Departament d'Enginyeria Informàtica i Matemàtiques
[DEIM] Universitat Rovira i Virgili

Arquitectura de Computadores
AC
Curso 17/18
Primera Convocatoria
Práctica 3: Comportamiento *multiprocesadores*

Procesadores MultiThread-Multicore:

Aplicación multithread

La práctica consiste en el análisis del comportamiento de una aplicación multithread cuando se ejecuta en máquinas con capacidad de ejecutar varios threads en paralelo. Concretamente se quiere estudiar la capacidad de cálculo, la productividad y la escalabilidad de un determinado algoritmo en máquinas claramente multiprocesador y en en máquinas claramente multithread.

Se dispondrá de un algoritmo secuencial al cual, mediante la librería *pthreads* de POSIX, se paralelizará creando múltiples threads que se ejecutaran en paralelo i/o concurrentemente.

La máquina multiprocesador tiene 4 procesadores unicore que se comunican entre ellos mediante link directos y acceden a memoria de 4GB mediante un FSB. En concreto se tratan de 4x Opteron 844 a 1.8Ghz.

La máquina multithread tiene 2 procesadores con 8 cores y 8 threads cada uno. Así la máquina tiene una capacidad de cálculo simultánea de 128 threads. La memoria es de 4GB. Se trata de un Sun Enterprise T5240 a 1.4 Ghz.

Comentarios

- La práctica se realizará **en GRUPOS DE 2 PERSONAS**
- Se realizará una entrevista con todos los integrantes del grupo en la sesión de laboratorio que tienen asignada.
- El informe (obligatoriamente en PDF) junto con el código implementado se comprimirán en un único archivo ZIP y se guardará en el moodle antes de realizar la entrevista.

Especificación

La tarea de esta práctica consiste en paralelizar un código secuencial mediante threads usando las llamadas de la librería pthread_create y pthread_join (ya explicadas en asignaturas como FSO).

El código proporcionado permite ser paralelizado en el primer nivel del bucle. Así no hace falta realizar ningún estudio de dependencias ya que permite dividir las diferentes iteraciones del bucle entre los threads que se quieran crear. Por ejemplo el bucle de nivel 1 (el de las i) se podría dividir entre 4 threads de la siguiente forma:

1. *for(i=0; i< N/4 ;i++)*
2. *for(i=N/4; i< N/2 ;i++)*
3. *for(i=N/2; i< 3*N/4 ;i++)*
4. *for(i=3*N/4; i< N ;i++)*

De todas formas, se debe programar el código para que admita un numero arbitrario de threads, que se especificará con un argumento de entrada.

El algoritmo se ejecutará en cada una de las dos máquinas variando el tamaño de los datos y el número de threads que lo están ejecutando. Se obtendrá el *tiempo de ejecución* de cada una de las alternativas, para luego poder hacer comparativas del tiempo de ejecución y del speedup respecto a la versión secuencial más lenta.

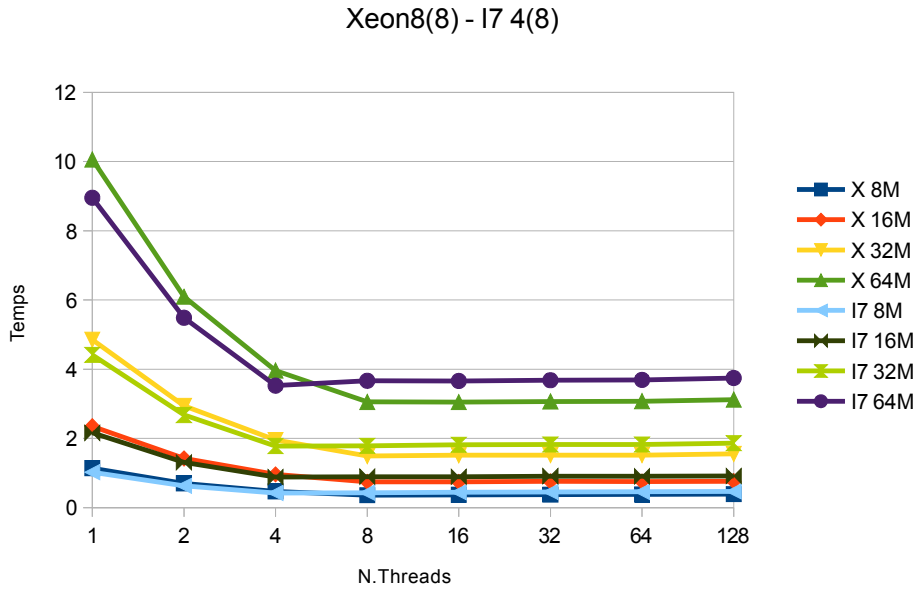
El número de threads será de 2, 4, 8, 16, 32, 64 y 128. Es de esperar que la máquina de 4 procesadores vea cierto grado de saturación a partir de 4 threads. Por otro lado la máquina de 16 cores x 8 threads es de esperar que mejore hasta los 4 threads, y siga mejorando de manera más contenida hasta los 128 threads (o inferior) y finalmente vea cierto grado de saturación a partir de ese valor.

El algoritmo también se verá incrementado por los datos que manipula y calcula. De esta forma se verá el comportamiento de la escalabilidad en función del tamaño del problema. Se consideraran varios tamaños y se incluirán en las diferentes gráficas.

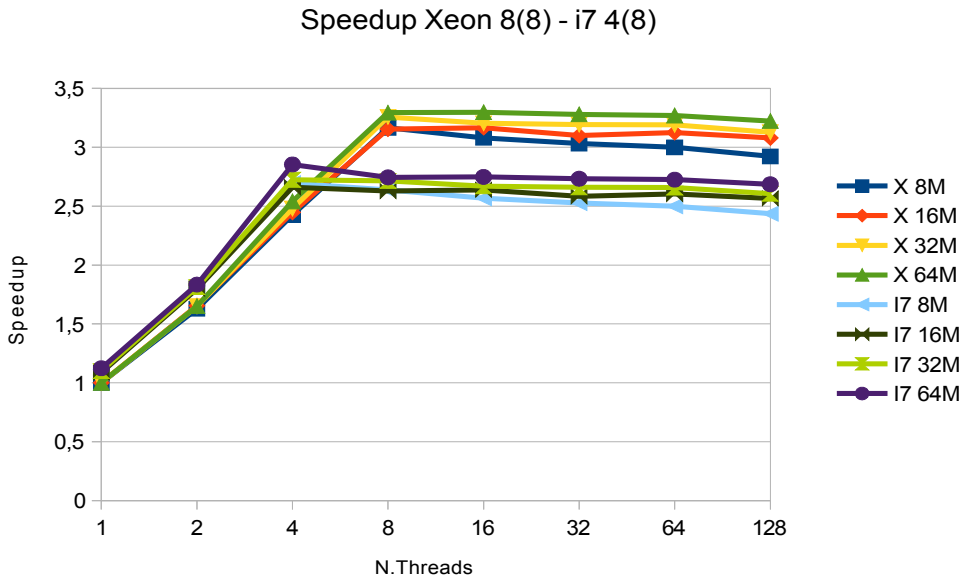
Los resultados se mostrarán en gráficas.

Para los distintos estudios que se deben realizar, y si no se indica lo contrario, tened en cuenta los siguientes comentarios:

1) Se mostrará una gráfica con el tiempo de ejecución real para cada uno de los tamaños del problema. Donde en el eje de las Y estará el tiempo de ejecución con escala lineal o logarítmica (según se aprecien mejor los valores) y en el eje de las X las diferentes ejecuciones que se han hecho variando el número de threads.



2) Del mismo modo se mostrará otra gráfica con el speedup relativo a la versión secuencial mas lenta. En este caso la versión secuencial del i7, donde en el eje de las Y estará el speedup y en el eje de las X las diferentes ejecuciones que se han hecho variando el número de threads.



3) Guía de paralelización:

1. Calculad la porción que le correspondería a cada thread (N/nTheads)
2. Cread una función que será ejecutada por cada thread de manera que a partir del valor que se le pasa sepa que porción de 'i' le corresponde.
3. Ejecutad un bucle de creación de todos los threads.
4. Ejecutad un bucle de espera de finalización de los threads.

5. El código a paralelizar es quicksort a bloques con merge en arbol dos a dos. Siendo los valores de N: 8.000.000, 16.000.000 y 32.000.000. El primer argumento del programa indica el tamaño de N a usar y el segundo el particionado del vector (el número de threads a crear) 2, 4, 8, 16, 32, 64 y 128.

6. La idea es que cada thread haga un trozo del vector con el algoritmo *quicksort* y después se sincronicen para que en un primer paso los threads pares cojan sus datos y los datos del vecino (el siguiente thread impar) y ejecute la función de *merge* obteniendo un vector el doble de grande y ordenado. Seguir con los demás threads ejecutando el *merge* de un resultado con el de al lado, hasta que se obtenga un único vector total ordenado. Este último paso lo ejecutara el thread principal. La idea es que los threads pares esperan a que finalice su pareja impar (pthread_join ↔ pthread_exit).

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NN 64000000
#define MAX_INT ((int)((unsigned int)(-1)>>1))

int valors[NN+1];
int valors2[NN+1];

void qs(int *val, int ne)
 {
  int i,f,j;
  int pivot,vtmp,vfi;

  pivot = val[0];
  i = 1;
  f = ne-1;
  vtmp = val[i];

  while (i <= f)
   {
    if (vtmp < pivot) {
         val[i-1] = vtmp;
         i ++;
         vtmp = val[i];
    }
    else {
        vfi = val[f];
         val[f] = vtmp;
         f --;
         vtmp = vfi;
    }
   }
  val[i-1] = pivot;

  if (f>1) qs(val,f);
  if (i < ne-1) qs(&val[i],ne-f-1);
 }

void merge2(int* val, int n,int *vo)
 {
  int vtmp;
  int i,j,posi,posj;

   posi = 0;
   posj = (n/2);
```

```
   for (i=0;i<n;i++)
      if (((posi < n/2) && (val[posi] <= val[posj])) || (posj >= n))
       vo[i] = val[posi++];
     else if (posj < n)
       vo[i] = val[posj++];
 }

main(int nargs,char* args[])
{
  int ndades,i,m,parts;
  int *vin,*vout,*vtmp;
  long long sum=0;

  assert(nargs == 3);

  ndades = atoi(args[1]);
  assert(ndades <= NN);

  parts = atoi(args[2]);
  if (parts < 2) assert("Han d'haver dues parts com a minim" == 0);
  if (ndades % parts) assert("N ha de ser divisible per parts" == 0);

  for(i=0;i<ndades;i++) valors[i]=rand()%MAX_INT;

// quicksort original, per fer PROVES
//  qs(valors,ndades);
//  vin=valors;


// Quicksort a parts, Cada part l'hauria de fer un thread !!
  for (i=0;i<parts;i++)
  {
  //  printf("de %d a %d\n",i*ndades/parts,(i+1)*(ndades/parts));
    qs(&valors[i*ndades/parts],ndades/parts);
  }

// Merge de dos vectors. 1 de cada 2 threads uneix els dos vectors
// A cada volta nomes treballen la meitat dels threads fins arribar
// al thread principal
  vin = valors;
  vout = valors2;
  for (m = 2*ndades/parts; m <= ndades; m *= 2)
  {
    for (i = 0; i < ndades; i += m)
     merge2(&vin[i],m,&vout[i]);
    vtmp=vin;
    vin=vout;
    vout=vtmp;
  }


/*  EXEMPLE ASSIGNACIO FEINA a 4 THREADS
  qs(&valors[0],ndades/4);          //per a TH0
  qs(&valors[ndades/4],ndades/4);   //per a TH1
  qs(&valors[ndades/2],ndades/4);   //per a TH2
  qs(&valors[3*ndades/4],ndades/4); //per a TH3

  // sincro barrera

  merge2(&valors[0],ndades/2,&valors2[0]);   //per a TH0
  merge2(&valors[ndades/2],ndades/2,&valors2[ndades/2]); //per a TH2

  /sincro barrera

  merge2(valors2,N,valors); //per a TH0
  vin=valors;
 */

  for (i=1;i<ndades;i++) assert(vin[i-1]<=vin[i]);
  for (i=0;i<ndades;i+=100)
    sum += vin[i];
  printf("validacio %lld \n",sum);
}
```