UNIVERSIDAD DE GRANADA

Práctica 2: Cálculo de π

Arquitectura y Computación de altas prestaciones

Luis González Romero

23 de marzo de 2019

Índice

Índice de figuras

1. Enunciado

- Desarrollar un programa MPI del timpo SPMD MasterSlave que calcule π por integración numérica de la derivada $\arctan x$ en el intervalo [0,1].
- Compilarlo y ejecutarlo en el Aula de prácticas con 1, 2, 3, y 4 unidades de ejecución, comprobando la evolución del error de integración y el tiempo de ejecución.
- Representar gráficamente el tiempo de ejecución total y la ganancia en velocidad en función del número de PCs.

2. Cálculo de π

La derivada de la función $\arctan x$ es $1/(1 + x * x)$

$$\int_0^1 \frac{1}{1 + x^2} = \arctan \Big|_0^1 = \frac{\pi}{4} - 0$$

Figura 1: Cálculo integral

Por lo que si calculamos la integral y la multiplicamos por 4, obtendremos el valor de π

3. Selección método de aproximación

Para calcular la integral hacemos una aproximación calculando el área de los rectángulos, utilizando un solo punto de la función $1/(1+x^2)$

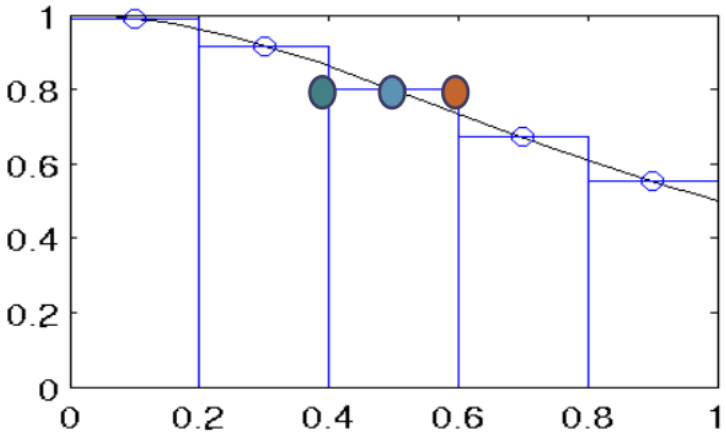
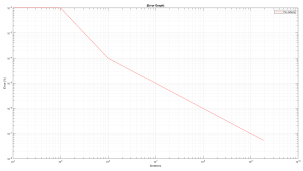


Figura 2: Métodos de aproximación

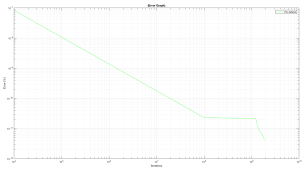
Podemos tomar distintos puntos con los que tendremos errores distintos:

- El punto izquierdo, $x = (i - 0{,}5) * width$; (*cpi-def.c*)
- El punto medio, $x = (i + 0{,}5) * width$; (*cpi-med.c*)
- El punto derecho, $x = (i + 1) * width$; (*cpi-exc.c*)

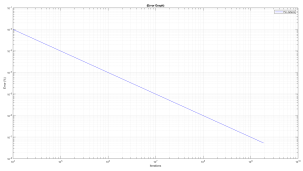
Cambiando el código para tomar estos tres distintos casos obtenemos estos errores:



Error por defecto



Error medio



Error por exceso

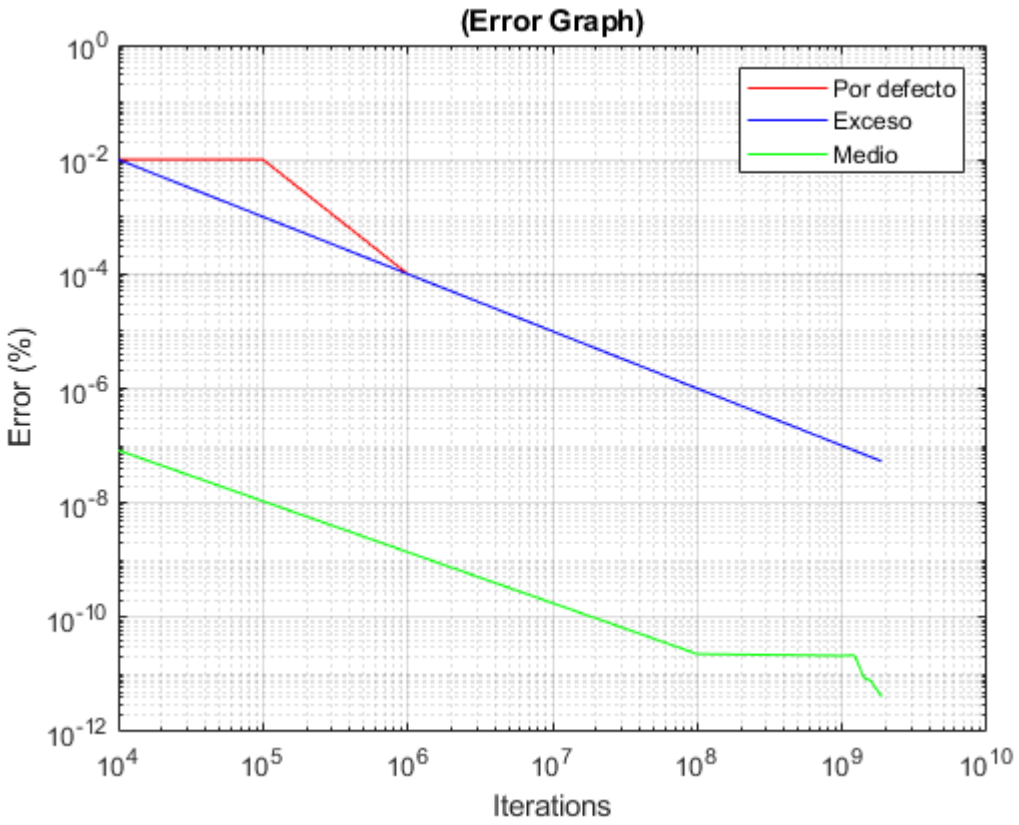


Figura 3: Errores obtenidos para los distintos casos

Como se puede observar el punto medio es el que menor error comete, los otros dos quedan prácticamente solapados.

4. Solución con MPI

Partiendo del siguiente código secuencial:

```
/* get the number of intervals */
intervals = atoi(argv[1]);
width = 1.0 / intervals;

/* do the computation */
sum = 0;
for (i=0; i<intervals; ++i) {
  register double x = (i + 0.5) * width;
  sum += 4.0 / (1.0 + x * x);
}
sum *= width;
```

Figura 4: Código secuencial del cálculo de π

mediante una **Reducción** reunimos los resultados de cada *rank*.

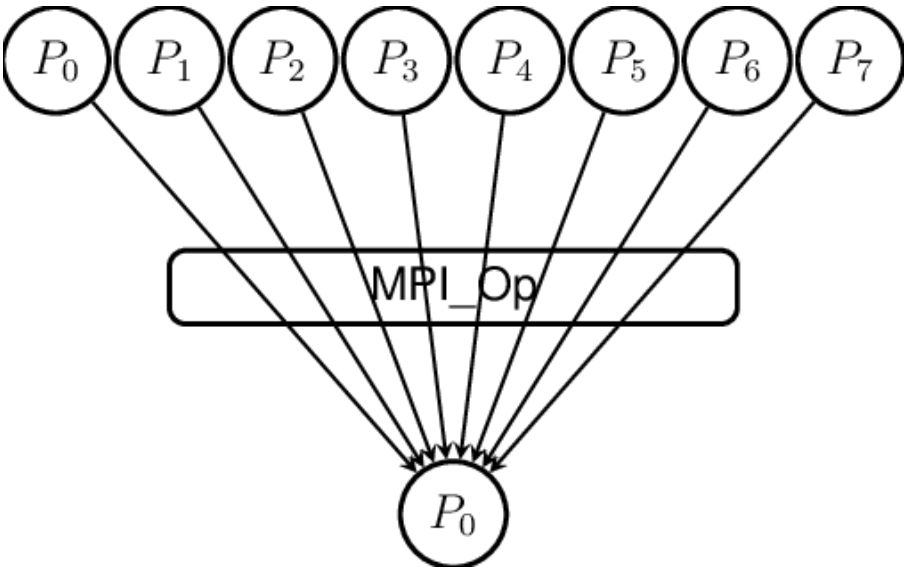


Figura 5: Esquema MPI_Reduce

Como MPI_Op, usamos MPI_SUM para recoger las soluciones parciales.

```
#define PI 3.14159265358979323846264 3

main(int argc, char **argv)
{
  register double width, x, sum;
  register int intervals, i;
  double global_sum, local_sum;
  int np, total_np;
  MPI_Status status;

  intervals = atoi(argv[1]);
  double start_time = omp_get_wtime();
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &total_np);
  MPI_Comm_rank(MPI_COMM_WORLD, &np);

  width = 1.0 / intervals;
  local_sum = 0;

  for (i=np; i<intervals; i+=total_np) {
    x = (i + 0.5) * width;
    local_sum += 4.0 / (1.0 + x * x);
  }
  local_sum *= width;

  MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Finalize();

  double parallel_time = omp_get_wtime() - start_time;
  double pi_error = (global_sum-PI)*100;
```

Figura 6: Código paralelo MPI del cálculo de π

5. Tiempo de ejecución y ganancia

Para obtener los tiempos de ejecución se han hecho distintas mediciones y se ha obtenido: la media de cada caso de las configuraciones y la dispersión estándar de estas.

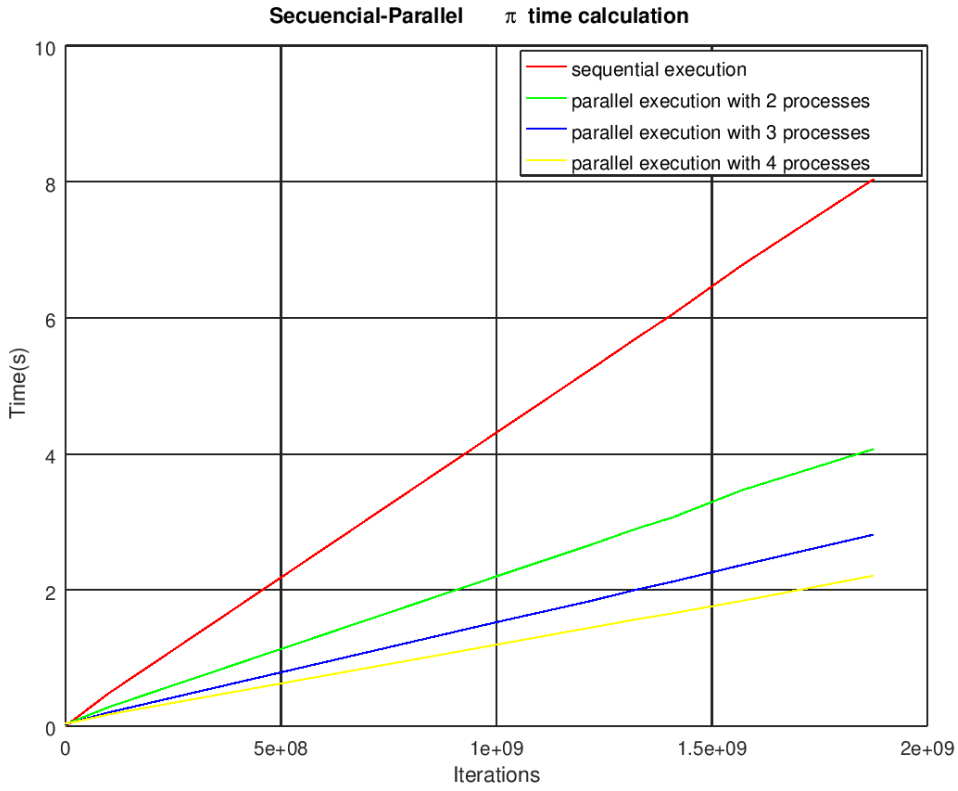


Figura 7: Tiempos de ejecución de todas las configuraciones

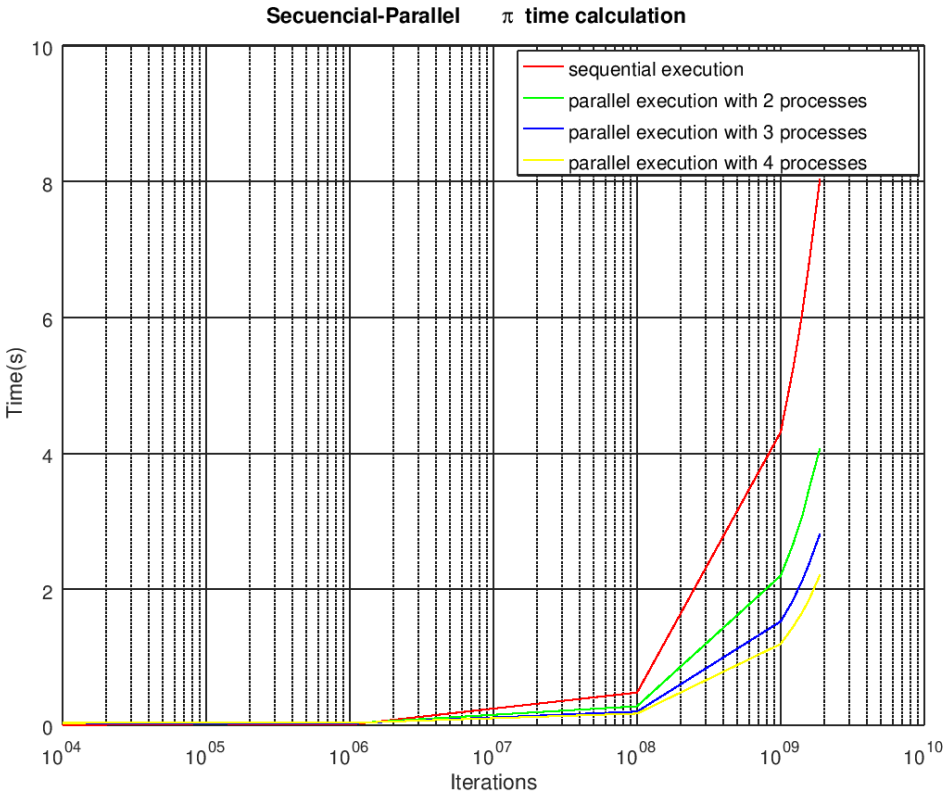


Figura 8: Tiempo de ejecución de todas las configuraciones con escala logarítmica en el eje x

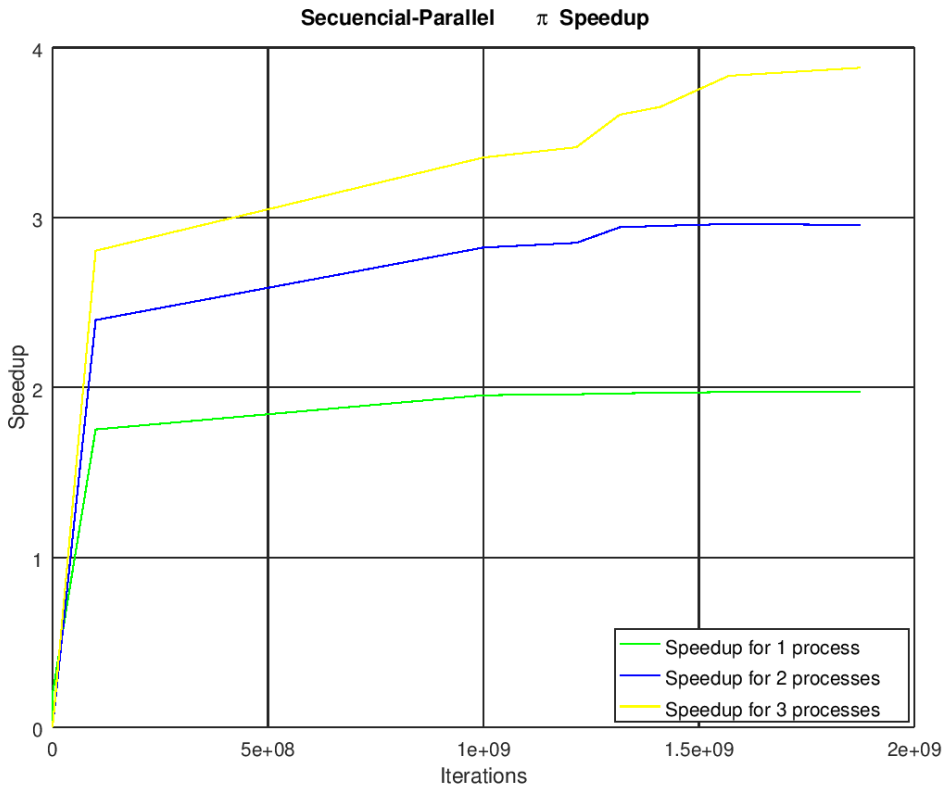


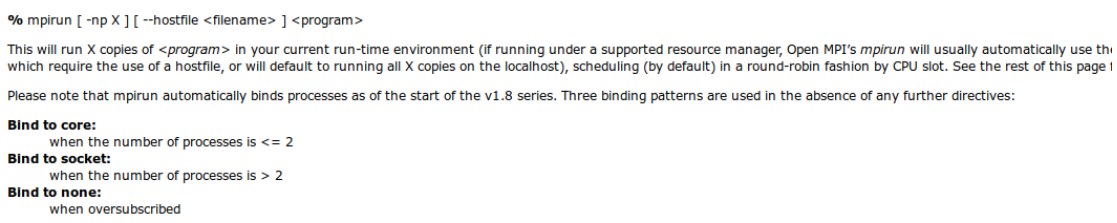
Figura 9: Ganancia para todas las configuraciones

La ganancia como podemos apreciar siempre tiende al número de procesos usados.

Todos los tiempos obtenidos junto a la media y desviación de estos, están en el fichero *calculos-estadisticos.txt*

6. Conclusión

Como se ha podido observar con las gráficas la ganancia tiende al número de procesos especificado al ejecutar el programa con *mpirun*.



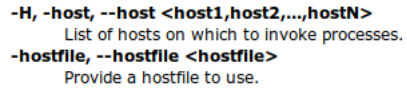
% mpirun [-np X] [--hostfile <filename>] <program>

This will run X copies of *<program>* in your current run-time environment (if running under a supported resource manager, Open MPI's *mpirun* will usually automatically use th
which require the use of a hostfile, or will default to running all X copies on the localhost), scheduling (by default) in a round-robin fashion by CPU slot. See the rest of this page

Please note that mpirun automatically binds processes as of the start of the v1.8 series. Three binding patterns are used in the absence of any further directives:

Bind to core:
when the number of processes is <= 2
Bind to socket:
when the number of processes is > 2
Bind to none:
when oversubscribed

Figura 10: *mpirun* man



-H, -host, --host <host1,host2,...,hostN>
List of hosts on which to invoke processes.
-hostfile, --hostfile <hostfile>
Provide a hostfile to use.

Figura 11: -*hostfile* option

Cuando usamos en el aula un *hostfile* con los nodos a los que nos conectamos(usando en mi caso los *ei142XXX* que están conectados en estrella para minimizar el camino) *OpenMPI* ejecuta una copia del programa en cada uno.