Oscilador Armónico Acoplado

Jhonatan Macazana and Samir Muñoz
Departamento de Ingeniería Electrónica
Universidad de Ingeniería y Tecnología
Lima, Perú
{jhonatan.macazana & emanuel.munoz}@utec.edu.pe

***Abstract*—Los modelos armónicos representan un reto para su análisis, pero es beneficioso su solución para distintas aplicaciones. En el presente trabajo, se presenta la implementación y análisis de un sistema oscilador armónico acoplado. Se propone además una paralelización usando OpenMP utilizando convenientemente una de las soluciones conocidas matricialmente. Tambien, se específican los modelos matemáticos como también particularidades del código utilizado. Finalmente, se discuten los resultados gráficos en distintos dispositivos de tiempo y desempeño.**

***Index Terms*—oscilador, OpenMP, desempeño**

I. Introducción

El análisis de multiples objetos acoplados es recurrente para distintas aplicaciones de control o mantenimiento. Su diseño parte de un modelo matemático que puede describir su comportamiento en el tiempo. Particularmente, el sistema de osciladores armónicos oscilatorios también pueden describirse apropiadamente por su frecuencia. A partir de su dinámica dada por la ecuación clásica 1, se puede formular una solución análitica.

$$\begin{aligned} m\ddot{x}_i &= -D(x_i - x_{i+1}) - D(x_i - x_{i-1}) \\ &= -D(2x_i - x_{i+1} - x_{i-1}) \end{aligned} \tag{1}$$

De la ecuación 1, x_i representa a la desviación de la partícula i de su posición de reposo que oscila acoplada entre n partículas, D es la constante de elasticidad de todos los resortes que unen cada partícula. Se plantea que los extremos de la cadena están fijos a $x_0 = x_n =$ constante, con las relaciones representadas en el sistema de ecuaciones 2

$$\begin{aligned} \sqrt{m}x_i(t) &= v_i \exp(i\omega t) \\ y_i &= \sqrt{m_i}x_i \end{aligned} \tag{2}$$

Actualmente, se usa para el control o simulación de estos modelos computadores digitales que mediante distintas variaciones de procesos, puede incrementarse el desempeño. En este trabajo, se plantea resolver el problema utilizando paralelismo mediante OpenMP. Para ello, se arreglará el sistema de ecuaciones para su resolución matricial. Además de ello, se mostrará un análisis del desempeño del programa respecto al tamaño n del sistema y el número de hilos utilizados. Finalmente, se discuten los patrones de los resultados y su implicancia para la escalabilidad.

II. Metodología

A. Solución matemática

Un método de resolución es mediante un arreglo matricial. Considerese el sistema de ecuaciones mostrado en 2, se puede formar una matriz tridiagonal M definida como en 3, donde δ se define como $\delta_{i,j} = 1$, si i = j.

$$M_{ij} = \frac{D}{\sqrt{m_i m_j}}(2\delta_{ij} - \delta_{i,j+1} - \delta_{i,j-1}) \tag{3}$$

Esta representación de arreglo es plateada como alternativa de solución en la descripción del problema para este proyecto. En ese sentido, se conoce que el cálculo de las vibraciones del sistema se puede reducir al cálculo de los valores propios y vectores propios de la matriz tridiagonal. Se asume que dado el arreglo M, sus valores propios (λ_i) y vectores propios v_i pueden reemplazarse directamente en 2. Sea la ecuación 4 la solución del sistema a partir del uso de los valores y vectores propios de M.

$$y_i = \sum_{i=0}^{n} v_i \cos(\lambda_i t) + v_i \sin(\lambda_i t) \tag{4}$$

B. Código

1) Código general: Para la obtención de los valores y vectores propios, se usa el método Tridiagonal Quadratic Linear Implicit (tqli). El algoritmo funciona para una matriz simétrica tridiagonal, por lo que propone utilizar un método de tridiagonalización utilizando un algoritmo tred2. Sin embargo, particularmente el modelo definido M es simétrica tridiagonal. Por ello, se corrieron ambas opciones que considera el uso de tred2. Por conveniencia y seguir lo recomendado por la funcion, se utilizo tred2. Estas recomendaciones fueron extraídas del libro Numerical Recipes [1]. Como se puede apreciar, esta función no puede ser paralelizada a simple vista, por la presencia de bucles **while** y declaraciones **break**.

El codigo implementado sigue el siguiente pseudocodigo mostrado en la Figura 1.

Las variaciones realizadas y es donde se midio el desempeño del codigo es para la función tqli, que además de haber sido modificado del libro, se ha paralelizado mediante OpenMP [2].

```
m = initializeMass(N, m0, n0);
matrix = initializeMatrix(rows, cols);
diag,subdiag <- tred2(matrix,N);
aval, avect <- tqli(diag, subdiag);

for (auto t in 0:dt:T){
    X = x0 +y(aval, avect, t);
}
```

Fig. 1. Pseudocódigo del cóidigo implementado.

```
for (k = 0; k < n; k++) {
  f           = z[k][i + 1];
  z[k][i + 1] = s * z[k][i] + c * f
; z[k][i]     = c * z[k][i] - s * f;
 }
```

Fig. 2. Fragmento de código 1.

2) Código paralelo: Particularmente para la implementación en paralelo, se tuvo que modificar el código original. OpenMP no es compatible para statements break o while, los cuales aparecían varias veces en el código. Por lo tanto, estas secciones se modificaron para que puedan utilizarse solo for y if statements. Se registró cuáles eran las secciones no paralelizables y las que tenían mayor recurrencia en el código. Converientemente, la sección principal más recurrente, es compatible con paralelismo mediante el statement #pragma-for.

Respecto al resto del código, cabe resaltar que gran parte del código se basa en condicionales de forma break que interrupen el código. Además, hay recursividad en algunas variables, lo cual no puede ser manejado adecuadamente con OpenMP, a menos que se modifique la forma del algoritmo, por lo que escapó del objetivo a realizarse. Estos segmentos de código pueden apreciarse en las Figuras 2, 3, 4, 5. Pese a ello, se resalta nuevamente que la sección del código con más recurrencia sí ha sido paralelizado.

Estos fragmentos de código presentan diversos factores que no permiten una paralelización normal con OpenMP, tal como la comunicación necesaria con la memoria adyacente para cada iteración, ya se adelante a otras con un índice +1 o -1.

3) Implementación: El código realizado sigue el control de versiones git y se muestra en el siguiente repositorio en Github *https://github.com/jhonatanmacazana/parallel-computing-project*. Notese el historial y la linea de commits realizados conjunto los releases.

Se plantearon dos tipos de experimentos para revisar el comportamiento del sistema. Para el primer caso, se plantea revisar el comportamiento por un tiempo $T = 10s$, para un $m_i = 1kg$, excepto para un $n_0 = 30, 60$, donde $m_{n_0} = 100kg$.

```
for (i = m - 1; i >= l; i--) {
  if (flag_inner_1) {
    f        = s * e[i];
    b        = c * e[i];
    e[i + 1] = (r = pythag(f, g));

    if (r == 0.0) {
      d[i + 1] -= p;
      e[m]      = 0.0;
      flag_inner_1 = false;
    }

    if (flag_inner_1) {
      s        = f / r;
      c        = g / r;
      g        = d[i + 1] - p;
      r        = (d[i] - g) * s + 2.0 * c * b
;     d[i + 1] = g + (p = s * r);
      g        = c * r - b;
      for (k = 0; k < n; k++) {
        f           = z[k][i + 1];
        z[k][i + 1] = s * z[k][i] + c * f;
        z[k][i]     = c * z[k][i] - s * f;
      } /* end k-loop */
    }
  }
}
```

Fig. 3. Fragmento de código 2.

```
for (m = l; m < n - 1; m++) {
  dd = abs(d[m]) + abs(d[m + 1]);
  if (abs(e[m]) <= EPS * dd) break;
}
```

Fig. 4. Fragmento de código 3.

```
for (i = 1; i < n; i++) {
  e[i - 1] = e[i];
}
```

Fig. 5. Fragmento de código 4.

El comportamiento en tiempo, se analizará posteriormente. Para el segundo lugar, se correrá el código para medir su desempeño. Se plantea correr 10 veces el código para $n = 59, 69, 79, 89, 99$. Para el caso en paralelo, se correrá el código para $p = 2, 4, 6, 8$. Todos estos casos se correrán de manera local, en el cluster proporcionado por UTEC y en un servidor propio para benchmarking.

Para agilizar el desarrollo y poder compilar en diferentes entornos de manera simple, se crea un archivo *Makefile* en donde las directivas existentes puededn mostrarse con el comando *make help*. Con estas directivas, es posible compilar, ejecutar y limpiar los archivos generados con los programas tanto secuencial como paralelo. Se utiliza el formato de *make seq* o *make par* para compilar el codigo en secuencial o paralelo respectivamente. Asimismo, existen directivas en el código para mostrar salidas en consola (-DDEBUG) o en un archivo (-DEXPORT). Se puede ejecutar ambas versiones con ayuda del *Makefile* utilizando el formato *make version-environment*, en donde *version* puede ser **seq** o **seq** y *environment* puede ser **debug** o **output**.

Es así que con estos comandos disponibles, se creó un archivo para facilitar la generación de tests, el cual es el archivo *testLocal.sh*. Este archivo permite compilar, ejecutar, y limpiar los archivos de manera correcta.

Por último, para el cálculo de las operaciones realizadas se utilizó el software *perf*, el cual se utilizó en el servidor remoto con 8 GB RAM (7.78) y 4 núcleos, cada núcleo siendo del modelo Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz.

III. Resultados

A. Simulaciones

De acuerdo al diseño planteado, se registraron los tiempos promedios de la función tqli, que es la sección que ha sido paralelizada, y también del código para un análisis por MIPS. Veáse en la figura 6, el comportamiento de las partículas en tiempo para las partículas vecinas de $n_0 = 30$. Cabe notar que el comportamiento en tiempo de cada partícula es oscilatoria a excepción de $n_0 = 30$. El movimiento de esta partícula es constante, pues debido a su excesiva masa con respecto a sus vecinos que impedia el movimiento inercial.

En la figura 7, se muestra el comportamiento para $n_0 = 60$ de las partículas entre 25 y 45. En ella se esalta que el comportamiento constante de la partícula 30, yo no se presenta pues ahora $n_0 = 60$. Además, en la figura 8, nuevamente se observe una de las lineas con comportamiento constante que representa el movimiento de la partícula 60

Cabe resaltar que convenientemente, se eligen los parámetros del sistema para que las oscilaciones sean correctas y aproximadamente reales.

B. Desempeño en tiempo

Para medir el desempeño, se corrieron los códigos múltiples veces hasta 10, y así obtener un valor promedio respecto a los procesadores. Se utilizaron dos dispositivos para medir el desempeño. Una pc local de 8 cores AMD Ryzen 5 3400G de 3.69GHz y 22 Gb de RAM. También se usó un servidor

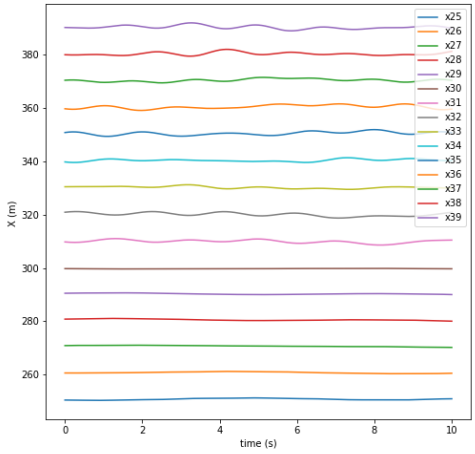

Fig. 6. Plot x25 a x40 en tiempo para n0 = 30

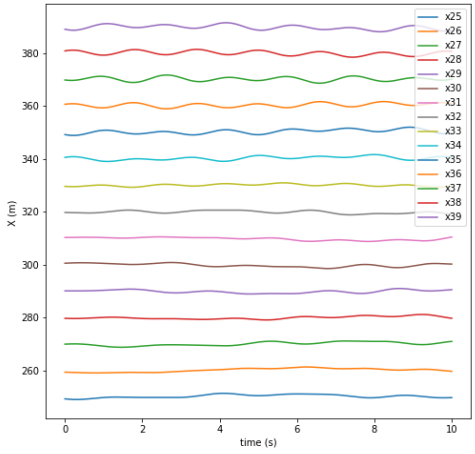

Fig. 7. Plot x25 a x40 en tiempo para n0 = 60

customizable de 8 Gb de RAM, y 4 cores de procesadores de 2.2 GHz. En las figuras 9 y 10 , se muestran el desempeño en tiempo. Notese que para distintos procesadores, el tiempo permanece siendo el mismo. Se justifica este comportamiento debido a que la sección paralela solo posee pocas operaciones y OpenMP implicitamente añade ciertos tiempos cuando se usa los statements pragma. Se justifica que los tiempos en el servidor sean menores al de la pc local, pues este servidor está dedicado a la tarea, mientras que la pc usa sus recursos para multiples procesos.

C. Velocidad

Para medir la velocidad del programa, se decidió por usar la librería perf que puede calcular el número total de instrucciones y el tiempo de proceso. A partir de estos resultados se

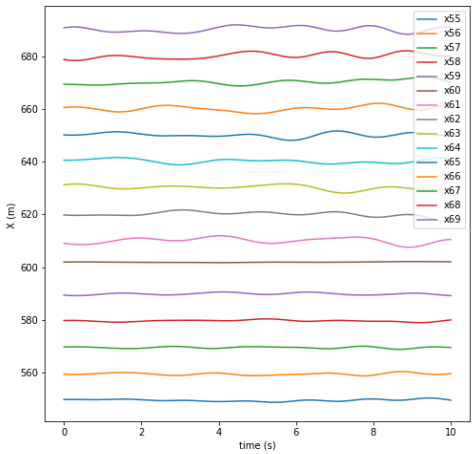

Fig. 8. Plot x55 a x70 en tiempo para n0 = 60

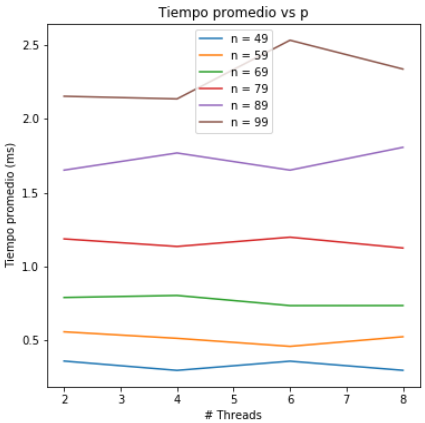

Fig. 9. Tiempo promedio de la función tqli respecto a numero de procesos localmente.

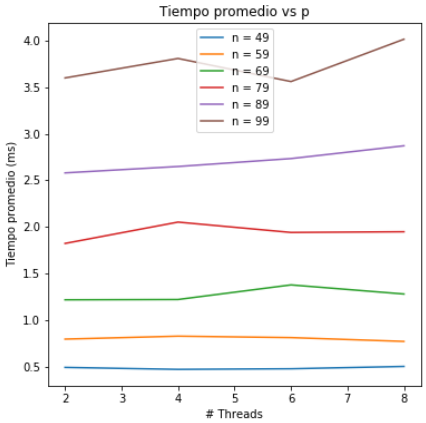

Fig. 10. Tiempo promedio de la función tqli respecto a numero de procesos en el servidor.

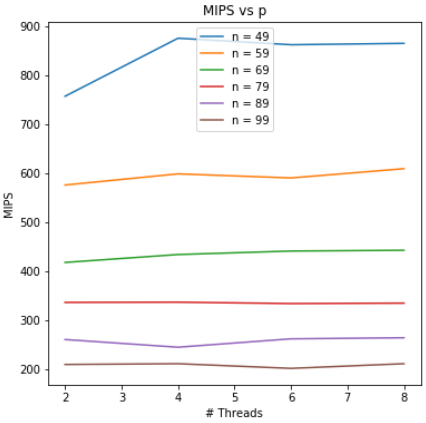

Fig. 11. MIPS del programa respecto a numero de procesos localmente.

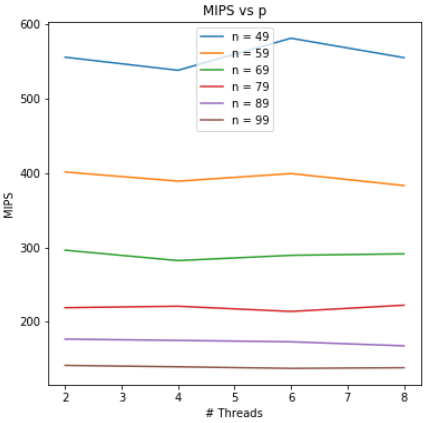

Fig. 12. MIPS del programa respecto a numero de procesos en el servidor.

graficaron las figuras 9 y 10. Notese que el comportamiento de todas las curvas no depende del numero de procesadores significativamente. Se puede justificar este comportamiento debido por la recursividad del algoritmo y la imposibilidad del paralelismos en todas sus secciones. Además, debido a su independencia del número de procesos, y la creciente respecto al número de elementos, también se dirá que el sistema no es escalable pues no cumple con la definición de convergencia de las curvas de velocidad.

IV. Conclusiones

Fue posible recrear el modelo de ecuaciones diferenciales para la resolución del problema de un oscilador armónico acoplado con modificaciones a la librería mostrada en el libro Numerical Recipes. Se modificaron los algoritmos para que sean compatibles con el modelo de trabajo de OpenMP sin una drástica modificación en su desempeño. Se logró implementar

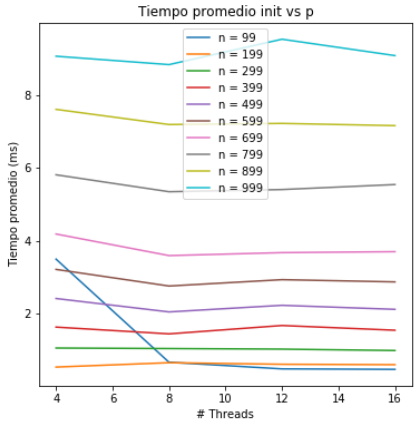

Fig. 13. Tiempo de sección del código de inicio. Obtenido del promedio de 10 iteraciones localmente.

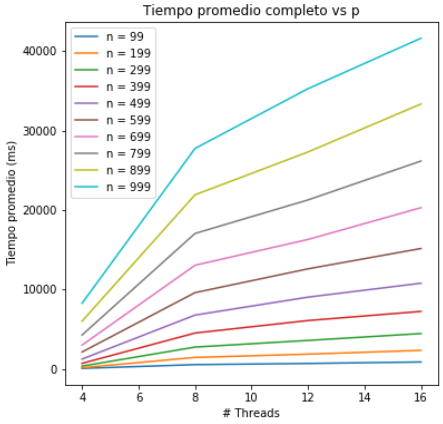

Fig. 14. Tiempo de sección del código de generación de matrices. Obtenido del promedio de 10 iteraciones localmente.

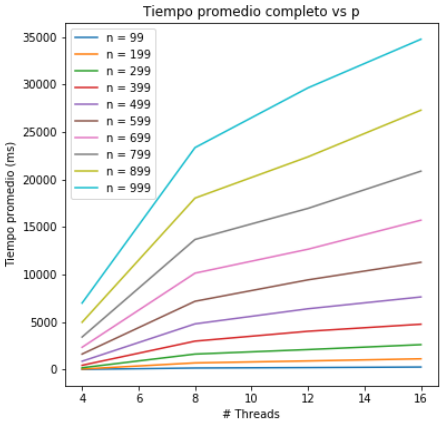

Fig. 15. Tiempo de sección del código de función tqli. Obtenido del promedio de 10 iteraciones localmente.

el caso con la cadena defectuosa y la cadena arregalada con condiciones iniciales. Se realizaron pruebas en local y en un servidor remoto para comparar tiempos exitosamente.

Referencias

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[2] A. M. Al-Oraiqat, "Parallel implementation of the coupled harmonic oscillator," *arXiv preprint arXiv:1702.02207*, 2017.