Aceleración de la técnica de Monte Carlo para integración numérica usando memoria compartida y memoria distribuida

Edwin Urbina Quiroz
Ervin Villalta Cáceres
Isao Núñez Okubo

Universidad de Costa Rica

December 4, 2025

Contenido

Motivación del proyecto

- La integración numérica tradicional se vuelve muy costosa en altas dimensiones.
- La técnica de Monte Carlo reduce este costo usando muestreo aleatorio.
- Objetivo del proyecto:
  - Implementar la integración de Monte Carlo multidimensional.
  - Acelerar el método usando memoria compartida y memoria distribuida.
  - Estudiar la escalabilidad del método.

Idea general del método

- Se desea calcular una integral multidimensional:

$$I = \int_{[a,b]^d} f(\mathbf{x})\, d\mathbf{x}.$$

- Se reescribe como valor esperado:

$$I = V\, \mathbb{E}[f(\mathbf{X})], \quad V = (b-a)^d.$$

- Aproximación por Monte Carlo:

$$\hat{I}_N = V\, \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{X}_i).$$

- Ventaja: el error decrece como $N^{-1/2}$ y no explota con la dimensión.

Función objetivo utilizada

- Se toma como función de prueba:

$$f(\mathbf{x}) = \exp\left(-\sum_{i=1}^{d} x_i^2\right).$$

- Aparición en:
  - física estadística,
  - probabilidad multivariada,
  - integrales gaussianas en altas dimensiones.
- Es una función suave y bien comportada para estudiar convergencia y error.

Generación de puntos aleatorios

- Se usa el generador `std::mt19937` (Mersenne Twister).
- Se genera un uniforme en $[0, 1]$:

$$r = \frac{\texttt{generador()}}{\texttt{generador.max()}}.$$

- Se escala al intervalo $[a, b]$:

$$x_i = a + (b - a)\, r.$$

- El proceso se repite para cada dimensión y para cada punto.

Código principal del método

Bucle de Monte Carlo

```
for (int i = 0; i < N; i++) {
  std::vector<double> punto(dimensiones);
  for (int d = 0; d < dimensiones; d++) {
    double r = double(generador()) / generador.max();
    punto[d] = lim_inf + (lim_sup - lim_inf)*r;
  }
  double valor = func(punto);
  suma_final += valor;
  suma_final2 += valor * valor;
}
```

Cálculo de la integral y del error

- Promedio:

$$\hat{f} = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_i).$$

- Varianza:

$$\hat{\sigma}^2 \approx \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_i)^2 - \hat{f}^2.$$

- Integral estimada:

$$I \approx V \hat{f}.$$

- Error estadístico:

$$\Delta I = V \sqrt{\frac{\hat{\sigma}^2}{N}}.$$

Cálculos

```
// Cálculos finales
double promedio = suma_final / N;
double promedio_cuadrado = suma_final2 / N;
double varianza = promedio_cuadrado - promedio * promedio;

// Volumen
double volumen = 1.0;
for (int d = 0; d < dimensiones; d++) {
  volumen *= (lim_sup - lim_inf);
}

// Estimación de la integral
double integral = promedio * volumen;

// Estimación del error
double error = volumen * std::sqrt(varianza / N);
```

Versión con argumentos de línea de comandos

- El programa permite elegir parámetros desde la terminal:

```
./mc --li 0 --ls 1 --d 3 --n 5000000
```

- Parámetros:
  - `--li`: límite inferior.
  - `--ls`: límite superior.
  - `--d`: número de dimensiones.
  - `--n`: número de puntos N.
- Esta versión facilita:
  - barrer distintos valores de N,
  - cambiar dimensión d,
  - automatizar experimentos para medir tiempos y error.

Código Memoria Compartida

```
double suma_final = 0.0;
double suma_final2 = 0.0;
double time_1 = omp_get_wtime();
int num_procs;
#pragma omp parallel
{
```

Memoria Compartida

```
int num_procs = omp_get_thread_num();
std::mt19937 generador(seed + num_procs * 7919);
std::uniform_real_distribution<double> dist(lim_inf, lim_sup);
std::vector<double> punto(dimensiones);
#pragma omp single
num_procs = omp_get_num_threads();
#pragma omp for reduction (+:  suma_final, suma_final2)
or (int i = 0; i < N; i++) {
  for (int d = 0; d < dimensiones; d++) {
    punto[d] = dist(generador);
  }
  double val = func(punto);
  suma_final += val;
  suma_final2 += val * val;
```

Escalabilidad en memoria compartida

- Se define el **speedup**:

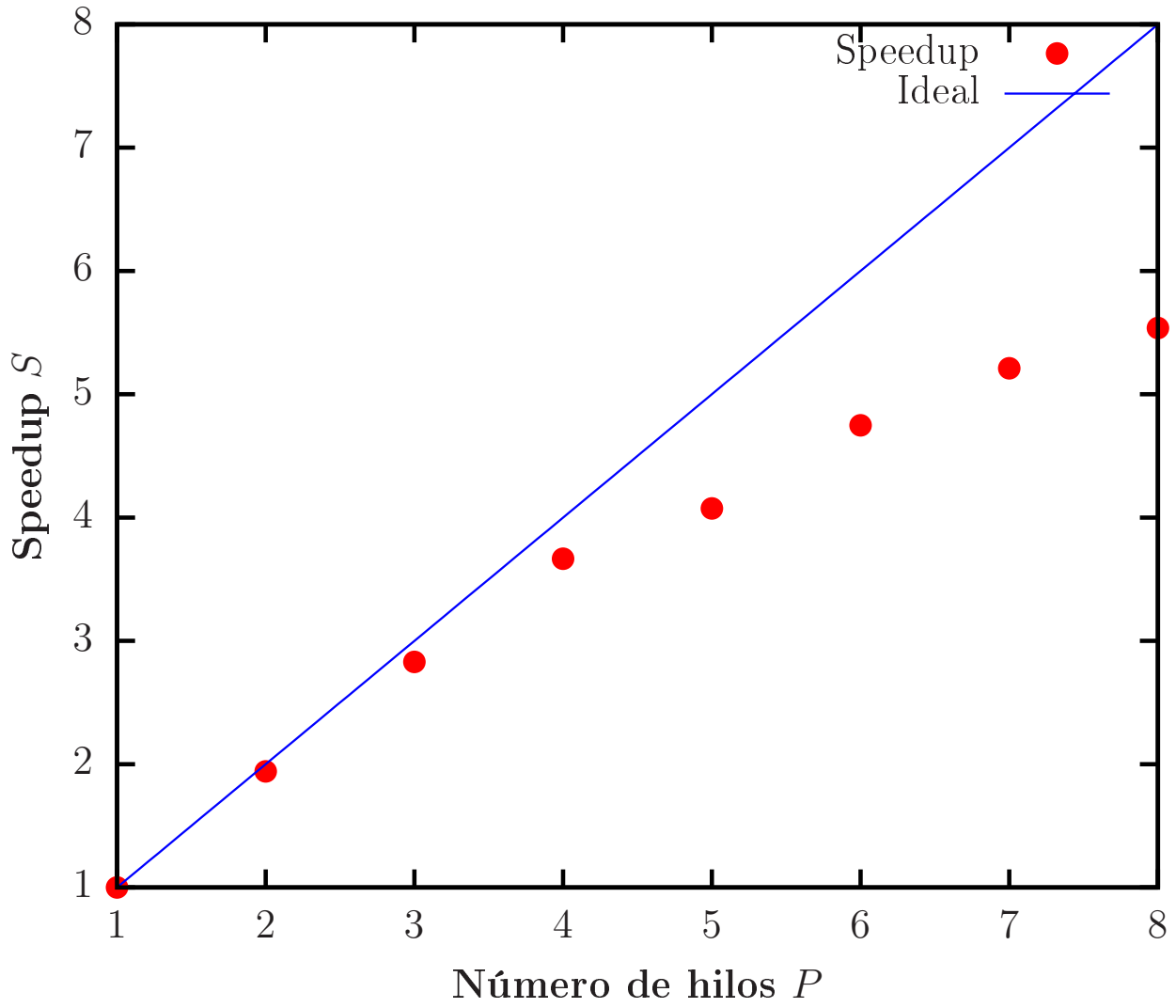
$$S(p) = \frac{T_1}{T_p},$$

  donde T_1 es el tiempo con 1 hilo y T_p con p hilos.
- La **eficiencia** se define como:

$$E(p) = \frac{S(p)}{p}.$$

- Para N grande, el método Monte Carlo escala bien con OpenMP (gran parte del tiempo está en el bucle paralelo).
- Factores que limitan la eficiencia:
  - partes secuenciales del código,
  - overhead de creación y sincronización de hilos.

Gráfico de escalabilidad (Memoria compartida)



Memoria distribuida (MPI)

- Llamada inicial: `MPI_Init(NULL, NULL);`
- Determinar número de procesos y el identificador de cada uno:
  - `MPI_Comm_size(MPI_COMM_WORLD, &size);` // número total de procesos
  - `MPI_Comm_rank(MPI_COMM_WORLD, &rank);` // id del proceso [0, size1]
- Registrar tiempo de inicio: `double time_start = MPI_Wtime();`
- Los parámetros globales (como número total de muestras N, límites del intervalo, dimensión d, semilla) se leen una sola vez — usualmente por el proceso con `rank == 0` — y pueden ser compartidos si se desea.

MPI: División del trabajo entre procesos

- El número total de muestras N se reparte entre los procesos:

$$N_r = \left\lfloor \frac{N}{\text{size}} \right\rfloor$$

  donde `size` es el total de procesos y `rank` el identificador del proceso.
- Cada proceso calcula su número local de puntos `N_local`:

```
N_local = N / size
```

- Si N no es múltiplo de size, el proceso 0 puede procesar los puntos sobrantes:

$$N_0 = N_r + (N \bmod \text{size})$$

- Con esta estrategia:
  - cada proceso trabaja de forma independiente,
  - se minimiza comunicación durante el cálculo,
  - la carga computacional queda balanceada.

MPI: Process Safety

- Cada proceso genera sus propios puntos aleatorios dentro del intervalo $[a, b]$.
- Se usa una semilla distinta para cada proceso:

  `std::mt19937 gen(seed + rank ·1234567);std::mt19937 gen(see`

- Cada proceso acumula:

$$S_r = \sum f(\mathbf{x}_i), \qquad Q_r = \sum f(\mathbf{x}_i)^2$$

  donde S_r y Q_r son las sumas locales.
- Esta etapa es completamente independiente $\rightarrow$ **no requiere comunicación**, lo que favorece la escalabilidad.

Memoria distribuida (MPI)

- En resumen, la versión con MPI, el trabajo se reparte entre varios procesos con memoria independiente.
- Cada proceso calcula una fracción de los N puntos:

$$N = \sum_{r=0}^{p-1} N_r, \quad N_r \approx \frac{N}{p}.$$

- Se usan semillas distintas por proceso:

```
std::mt19937 gen(seed + rank * 1234567);
```

- Cada proceso obtiene sus sumas locales `suma_local`, `suma_cuadrados_local`.
- El proceso 0 combina los resultados con `MPI_Reduce` y calcula la integral global.

Escalabilidad en memoria distribuida

- Speedup en MPI:

$$S(p) = \frac{T_1}{T_p},$$

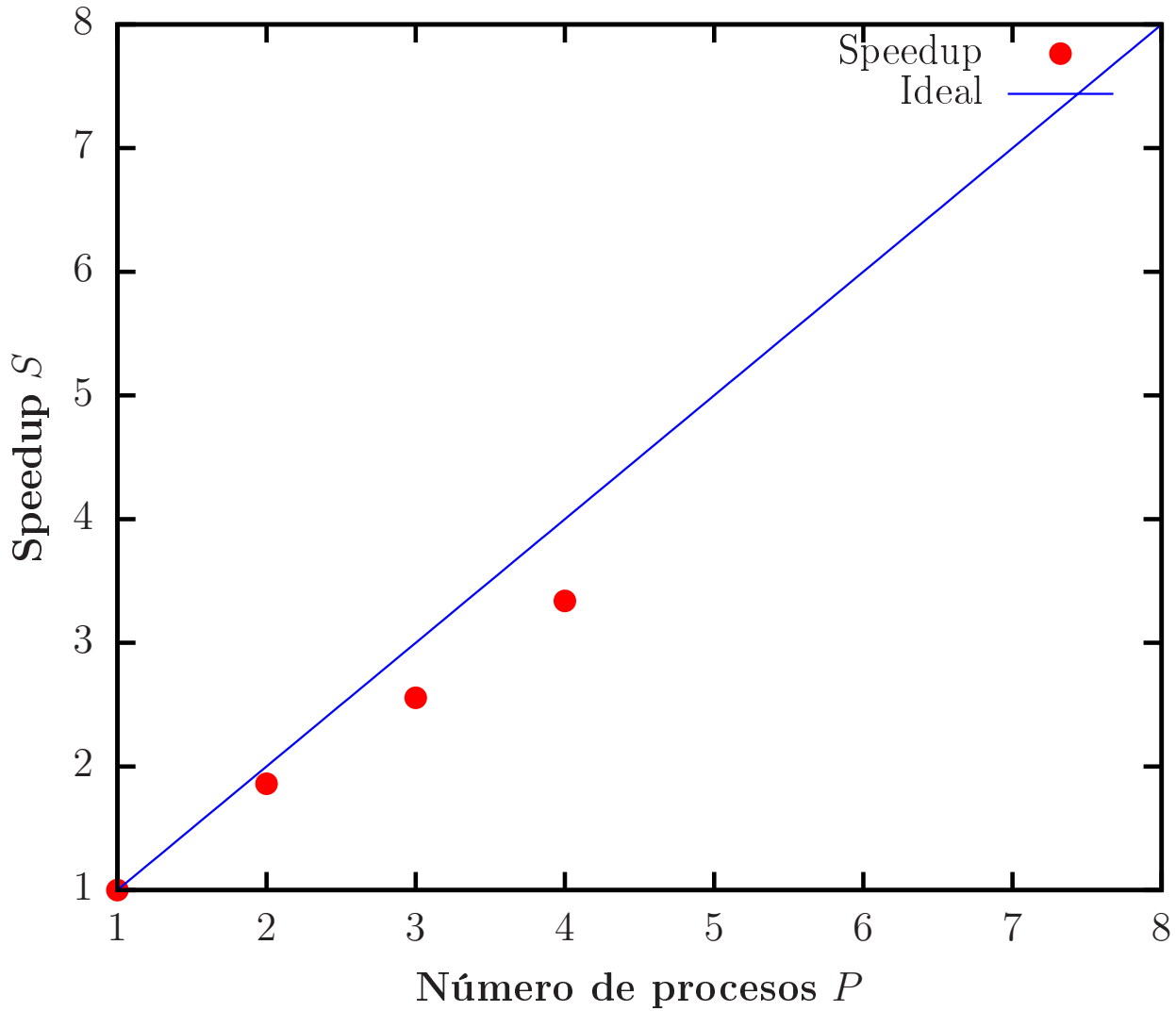
  con T_p el tiempo usando p procesos.

- Eficiencia:

$$E(p) = \frac{S(p)}{p}.$$

- Para valores grandes de N, el costo de comunicación es pequeño frente al costo de cómputo y se logra buen escalamiento.
- Para N pequeños, el overhead de comunicación y sincronización puede reducir el speedup.

Gráfico de escalabilidad (Memoria compartida)



Conclusiones del proyecto

- El método de Monte Carlo es adecuado para integrales multidimensionales debido a que el error estadístico no depende fuertemente de la dimensión.
- La implementación secuencial sirve como referencia para comparar rendimiento y precisión.
- La paralelización en memoria compartida (OpenMP) permite aprovechar eficientemente los núcleos de un solo nodo.
- La paralelización en memoria distribuida (MPI) extiende el método a clústeres y múltiples nodos.
- El análisis de escalabilidad muestra hasta dónde se puede acelerar el algoritmo antes de que dominen los costos de comunicación y sincronización.

Gracias

¡Preguntas?