzione** T_p, è il tempo effettivo utilizzato dal programma per eseguire l'ordinamento. È dato dal tempo trascorso dall'inizio dell'esecuzione del programma parallelo alla terminazione dell' ultimo elemento di processamento.

- **tempo totale di overhead** T_o, misura il tempo in più utilizzato da un programma parallelo rispetto alla versione sequenziale. È dato dalla formula $T_o = pT_p – T_s$, dove p è il numero di elementi di processamento, T_p il tempo di esecuzione descritto sopra e T_s è il tempo di esecuzione per il programma sequenziale.

- **speedup** S, è una misura dell'aumento di performance ottenuto dal programma parallelo rispetto al programma sequenziale. È dato dal rapporto tra T_s e T_p, $S = T_s/T_p$.
- **efficienza** E, è la misura della frazione di tempo utile di impiego di un elemento di elaborazione (rispetto al tempo totale di impiego). È dato dal rapporto tra speedup e numero di elementi di elaborazione, $E = S/p$.

4.2 Risultati

Nella tabella seguente sono visibili i **tempi di esecuzione medi** per gli algoritmi implementati. *Squicksort* corrisponde all'implementazione seriale, *MPIquicksort* all'implementazione parallela MPI e *t9.4.3aquicksort* all'implementazione multithread.

		Dimensione Istanza			
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
1 workstation GNU/Linux	Squicksort	0.11	1.39	9.32	19.45
7 workstation GNU/Linux	MPIquicksort(5)	0.08	0.63	3.57	7.64
	MPIquicksort(10)	0.15	0.7	3.29	6.85
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
Monolito (AIX)/Python	Squicksort	0.48	5.93	33.86	70.71
	t9.4.3aquicksort(5)	10.82	114.21	625.52	1309.71
	t9.4.3aquicksort(10)	18.33	175.71	928.68	1922.29
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
Monolito (AIX)/Jython	Squicksort	0.56	5.79	30.71	–
	t9.4.3aquicksort(5)	2.32	16.13	84.12	–
	t9.4.3aquicksort(10)	5.4	42.96	213.05	–

Nella tabella seguente sono mostrati i valori per lo **speedup medio** rispetto agli algoritmi implementati:

		Dimensione Istanza			
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
7 workstation GNU/Linux	MPIquicksort(5)	1.42	2.2	2.61	1.71
	MPIquicksort(10)	0.77	1.99	2.83	1.91
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
Monolito (AIX) Python	t9.4.3aquicksort(5)	0.04	0.05	0.05	0.04
	t9.4.3aquicksort(10)	0.03	0.03	0.04	0.02
	Algoritmo	**10˙000**	**100˙000**	**500˙000**	**10^6**
Monolito (AIX) Jython	t9.4.3aquicksort(5)	0.24	0.36	0.36	–
	t9.4.3aquicksort(10)	0.1	0.14	0.14	–

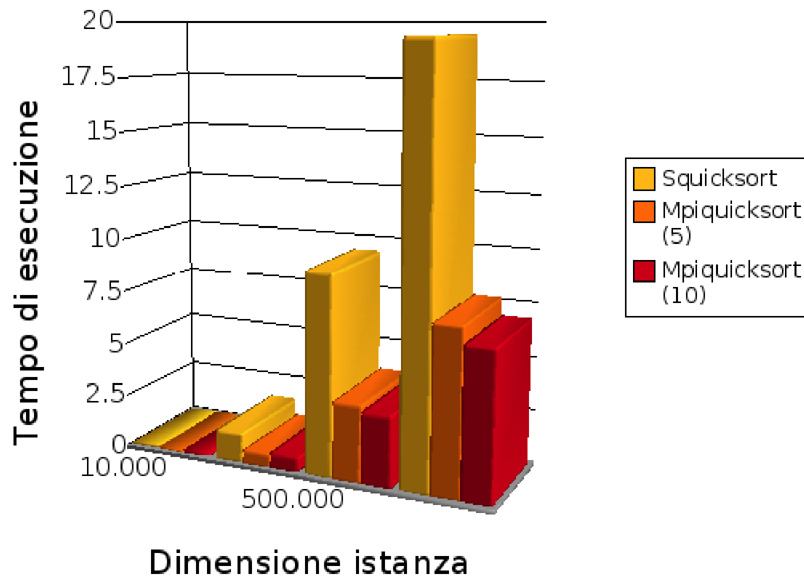

Figura 4: Tempi di esecuzione del Quicksort MPI

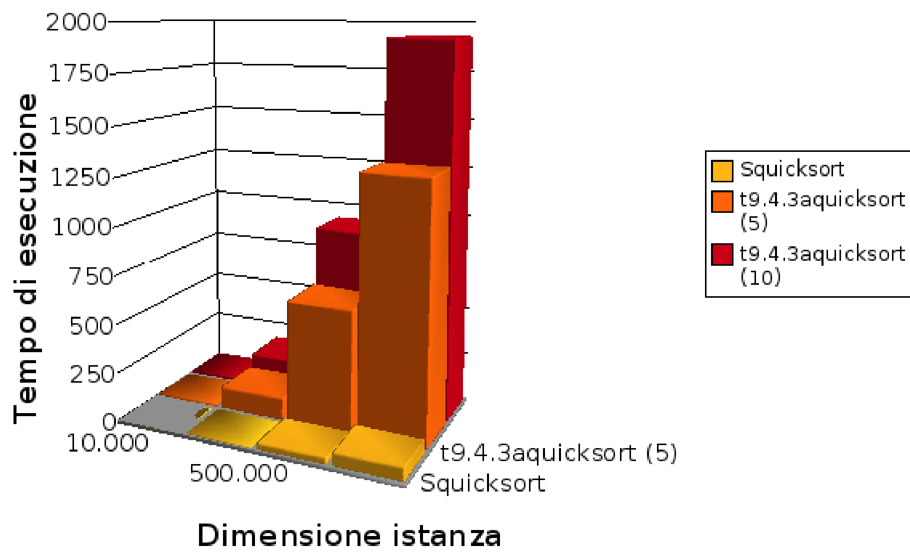

Figura 5: Tempi di esecuzione del Quicksort multithread utilizzando l'interprete Python tradizionale (CPython)

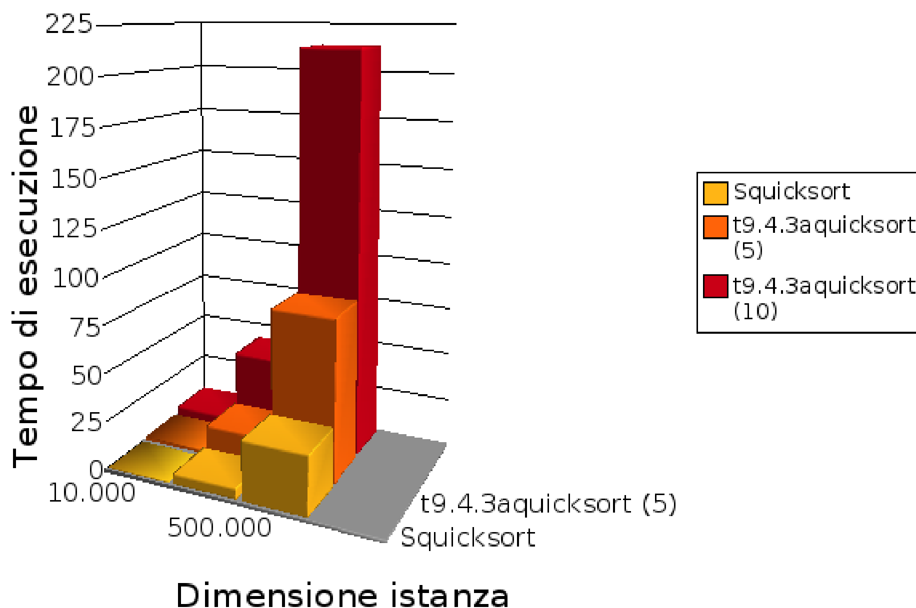

Figura 6: Tempi di esecuzione del Quicksort multithread utilizzando l'interprete Jython

4.3 Conclusioni

Le prove sperimentali sono state svolte per la soluzione MPI su un cluster formato da 7 workstation del laboratorio PLM, con le seguenti caratteristiche: IBM M-Pro, P4 3GHz, 1GB RAM, GPU GeForce Quadro FX 3000, S.O. GNU/Linux.

Mentre la soluzione multi thread è stata testata sulla macchina shared space del laboratorio PLM, con le seguenti caratteristiche: Server IBM P650, 8 processori PowerPC4+ 1400MHz, 16 GB RAM, 900 GB HD, S.O. AIX.

Per quanto riguarda la soluzione MPI le prove testimoniano come una parallelizzazione del Quicksort possa aggiungere efficacia allo sforzo computativo, garantendo i medesimi risultati *in un tempo inferiore*.

Naturalmente i risultati sperimentali variano in funzione della dimensione dell'input. Per istanze di dimensioni ridotte si fa sentire di più l'overhead dovuto allo scambio dei messaggi sulla rete, mentre per istanze sufficientemente grandi si apprezza un netto miglioramento delle prestazioni.

L'aumento di elementi di elaborazione non porta necessariamente ad un aumento delle prestazioni. Lasciando invariato il numero di processori fisici, il raddoppio degli elementi di elaborazione fa crescere di poco le prestazioni. Anche la rete può risultare essere un collo di bottiglia per via dell'aumento di messaggi legato all'aumento di elementi di elaborazione.

Per la soluzione multithread non si può arrivare alle stesse conclusioni della

soluzione MPI. Infatti per come è stato concepito l'interprete Python (un unico processo che centralizza la gestione del locking delle risorse condivise dai thread) non si riescono ad avere miglioramenti. Il maggior overhead dovuto alla creazione dei thread stessi fa peggiorare le prestazioni. Utilizzando Jython, un interprete Python scritto in Java, vi è un lieve miglioramento, ma ancora non si risparmia tempo rispetto al programma seriale. Una soluzione scritta in linguaggio C oppure Java avrebbe portato risultati diversi, questo non toglie che future versioni di Python possano portare a risolvere tale situazione.

Riferimenti bibliografici

[IPC-03] A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing* - Second Edition, Pearson - Addison Wesley

[QW-06] Quicksort (Wikipedia): http://en.wikipedia.org/wiki/Quicksort/

[MYM] Libreria Python MyMPI: http://peloton.sdsc.edu/˜tkaiser/mympi/

[SQS] Implementazione seriale del Quicksort di Aggelos Orfanakos: http://cs.uoi.gr/˜csst0266/sort.html

Powered by LaTeX!