Práctica 5: Paralelización automática y mediante directivas OpenMP
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores

Jesús Alastruey Benedé y Víctor Viñals Yúfera
Área Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

28-abril-2017

Resumen

En esta práctica se analizará una solución al problema del cálculo del número π. El código que implementa el cálculo se compilará de tres formas según distintos objetivos: ejecución serie, paralelización automática por parte del compilador y paralelización mediante directivas OpenMP. También se medirán tiempos de ejecución de los distintos ejecutables para calcular aceleraciones (speedups) y rendimiento (MFLOPS).

1. Introducción
2. Análisis de las dependencias del código
3. Ejecución secuencial
4. Ejecución paralela
5. Bibliografía

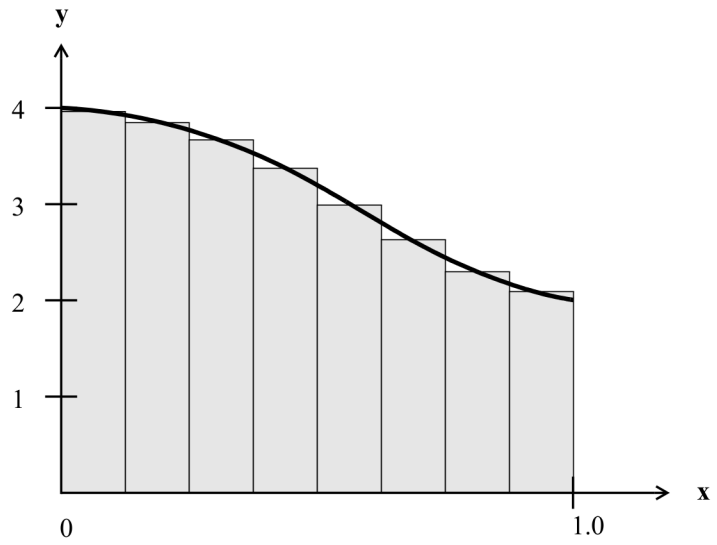

Figure 1: Aproximación para el cálculo de pi

1. Introducción

En esta práctica se va a calcular una aproximación del número π mediante la siguiente ecuación:

$\pi = \int_0^1 \frac{4}{1+x^2} dx$

Para resolver la integral se divide el dominio $[0, 1]$ en intervalos, se calcula el valor de la función $4/(1 + x^2)$ en el punto medio de cada intervalo y después se multiplican dichos valores por la anchura de cada rectángulo. Finalmente se suman las áreas de todos los rectángulos para obtener un valor aproximado de π. Por ejemplo, el área de los 8 rectángulos mostrados en la figura 1 aproximan esta integral.

Se proporciona el código Fortran que implementa este cálculo de π. En concreto, se presentan distintas versiones orientadas a la ejecución serie (con y sin desenrollado), paralela (paralelización automática por parte del compilador) y plantilla para la paralelización manual mediante directivas OpenMP. Su ubicación es:

```
hendrix://home/chus/pub_code/p5.tar.gz
```

2. Análisis de las dependencias del código

El siguiente algoritmo realiza la aproximación de π con `nsubinterval` rectángulos:

```
    ! nsubinterval = nºsubintervalos en que se divide el intervalo [0,1]
    subinterval = 1.0 / nsubintervals
    area = 0.0
    DO i = 1,nsubintervals
S1      x = (i-0.5)*subinterval
S2      area = area + 4.0/(1.0 + x*x)
    ENDDO
```

Analiza las dependencias de este código.

3. Ejecución secuencial

3.1. Experimento 1

```
    PROGRAM PI_SERIE
        IMPLICIT NONE
        real(8), PARAMETER:: nsubintervals=100000000
        real(8) x, pi, area, subinterval
        integer i
        integer count1, count2, cr
        real etime, dtime
        real t1, t2, tarray1(2), tarray2(2)

        call system_clock(count_rate = cr)

        print *," "
        print *,"Nº de procesadores: 1"


        print *,"Nº de threads: 1"
        print *,"Resolución de los relojes:"
        print *,"- system_clock:",1e6/cr,"usg->",cr/1e6," MHz"
        print *," "

        call system_clock(count=count1) ! start timing
        t2 = etime(tarray2)
        t1 = dtime(tarray1)
```

```
    subinterval = 1.0 / nsubintervals;
    area = 0.0;

    DO i = 1,nsubintervals
        x = (i-0.5)*subinterval;
        area = area + 4.0/(1.0 + x*x);
    ENDDO

    pi = subinterval*area;

    t1 = dtime(tarray1)
    t2 = etime(tarray2)
    call system_clock(count=count2) ! stop timing

    print *,"***************************************"
    print *,"  PIcalculado =", pi
    print *,"  PIreal      = 3.1415926535897932385"
    print *,"***************************************"
    print *, " "

    print *,"*** tiempos ***"
    print *,"system_clock =", (count2-count1)/1e6," sg"
    print *,"dtime =", t1," sg"
    print *,"etime =", t2," sg"
    print *,"-----------------------------------------"
END
```

Compila el código `pi_serie.f90` en `hendrix`:

```
$ f90 -g -xO3 pi_serie.f90 -o pi_serie
```

Las opciones de compilación se describen en la documentación de Sun Studio Collection:

http://docs.oracle.com/cd/E19205-01/index.html

A falta de informe de compilación, observa las anotaciones del código fuente generadas por el compilador:

```
$ er_src pi_serie
```

Ejecuta el binario generado [1]. Anota su tiempo de ejecución y calcula los MFLOPS alcanzados.

3.2. Experimento 2

El desenrollado del bucle (grado 4) puede mejorar el rendimiento del código.

```
! sólo se muestran los cambios respecto al código anterior
PROGRAM PI_G4
    real(8) varea(4)

    varea(1) = 0.0
    varea(2) = 0.0
    varea(3) = 0.0
    varea(4) = 0.0

    DO i = 1,nsubintervals,4
        x = (i-0.5)*subinterval
        varea(1) = varea(1) + 4.0/(1.0 + x*x);
```

[1] Para análisis más detallados puedes redirigir la salida estándar a un fichero.

```
        x = (i+0.5)*subinterval
        varea(2) = varea(2) + 4.0/(1.0 + x*x);

        x = (i+1.5)*subinterval
        varea(3) = varea(3) + 4.0/(1.0 + x*x);

        x = (i+2.5)*subinterval
        varea(4) = varea(4) + 4.0/(1.0 + x*x);
    ENDDO
```

Compila el programa `pi_g4.f90` con las mismas opciones que el programa anterior y observa las anotaciones del código fuente generadas.

Ejecuta el binario generado. A partir del tiempo de ejecución, calcula el rendimiento en MFLOPS y la mejora (*speedup*) respecto el código del experimento 1.

3.3. Experimento 3

El algoritmo puede ejecutarse efectuando menos operaciones de coma flotante. Para ello, aplicamos una optimización manual: el valor de `x` al comienzo de cada iteración se calcula a partir del su valor en la iteración anterior. El código resultante del bucle se muestra a continuación:

```
x = 0.5*subinterval
DO i = 1,nsubintervals,4
    varea(1) = varea(1) + 4.0/(1.0 + x*x)
    x = x + subinterval

    varea(2) = varea(2) + 4.0/(1.0 + x*x)
    x = x + subinterval

    varea(3) = varea(3) + 4.0/(1.0 + x*x)
    x = x + subinterval

    varea(4) = varea(4) + 4.0/(1.0 + x*x)
    x = x + subinterval
ENDDO
```

Compila este programa (`pi_g4p.f90`) y analiza las anotaciones del compilador.

Ejecuta esta nueva versión del programa. Calcula el rendimiento en MFLOPS y la mejora (*speedup*) respecto el código del experimento 1.

4. Ejecución paralela

4.1. Paralelización automática por parte del compilador

Compila el código `pi.f90` con las opciones necesarias para que el bucle de cálculo se paralelice de forma automática y la salida del compilador muestre los bucles paralelizados.

Tendrás que buscar dichas opciones en la documentación de Sun Studio Collection 12:

http://docs.oracle.com/cd/E19205-01/index.html

Analiza la salida del compilador. Observa más detalles del proceso de compilación mediante el análisis de las anotaciones del compilador en el código fuente.

Ejecuta el programa utilizando 1, 2, 4, 8, 16 y 32 threads. Recuerda que la variable de entorno `OMP_NUM_THREADS` controla el número de threads que van a ejecutar un programa compilado para ejecución paralela. Su valor por defecto en hendrix es 1. Para más información sobre ejecución paralela, consulta el capítulo “Parallelization” del documento “Sun Studio 12: Fortran Programming Guide”:

http://docs.oracle.com/cd/E19205-01/819-5262/aeuie/index.html

Al respecto de los tiempos de ejecución observados:

- ¿Cuál es la diferencia entre los tiempos que devuelven las funciones `dtime()` y `etime()`?
- Compara el tiempo de ejecución de 1 thread con el obtenido por la versión secuencial (experimento 1).
- Calcula el rendimiento en MFLOPS alcanzado en cada ejecución.
- Calcula las aceleraciones (*speedups*) respecto a la ejecución de este código con 1 procesador.
- Trata de relacionar los *speedups* con las características de `hendrix`.

```
PROGRAM PI_F90
      IMPLICIT NONE
      real(8), PARAMETER:: nsubintervals=100000000
      real(8) x, pi, area, subinterval
      integer i, count1, count2, cr
      integer num_procs, num_threads    ! funciones
      integer nprocs, nthreads      ! variables
      real etime, dtime
      real t1, t2, tarray1(2), tarray2(2)

      ! nprocs = num_procs()
      ! nthreads = num_threads()
      call system_clock(count_rate = cr)

      print *," "
      ! print *,"Nº de procesadores"
      ! print *,"- num_procs: ",nprocs
      ! print *,"Nº de threads"
      ! print *,"- num_threads:", nthreads
      print *,"Resolución de los relojes:"
      print *,"- system_clock:",1e6/cr,"usg->",cr/1e6," MHz"
      print *," "

      call system_clock(count = count1) ! start timing
      t2 = etime(tarray2)
      t1 = dtime(tarray1)

      subinterval = 1.0 / nsubintervals;
      area = 0.0;

      DO i = 1,nsubintervals
          x = (i-0.5)*subinterval;
          area = area + 4.0/(1.0 + x*x);
      ENDDO

      pi = subinterval*area;

      ! call sleep(10)

      t1 = dtime(tarray1)
      t2 = etime(tarray2)
      call system_clock(count=count2) ! stop timing

      print *,"************************************"
      print *,"  PIcalculado =", pi
      print *,"  PIreal      = 3.1415926535897932385"
      print *,"************************************"
      print *, " "
```

```
      ! print *,"- Threads utilizados (num_threads)", nthreads
      ! print *, " "

      print *,"*** tiempos ***"
      print *,"system_clock =", (count2-count1)/1e6," sg"
      print *,"dtime =", t1," sg"
      print *,"etime =", t2," sg"
      print *,"-----------------------------------------------"
END
```

4.2. Paralelización manual mediante directivas OpenMP

En este último experimento debes especificar de forma manual el paralelismo existente en el bucle principal del programa. Para ello se utilizarán directivas OpenMP.

```
PROGRAM PI_OMP
      IMPLICIT NONE
      real(8), PARAMETER:: nsubintervals=100000000
      real(8) x, pi, area, subinterval
      integer i, count1, count2, cr
      !! real(8) wtime1, wtime2
      real(8) wtr, tth1, tth2
      real(8) omp_get_wtick, omp_get_wtime             ! funciones
      integer omp_get_max_threads, omp_get_num_procs   ! funciones
      integer omp_get_num_threads, omp_get_thread_num  ! funciones
      integer omp_set_dynamic, omp_get_dynamic         ! funciones
      integer nprocsOMP, nthreadsOMP        ! variables
      integer maxnthreadsOMP, threadID      ! variables
      real etime, dtime                     ! funciones
      real t1, t2, tarray1(2), tarray2(2)   ! variables

      maxnthreadsOMP = omp_get_max_threads()
      nprocsOMP = omp_get_num_procs()
      call system_clock(count_rate = cr)
      wtr = omp_get_wtick() ! resolucion omp_get_wtime() en segundos

      !! CALL OMP_SET_DYNAMIC(.TRUE.)

      print *," "
      print *,"Nº de procesadores"
      print *,"- omp_get_num_procs: ",nprocsOMP
      print *,"Nº de threads"
      print *,"- omp_get_max_threads: ",maxnthreadsOMP
      print *,"Resolución de los relojes:"
      print *,"- system_clock:", 1e6/cr," usg ->", cr/1e6," MHz"
      print *,"- omp_get_wtick:",1e6*wtr,"usg->",1.0/(1e6*wtr)," MHz"
      print *," "

      print *, "Calculando pi (nº intervalos: ", nsubintervals,")"
      print *, " "

      call system_clock(count=count1)   ! start timing
      t2 = etime(tarray2)
      t1 = dtime(tarray1)

      ! omp_get_wtime() returns a double-precision fp value
      ! equal to the elapsed wall clock time in seconds
```

```
      ! since some time in the past
      ! wtime1 = omp_get_wtime()


      subinterval = 1.0 / nsubintervals;
      area = 0.0;

      nthreadsOMP = omp_get_num_threads()
      threadID = omp_get_thread_num()

      tth1 = omp_get_wtime()

      DO i = 1,nsubintervals
         ! threadID = omp_get_thread_num()
         ! print *, "Thread ",threadID, " : iteración", i
         x = (i-0.5)*subinterval;
         area = area + 4.0/(1.0 + x*x);
      ENDDO

      tth2 = omp_get_wtime()
      print *,"omp_get_wtime thread",threadID,":",tth2-tth1," sg"

      ! call sleep(10)
      ! wtime2 = omp_get_wtime()
      t1 = dtime(tarray1)
      t2 = etime(tarray2)
      call system_clock(count=count2) ! stop timing

      pi = subinterval*area;

      print *,"****************************************"
      print *,"  PIcalculado =", pi
      print *,"  PIreal      = 3.1415926535897932385"
      print *,"****************************************"
      print *, " "

      print *,"- threads utilizados:", nthreadsOMP
      print *, " "

      print *,"*** tiempos ***"
      ! print *,"omp_get_wtime = ", wtime2-wtime1, " sg"
      print *,"system_clock =", (count2-count1)/1e6," sg"
      print *,"dtime =", t1," sg"
      print *,"etime =", t2," sg"
      print *,"-----------------------------------------------"
END
```

Inserta las directivas OpenMP correspondientes en el fuente `pi_omp.f90` (en torno al bucle principal).

Compila el código, y haz un breve análisis de la salida del compilador y de sus anotaciones en el código fuente.

Ejecuta el programa utilizando 1, 2, 4, 8, 16 y 32 threads. A partir de los tiempos de ejecución, calcula el rendimiento en MFLOPS alcanzado en cada ejecución y las aceleraciones (*speedups*) respecto a la ejecución de este código con 1 procesador.

5. Planificación de iteraciones

La planificación de iteraciones puede controlarse mediante la orden `SCHEDULE` que se utiliza junto con la directiva `OMP_DO`. Ejecuta el programa anterior con las siguientes estrategias de asignación de iteraciones a threads (sólo para 4 threads):

1. `static`.
2. `static`, con valores de *chunk* 1, 100 y 10000.
3. `dynamic`, con valores de *chunk* 1, 100 y 10000.
4. `guided`, con valores de *chunk* 1, 100 y 10000.

Compara los tiempos de ejecución obtenidos con las distintas estrategias de planificación y saca conclusiones de dicha comparación.

(Optativo) Cálculo de aceleraciones en otros sistemas

Si tienes un procesador con varios núcleos (Dual-Core, Quad-Core, Hexa-Core ... de Intel o AMD), puedes repetir la práctica con dicho sistema. Para ello necesitarás un compilador que soporte las directivas OpenMP (`gcc`, `icc`).

Bibliografía

- Sun Studio 12 Collection: compiladores C, C++, Fortran y herramientas de desarrollo, depuración y optimización. Su documentación puede consultarse en el siguiente enlace:

  http://docs.oracle.com/cd/E19205-01/index.html