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

Arquitectura de Computadores
AC
Curso 21/22
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 multicore.

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

Las maquinas a estudiar son dos servidores con diferentes características de potencia y consumo. El primero (teen) tiene 2 procesadores Intel Xeon E5-2660 a 2.2 GHz con 8 cores y 2 threads por core cada uno, una memoria RAM de 32GB y un TDP de 95W + 95W. El segundo (orca) tiene un procesador AMD Ryzen Threadripper PRO 3995WX a 2.7Ghz con 64 cores y 2 threads por core, una memoria RAM de 128GB y un TDP de 280W.

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 video de cada miembro del grupo y 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 un bucle de N iteraciones y controlado per la variable 'num' se podría dividir entre 4 threads de la siguiente forma:

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

Aunque el ejemplo sea de 4 threads 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.

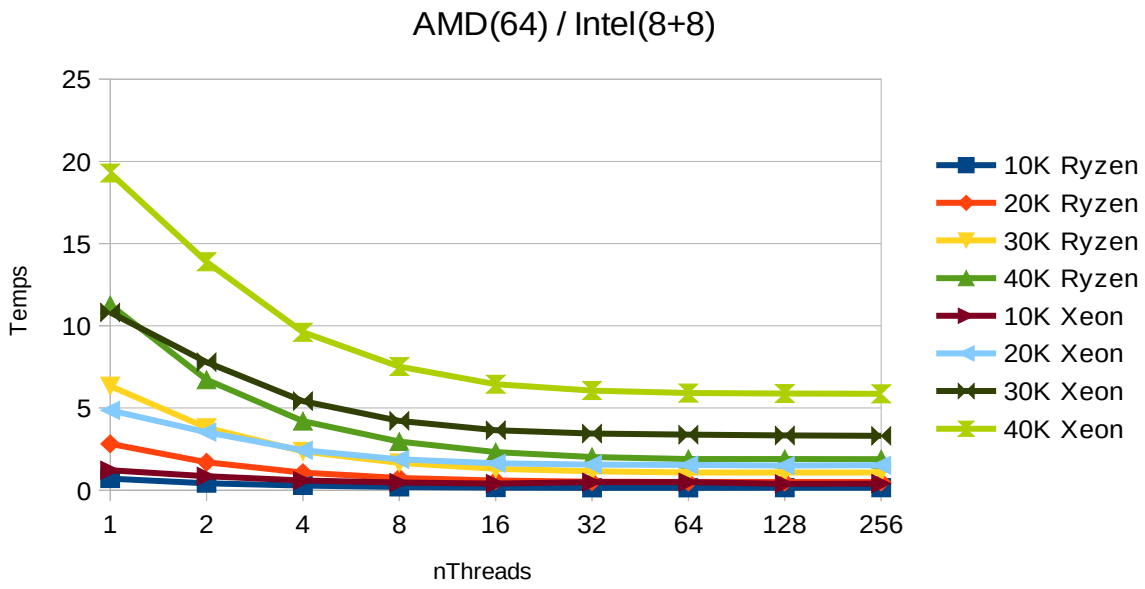
El número de threads serán de 2, 4, 8, 16, 32, 64, 128 y 256. Es de esperar que la máquina de 2 cpus x 8 cores x 2 threads vea cierto grado de saturación a partir de 16-32 threads. Por otro lado la máquina de 64 cores x 2 threads es de esperar que mejore hasta los 64 threads, y siga mejorando de manera más contenida hasta los 128 threads 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.

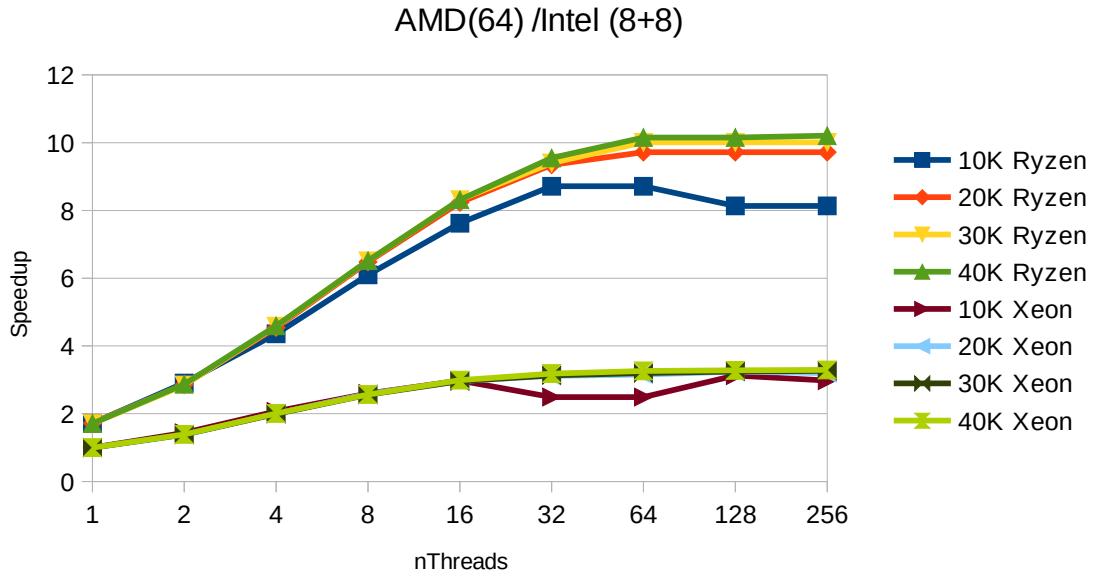
Los resultados se mostrarán en gráficas.

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 Intel, 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. Antes de paralelizar, con el código secuencial, estudiad mediante la función clock(), el tiempo de ejecución de cada bucle. Aplicad la ley de Amdahl.
2. Calculad la porción que le correspondería a cada thread del bucle/s que vayáis a paralelizar (N/nTheads).
3. Cread una función que será ejecutada por cada thread de manera que a partir del valor que se le pasa, sepa que porción de 'num' que le corresponde.
4. Ejecutad un bucle de creación de todos los threads.
5. Ejecutad un bucle de espera de finalización de todos los threads.
6. El código a paralelizar es un cálculo de regresión lineal. Siendo los valores de N (que dimensiona el problema): 15000, 25000, 35000 y 45000.
7. La idea es que cada thread haga un trozo de las iteraciones de calculo de los sumatorios de X e Y.
8. El programa tendrá dos parámetros: N y Nthreads.

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

#define N 50000

int nn;
int *X[N+1],*apX, *Y;
long long *sumaX, *sumaX2, sumaY, *sumaXY;
double *A, *B;

int main(int np, char*p[])
{
    int i,j;
    double sA,sB;
    clock_t ta,t;

    assert(np==2);

    nn = atoi(p[1]);
    assert(nn<=N);
    srand(1);

    printf("Dimensio dades =~ %g Mbytes\n",
((double)(nn*(nn+11))*4)/(1024*1024));

    //creacio matrius i vectors
    apX = calloc(nn*nn,sizeof(int)); assert (apX);
    Y = calloc(nn,sizeof(int)); assert (Y);
    sumaX = calloc(nn,sizeof(long long)); assert (sumaX);
    sumaX2 = calloc(nn,sizeof(long long)); assert (sumaX2);
    sumaXY = calloc(nn,sizeof(long long)); assert (sumaXY);
    A = calloc(nn,sizeof(double)); assert (A);
    B = calloc(nn,sizeof(double)); assert (B);

    // Inicialitzacio
    X[0] = apX;
    for (i=0;i<nn;i++) {
        for (j=0;j<nn;j+=8)
            X[i][j]=rand()%100+1;
        Y[i]=rand()%100 - 49;
     X[i+1] = X[i] + nn;
    }
```

```
    // calcul de sumatoris
    sumaY = 0;
    for (i=0;i<nn;i++) {
     sumaX[i] = sumaX2[i] = sumaXY[i] = 0;
         for (j=0;j<nn;j++) {
          sumaX[i] += X[i][j];
          sumaX2[i] += X[i][j] * X[i][j];
          sumaXY[i] += X[i][j] * Y[j];
     }
     sumaY += Y[i];
    }

    // calcul linealitat
    for (i=0;i<nn;i++) {
     B[i] = sumaXY[i] - (sumaX[i] * sumaY)/nn;
     B[i] = B[i] / (sumaX2[i] - (sumaX[i] * sumaX[i])/nn);
     A[i] = (sumaY -B[i]*sumaX[i])/nn;
    }

    // comprovacio
    sA = sB = 0;
    for (i=0;i<nn;i++) {
            //printf("%lg, %lg\n",A[i],B[i]);
         sA += A[i];
         sB += B[i];
    }

    printf("Suma elements de A: %lg B:%lg\n",sA,sB);

exit(0);
}
```