Universidad Nacional Autónoma de México

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Malváez Flores Axel Daniel

Peralta Rionda Gabriel Zadquiel

Particle Swarm Optimization paralelo en Julia y C

Computación Concurrente

---

4 de Diciembre de 2022

1. Objetivo del proyecto

El objetivo del proyecto es presentar los conceptos básicos del algortimo de optimización por enjambre de partículas ("Particle Swarm Optimization",PSO) al igual que hacer una implementación de este algoritmo en el lenguaje de programación Julia y C haciendo uso de memoria distribuida.

2. Introducción

Los algoritmos de optimización por enjambre de partículas (PSO, Particle Swarm Optimization) surgieron en el sexto simposio internacional de micro-máquinas y ciencias humanas de 1995 [12], como un nuevo modelo de optimización propuesto por Russel Eberhart y James Kennedy. En este primer planteamiento se presentó un algoritmo bio-inspirado, diseñado para solucionar problemas no lineales como, por ejemplo: el entrenamiento de redes neuronales, tareas de aprendizaje para robots, etc. Este algoritmo tiene la característica de ser computacionalmente económico, pues se maneja una alta velocidad, requiere poco uso de memoria y además emplea operaciones matemáticas primitivas.

PSO está incluido en la tecnología de vida artificial (A-life), esto se debe a que requiere un conocimiento mutuo entre cada partícula y con esta información el «enjambre» se aproxima hacia la respuesta más acertada según el conocimiento colectivo, por ello este algoritmo se considera evolutivo.

El planteamiento inicial de PSO, se basa en el mecanismo de comunicación de los enjambres de abejas, los bancos de peces o los nichos de aves, donde cada partícula viaja sobre el campo de búsqueda teniendo una visión individual y sesgada del mismo, esta visión se complementa con la visión parcial de sus compañeras, lo que favorece a un conocimiento colectivo y completo del campo de búsqueda. De esta forma, las abejas pueden encontrar la región del campo de flores con más alimento, lo que en un problema general representa la mejor solución encontrada por las partículas, según unos criterios de evaluación.

A diferencia de otros métodos similares, la optimización por ejambre de partículas no tiene opereaciones que implican el cruzamiento o la mutación de elementos de la población. Con esto se evita el suso de sistemas de numeración distintos al del espacio de búsqueda. Los operadores de PSO permiten explicar el espacio de búsqueda y a la vez explotar las regiones en las que se tiene más posibilidad de encontrar el óptimo global.

3. Concepto y planteamiento del algoritmo

Cada partícula tiene una posición y una velocidad asociada, las cuales se van modificando de acuerdo a dos paradigmas planteados por Eberhart y Kennedy, pBest y gBest. La mejor posición localmente encontrada por cada partícula (pBest), se refiere a la región del campo de flores con más alimento, según una abeja en particular bajo su visión parcial. La mejor posición global del enjambre (gBest), la cual se selecciona entre las pBest de todas las partículas. Los criterios de evaluación permiten saber si una región del espacio de búsqueda es la mejor solución, donde hay «más alimento» para las abejas, la cual determina la gBest.

El objetivo es que en cada iteración las partículas encuentren una posición más cerca a la comida, para ello cada vez que se encuentra una nueva gBest las partículas cambian su velocidad asociada, una de sus componentes es dirigida a la mejor posición global, otra componente hacia la mejor posición personal y otra componente inercial (referida a donde originalmente estaba dirigida la partícula antes de la actualización de gBest)

3.1. Algortimo

Entendida la ideología que toma el algoritmo, se muestra a continuación las principales fases del PSO estándar en pseudocódigo:

1. Inicializar la población de las partículas
2. Evaluar la población en la función objetivo y elegir la mejor partícula
3. **while** (criterio de paro)
4. **for** (todas las particulas en todas las dimensiones)
5. Generar una nueva velocidad
6. Calcular una nueva posición
7. Evaluar la función objetivo y elegir la mejor partícula
8. **end for**
9. Actualizar la mejor partícula de la población
10. **end while**

3.2. Desgloce del algoritmo y modelo matemático

1. **Inicialización**

   El algoritmo inicia con la inicialización de las particulas (las cuales nos referiremos como suluciones candidatas), estas partículas se generan distribuyendo de forma aleatoria las soluciones en el espacio de búsqueda acotado por lo límites superior e inferior definidas previamente. En esta etapa de inicialización se definen los parámetros del problema a optimizar, es decir las dimensiones, límites de busqueda y las restricciones(en caso de que exita alguna). Así podemos empezar con la descripción matemática de la inicialización del algortimo:

   $$l = \{\text{Es el límite inferior}\}$$

   $$u = \{\text{Es el límite superior}\}$$

   $$B = \{\text{Es el conjunto de busqueda restringudo por u y } l\}$$

   La siguiente ecuación describe la creación de las particulas:

   $$x_k^i = l_k + rand(u_k - l_k), \qquad \hat{x}_i \in B$$

   Donde: x_k^i representa a la entreda k-esima de la partícula i-esima de la población, el indice i-ésimo es el que representa a la particula de la población, donde i va de 1 al número de partuculas que desemos crear ($i \in \{1, 2, .., \text{NUM\_P}\}$). El número k define a la entrada k-ésima de la particula, pues recordemos que la particula es un punto en el espacio. Notemos en la creación de las partulas necesita un $rand(u_k - l_k)$ esta es la parte de la creación aleatoria del método, es decir esteremos generando un valor de forma aleatoria que se encuentre entre los límites correspondientes. Despues de la creación de las particulas el algortimo va a cambiar la posición de la particula, por ello en cada iteración nos referimos a la misma partícula pero en su posición al timpo t, es decir estaremos denotando a las partículas como $\hat{x}^{i,t}$ lo cual representa a la i-ésima partícula en el tiempo t.

2. **Velocidad de las partículas**

   Para poder obtener la nueva posición que tendrán las partículas en el espacio de búesqueda, primero es necesario calcular la velocidad de cada una de ellas. Para esto se necesara el valor precio de la velocidad, en caso de la primera iteración este valor será cero. Ademas se necesitan los mejroes valores globales y locales de cada partícula. La ecuación matemática que describe la velocidad de la partícula es:

   $$v^{t+1} = v^t + rand1 \times (P - X^t) + rand2 \times (G - X^t) \tag{1}$$

   Donde: v^{t+1} es el valor de la velocidad qeu se calcula para el tiempo t+1, v^t es la velocidad anterior, X^t es el vector que contiene las posiciones de cada partícula, P contiene las lmejores posiciones actuales asociadas a la vecindad de cada partícula, mientras que G es la mejor partícula actual a nivel global. Por otra parte rand1 y rand2 son números aleatorias distribuidos entre cero y uno.

3. **Movimiento de las partículas** Despues de calcular la velocidad las partículas son desplazadas hacia nuevas posiciones en la iteración actual tal y como se mencionaba en la parte

de inicialización. Para realizar este movimiento se realiza una simple ooperación donde se combina la velocidad con las posiciones anterioeres, esta accción se describe en la siguiente ecuación:

$$\hat{x}^{i,t+1} = x^{i,t} + v^{i,t+1} \tag{2}$$

Donde ahora $\hat{x}^{i,t+1}$ es la nueva posición de la partícula i en el tiempo t, donde $x^{i,t}$ representa la posicón donde estaba anteriormente la partícula y $v^{i,t+1}$ la velocidad que tenía esa partícula.

3.3. Estructura básica del algoritmo PSO

El diagrama de flujo del PSO básico se muestra en la Figura 1

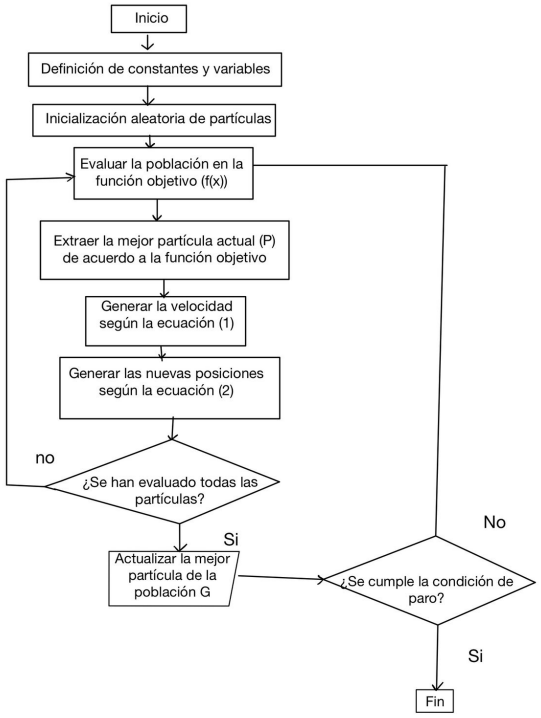

Figura 1: Diagrama de flujo del PSO

3.4. Estructura del algortimo PSO en MPI

Para implementarlo en una plataforma MPI es necesario modificar el algoritmo para que sean procesos paralelos, esto se hace haciendo uso de algunos de los métodos básicos de MPI. Prácticamente lo que se hizo fue crear un enjambre diferente en cada procesador, de esta forma se hace una mayor cobertura inicial del espacio de búsqueda, y luego cada procesador nos devuelve su mejor

partícula y lo que hará el nodo maestro será buscar entre este conjunto de particulas al mejor y el mejor de este conjunto será la mejor partícula al final.

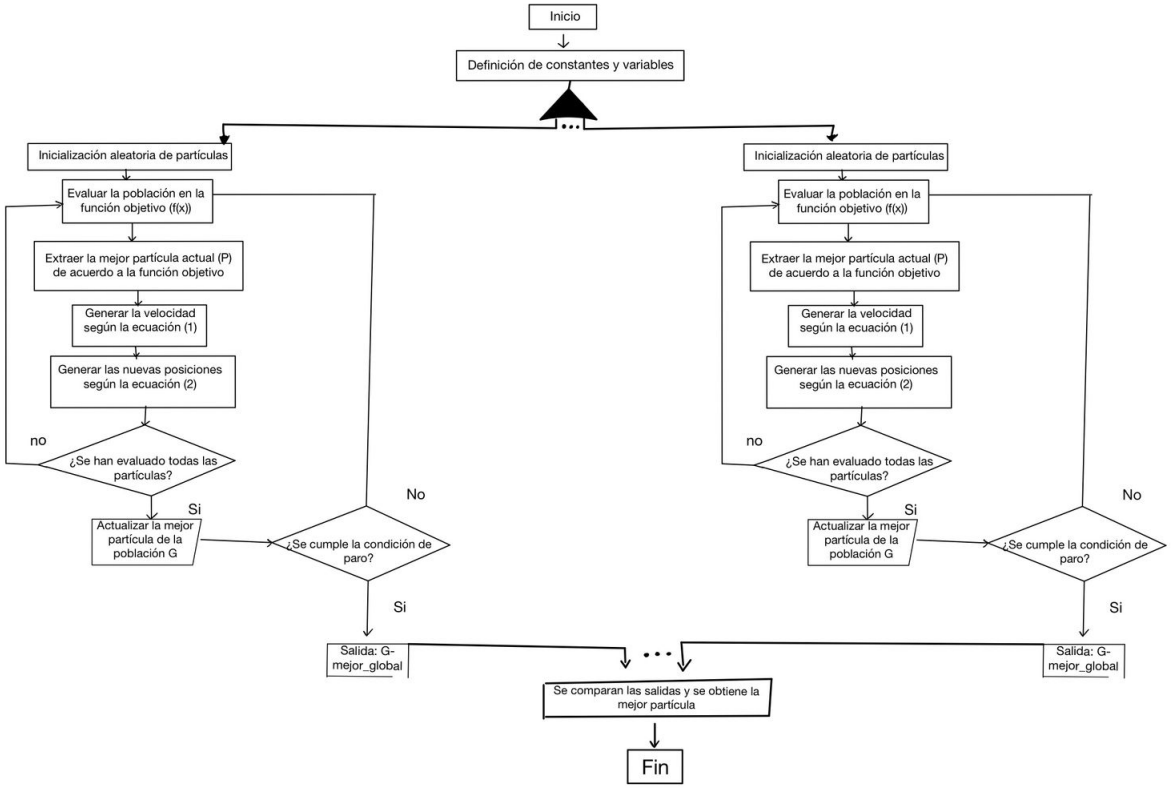

Figura 2: Diagrama de flujo del PSO en MPI

4. Implementaciones

4.1. Implementacion en Julia

En esta sección se verán los códigos que se realizaron para implementar el algoritmo PSO en el lenguaje de programación Julia, estos códigos se realizaron utilizando la parte teórica descritas en la sección anterior, es decir considerando el modelo matemático y la estructura de este algoritmo.

1. **Funciones con las que se prueban los algoritmos:**

almac > invitado_oaef > GabrielPeralta > Proyecto_Bueno > func.jl > ...

```
#--------------------------------
#Funciones de una dimensión
#--------------------------------
function f1_one(x,d=1)
  x = x[1]
  return (x^4)-(x^3)-10*(x^2)+4*(x)+24
end

function f2_one(x)
  #return exp(x) + x^2
  return x^2
end
#--------------------------------
#Funciones de varias dimensiones
#--------------------------------
function f1_multi(x,d=2)
    return (x[1])^2 + (x[2])^2
  end

function f2_multi(x, d=2) #Ecuación de Resenbrock
  res = 0
  for i = 1:d-1
    res += 100*(x[i+1]-x[i]^2)^2 + (1-x[i])^2
  end
  return res
end
```

Figura 3: Funciones

2. **Algoritmo PSO en una dimension:**

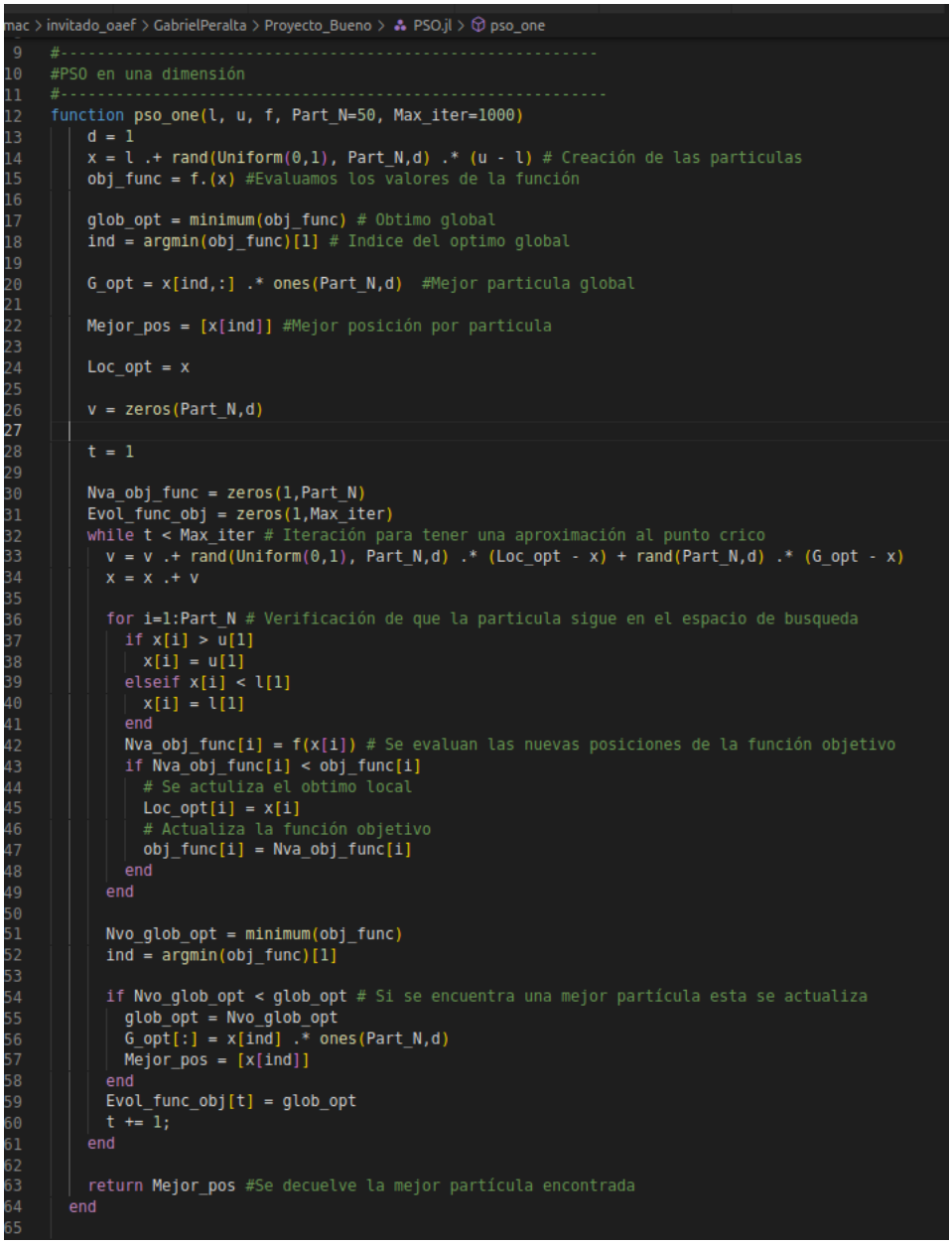

Figura 4: Algoritmo PSO para muchas dimensiones implementado en julia

3. **Algortimo PSO implementado en varias dimensiones**

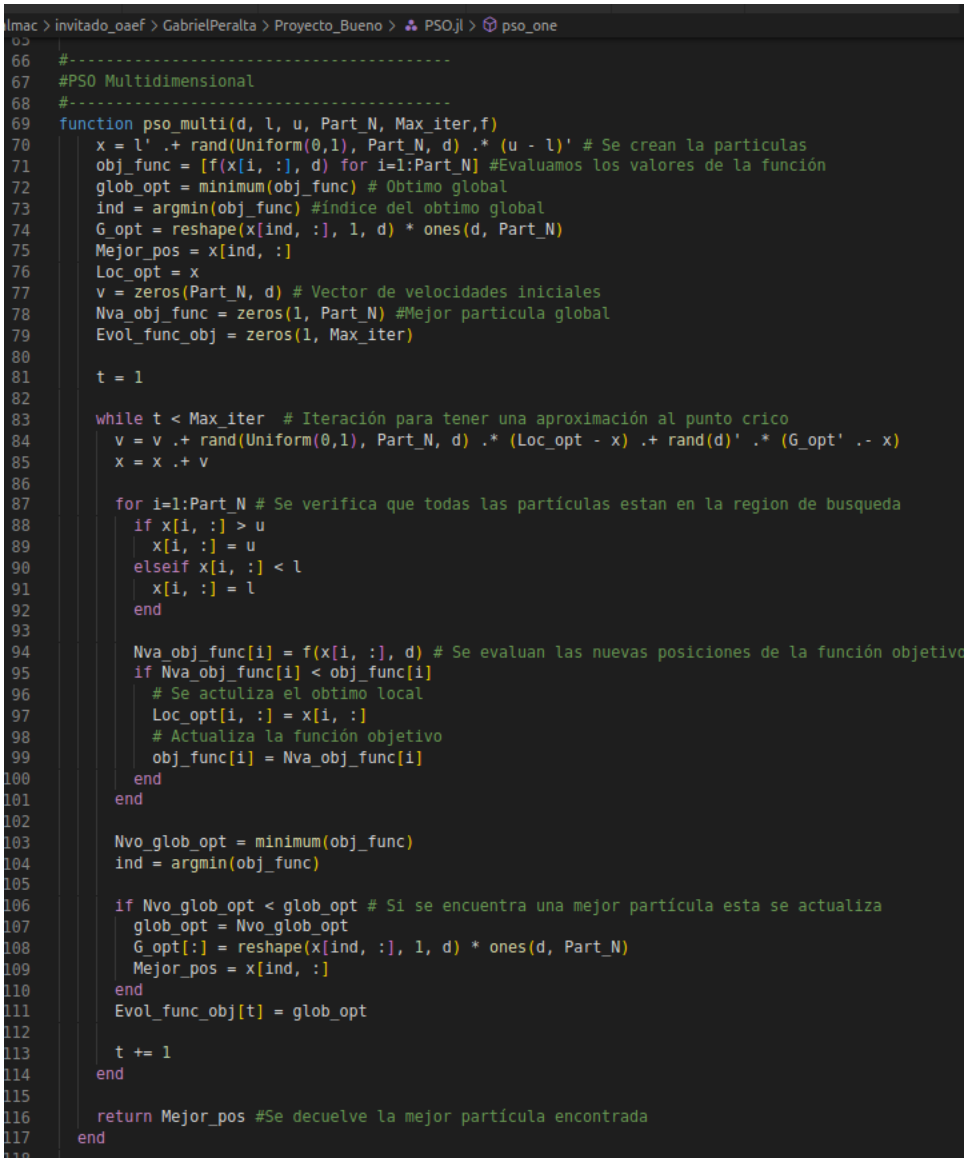

Figura 5: Algortimo PSO implementado en julia para varias dimensiones

4. **PSO con MPI implementado en una dimension:**

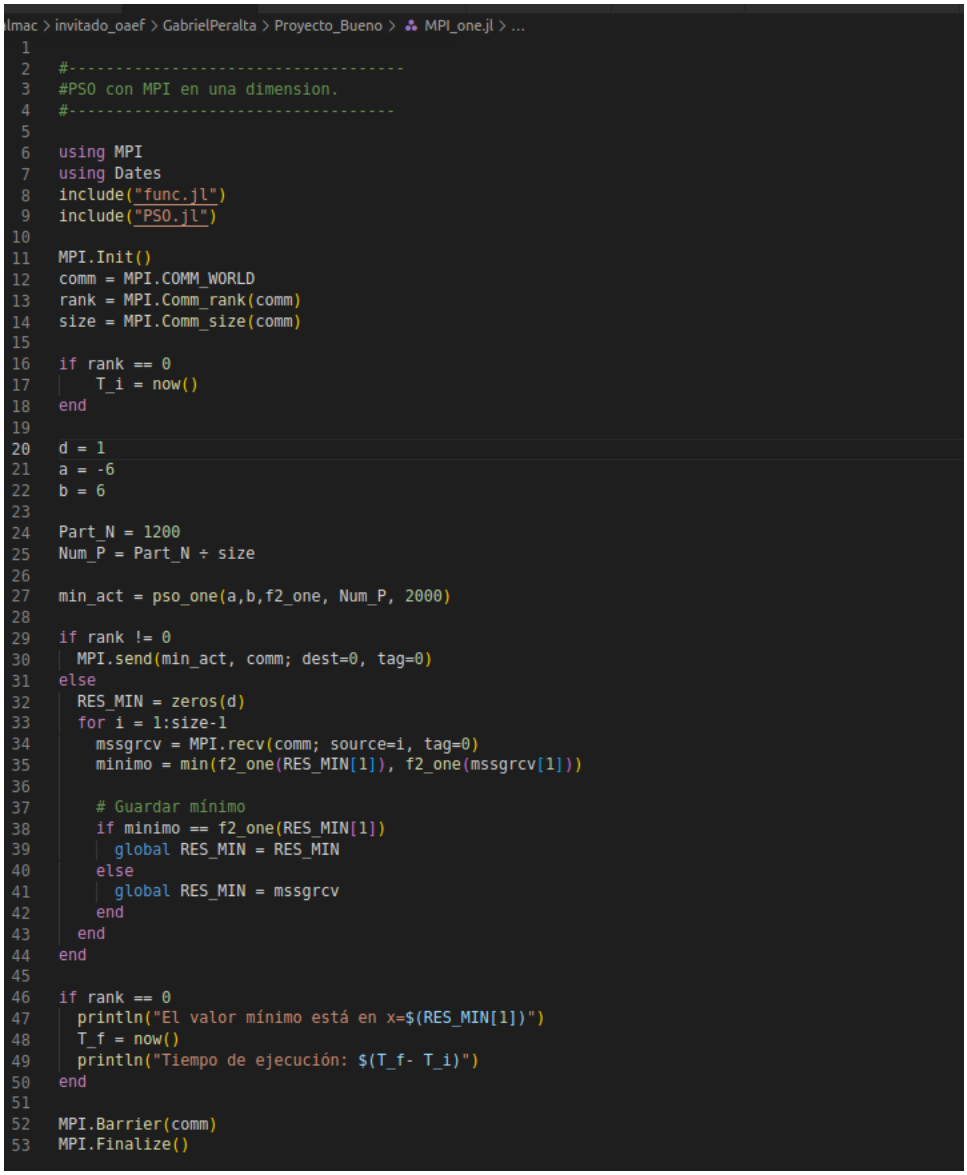

Figura 6: PSO con MPI implementado en una dimension en julia

5. **PSO con MPI implementado en muchas dimensiones:**

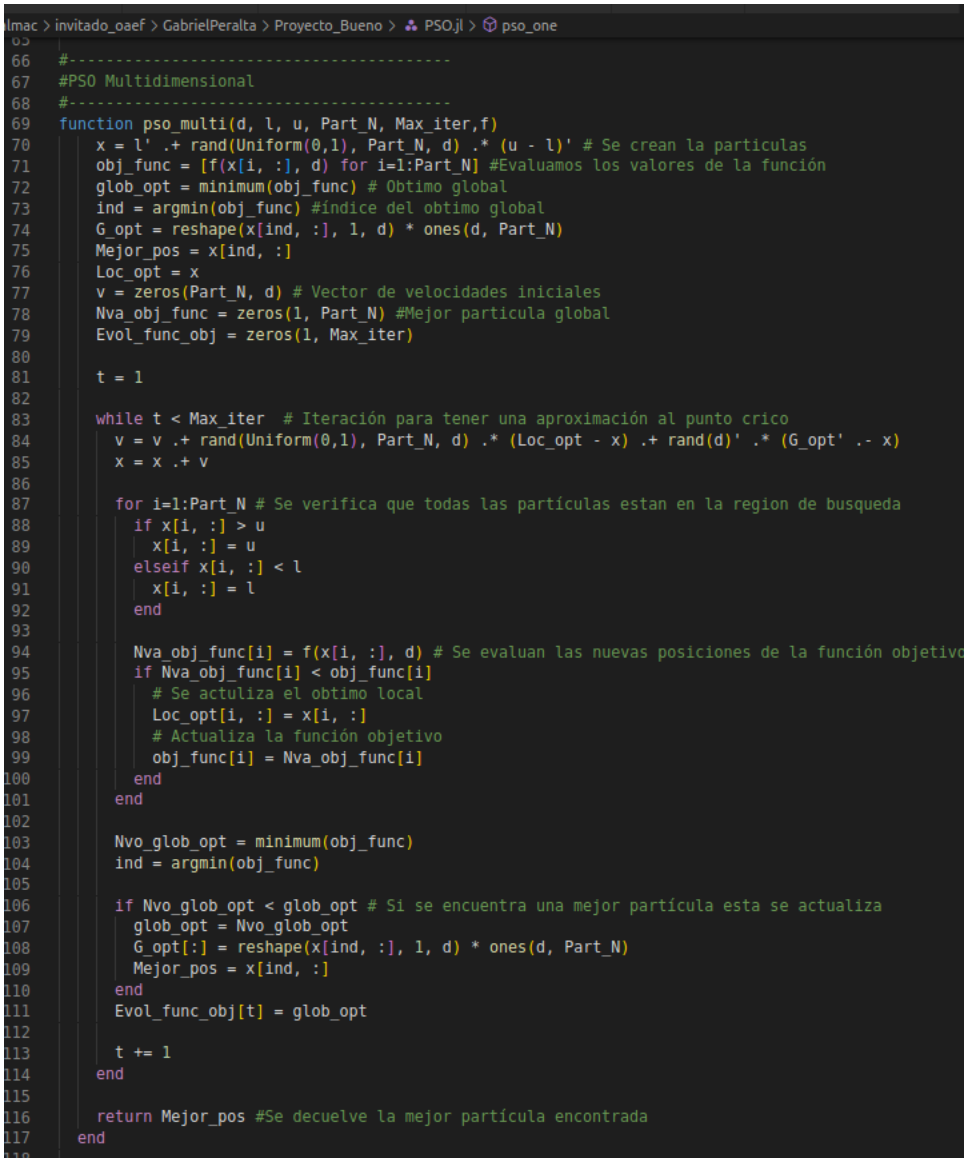

Figura 7: PSO con MPI implementado en muchas dimensiones en julia

4.1.1. Resultados de speed up

El código anteriormente visto se ejecuto en el cluster de computadoras del IIMAS y se optuvieron los siguientes resultados:

El siguiente comando se introdujo en la terminal y nos produjo los siguientes resultados:

```
for run in {1..10}; do mpiexec -n 12 julia MPI\_one.jl >> Pruebas\_one\_MPI.txt;
    done
```

Resultados del PSO secuencial en una dimension:

Las pruebas se realizaron con 1200 particulas y 2000 iteraciones

---

El valor mínimo está en [7.496127952810738e-7]

Tiempo de ejecución: 1966 milliseconds

El valor mínimo está en [6.782416832606941e-7]

Tiempo de ejecución: 1956 milliseconds

El valor mínimo está en [2.324253567720369e-6]

Tiempo de ejecución: 1949 milliseconds

El valor mínimo está en [-1.7918887320611532e-5]

Tiempo de ejecución: 1942 milliseconds

El valor mínimo está en [3.0502085583542816e-6]

Tiempo de ejecución: 1952 milliseconds

El valor mínimo está en [-3.2488787660156504e-7]

Tiempo de ejecución: 1940 milliseconds

El valor mínimo está en [7.616294892542896e-7]

Tiempo de ejecución: 1943 milliseconds

El valor mínimo está en [-1.0620802755219316e-6]

Tiempo de ejecución: 1933 milliseconds

El valor mínimo está en [2.314849015513931e-6]

Tiempo de ejecución: 1965 milliseconds

El valor mínimo está en [-2.3484925293359993e-7]

Tiempo de ejecución: 1931 milliseconds

Tiempos de ejcución: [1966, 1956, 1949, 1942, 1952, 1940, 1943, 1933, 1965, 1931]

Tiempo minimo: 1931

Tiempo maximo: 1966

Tiempo promedio: 1947.7

Std del tiempo: 12.165981715879369

Respuestas:

[7.496127952810738e-7, 6.782416832606941e-7, 2.324253567720369e-6, -1.7918887320611532e-5,

3.0502085583542816e-6,-3.2488787660156504e-7,7.616294892542896e-7,-1.0620802755219316e-6

2.314849015513931e-6,-2.3484925293359993e-7]

Pruebas de MPI_one.jl

Las pruebas se realizaron con 1200 particulas y 2000 iteraciones

---

El valor mínimo está en x=0.0

Tiempo de ejecución: 3699 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 4903 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 5061 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 4687 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 3972 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 4421 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 4320 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 3431 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 3422 milliseconds

El valor mínimo está en x=0.0

Tiempo de ejecución: 3483 milliseconds

Los tiempos de ejcución fueron: [3699, 4903, 5061, 4687, 3972, 4421, 4320, 3431, 3422, 3483]

Tiempo minimo: 3422

Tiempo maximo: 5061

Tiempo promedio: 4139.9

Std del tiempo: 624.8145413729669

Respuestas: [0,0,0,0,0,0,0,0,0,0]

Análisis de resultados

Notamos que el algortimo MPI fue mucho más preciso en todas las ejecuciones que el algortimo secuencial, esto se debe al aumento de eficiencia que tiene programarlo en MPI, pues todos los procesadores estan coperando para brindar la mejor partícula.

4.2. Implementación en C

En esta sección se verán los códigos que se realizaron para implementar el algoritmo PSO en el lenguaje de programación C, estos códigos se realizaron utilizando la parte teórica descritas en la sección anterior, es decir considerando el modelo matemático y la estructura de este algoritmo.

1. **Funciones con las que se prueban los algoritmos:**

```
/**
 * Funcion que sera evaluada (R en R)
*/
double f(double x) {
    double res = (x*x*x*x) - (x*x*x) - (10*x*x) + 4*x + 2;
    return res;
}

/**
 * Funcion que sera evaluada (R en R)
*/
double f2(double x) {
    double res = x*x;
    return res;
}

/**
 * Min function
*/
double min(double a, double b){
    return (a > b) ? b : a;
}
```

2. **Algoritmo PSO:**

```
/**
 * Funcion del metodo PSO
*/
double pso(int d, double l, double u, double (*f)(double), int Part_N,
    int Max_iter){
    // Generamos las partculas
    double x[Part_N];
    srand(time(NULL)); // randomize seed
    double diff;
    int i;
    diff = u - l;
    for(i = 0; i < Part_N; i++)
    {
        x[i] = l + ((double) (rand() % RAND_MAX) / (double)RAND_MAX) *
            diff;
    }


    //Obtenemos el vector de particulas evaluadas en la funcion objetivo
    double obj_func[Part_N], temp;
    int j;
    for(j = 0; j < Part_N; j++)
    {
        temp = x[j];
        obj_func[j] = f(temp);
    }

    // Obtenemos el optimo global
    double glob_opt = obj_func[0];
    int k;
    //Obtenemos el indice del optimo global
    int ind = 0;
    for(k=1; k < Part_N; k++)
    {
        if(glob_opt>obj_func[k])
        glob_opt = obj_func[k];
            ind = k;
    }

    //Obtenemos un G_opt de todas las particulas
    double G_opt[Part_N];
    for(i=0; i < Part_N; i++){
        G_opt[i] = x[ind];
    }

    // Obtenemos la mejor posicion
    double Mejor_pos = x[ind];
    double * Loc_opt = x;
```

```
// Vector de velocidad
double v[Part_N];
for(i=0; i < Part_N; i++){
    v[i] = 0;
}

int t = 1;

double Nva_obj_func[Part_N];
double Evol_func_obj[Max_iter];
for(i = 0; i < Part_N; i++){
    Nva_obj_func[i] = 0;
}
for(i = 0; i < Max_iter; i++){
    Evol_func_obj[i] = 0;
}

while (t < Max_iter){
    // Actualizacion de la nueva velocidad
    double diff_loc[Part_N], diff_glob[Part_N];
    for(i = 0; i < Part_N; i++){
        diff_loc[i] = Loc_opt[i] - x[i];
        diff_glob[i] = G_opt[i] - x[i];
    }
    double rnd_1, rnd_2;
    rnd_1 = ((double) (rand() % RAND_MAX) / (double)RAND_MAX);
    rnd_2 = ((double) (rand() % RAND_MAX) / (double)RAND_MAX);
    for(i = 0; i < Part_N; i++){
        v[i] = v[i] + rnd_1 * diff_loc[i] + rnd_2 * diff_glob[i];
    }
    for(i=0; i<Part_N; i++){
        x[i] += v[i];
    }

    // Verificacion de que la particula sigue en el espacio de busqueda
    for(i = 0; i < Part_N; i++){
        if(x[i] > u){
            x[i] = u;
        } else if (x[i] < l){
            x[i] = l;
        }
        Nva_obj_func[i] = f(x[i]); // Se evaluan las nuevas posiciones
        if (Nva_obj_func[i] < obj_func[i]){
            // Actualizamos el optimo local
            Loc_opt[i] = x[i];
            // Actualizamos la funcion objetivo
            obj_func[i] = Nva_obj_func[i];
        }
    }
```

```
        double Nvo_glob_opt;
        int idx;
        Nvo_glob_opt = obj_func[0];
        idx = 0;
        for(i = 1; i<Part_N; i++){
            if(Nvo_glob_opt > obj_func[i]){
                Nvo_glob_opt = obj_func[i];
                idx = i;
            }
        }

         if(Nvo_glob_opt < glob_opt){
            glob_opt = Nvo_glob_opt;
            for(i = 0; i < Part_N; i++){
                G_opt[i] = x[idx];
            }
            Mejor_pos = x[idx];
        }
        Evol_func_obj[t] = glob_opt;
        t += 1;
    }

    return Mejor_pos;
}
```

3. **PSO con MPI implementado en C para una dimension:**

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>

double pso(int d, double l, double u, double (*f)(double), int Part_N,
    int Max_iter);
double f(double x);
double f2(double x);
double min(double a, double b);

int main(void) {
    clock_t start, end;
    double execution_time;

    start = clock();

    int my_rank, comm_sz, d;
    double a, b, min_act;
    int source;
    int Part_N, Num_P, Max_iter;
```

```
    double RES_MIN, mssgrcv, minimo;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    d = 1;
    a = -6;
    b = 6;
    Part_N = 100000;
    Num_P = Part_N / comm_sz;
    Max_iter = 3000;
    min_act = pso(d, a, b, &f, Num_P, Max_iter);

    RES_MIN = 0.0;

    if (my_rank != 0) {
        MPI_Send(&min_act, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    } else {
        RES_MIN = min_act;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&min_act, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            minimo = min(f(RES_MIN), f(min_act));
            // Guardamos el mnimo
            if(minimo != f(RES_MIN)){
                RES_MIN = min_act;
            }
        }
    }

    if (my_rank == 0) {
        printf("Con %d procesadores\n", comm_sz);
        printf("El valor minimo esta en x = %f \n", RES_MIN);
        end = clock();
        execution_time = ((double)(end - start))/CLOCKS_PER_SEC;
        printf("Tiempo de ejecucin: %f \n", execution_time);
    }

    MPI_Finalize();

    return 0;
}
```

4.2.1. Resultados de speed up

El código anterior se ejecuto en el clúster de computadoras del IIMAS. Primeramente introduciremos el comando que ejecutamos en la terminal para que nos produciera los siguientes tiempos:

```
for run in {1..10}; do mpiexec -n 12 julia MPI\_one.jl >>
   Pruebas\_one\_MPI.txt; done
```

Resultados del PSO paralelo en C para la función f:

Las pruebas se realizaron con 100000 particulas y 3000 iteraciones

---

Con 12 procesadores El valor minimo esta en x = -3.522211 Tiempo de ejecución: 0.969078
Con 12 procesadores El valor minimo esta en x = 2.554248 Tiempo de ejecución: 0.948213
Con 12 procesadores El valor minimo esta en x = 2.554249 Tiempo de ejecución: 0.961496
Con 12 procesadores El valor minimo esta en x = -4.168364 Tiempo de ejecución: 0.832474
Con 12 procesadores El valor minimo esta en x = 2.554248 Tiempo de ejecución: 0.814402
Con 12 procesadores El valor minimo esta en x = 2.554247 Tiempo de ejecución: 0.825283
Con 12 procesadores El valor minimo esta en x = 2.554252 Tiempo de ejecución: 0.879470
Con 12 procesadores El valor minimo esta en x = 2.554246 Tiempo de ejecución: 0.870560
Con 12 procesadores El valor minimo esta en x = -3.274767 Tiempo de ejecución: 0.844582
Con 12 procesadores El valor minimo esta en x = 2.554244 Tiempo de ejecución: 0.864845

Tiempos de ejecución:

[0.969078, 0.948213, 0.961496, 0.832474, 0.814402, 0.825283, 0.879470, 0.870560, 0.844582, 0.864845]

Tiempo minimo: 0.814402

Tiempo maximo: 0.969078

Tiempo promedio: 0.8810403000000001

Std del tiempo: 0.0580688775374172

Respuestas:

[-3.522211, 2.554248, 2.554249, -4.168364, 2.554248, 2.554247, 2.554252, 2.554246, -3.274767, 2.554244]

Resultados del PSO paralelo en C para la función f2:

Las pruebas se realizaron con 100000 particulas y 3000 iteraciones

---

Con 12 procesadores El valor minimo esta en x = -0.000000 Tiempo de ejecución: 0.773119
Con 12 procesadores El valor minimo esta en x = 0.000009 Tiempo de ejecución: 0.792792
Con 12 procesadores El valor minimo esta en x = 0.000001 Tiempo de ejecución: 0.714995
Con 12 procesadores El valor minimo esta en x = -0.000001 Tiempo de ejecución: 0.692070
Con 12 procesadores El valor minimo esta en x = 0.000002 Tiempo de ejecución: 0.770666
Con 12 procesadores El valor minimo esta en x = 0.000001 Tiempo de ejecución: 0.733369
Con 12 procesadores El valor minimo esta en x = -0.000003 Tiempo de ejecución: 0.804421
Con 12 procesadores El valor minimo esta en x = -0.000006 Tiempo de ejecución: 0.844582
Con 12 procesadores El valor minimo esta en x = 0.000002 Tiempo de ejecución: 0.746474
Con 12 procesadores El valor minimo esta en x = 0.000006 Tiempo de ejecución: 0.887742

Tiempos de ejecución:

[0.773119, 0.792792, 0.714995, 0.692070, 0.770666, 0.733369, 0.804421, 0.844582, 0.746474, 0.887742]

Tiempo minimo: 0.69207

Tiempo maximo: 0.887742

Tiempo promedio: 0.7760230000000001

Std del tiempo: 0.05941499808970798

Respuestas:

[-0.000000, 0.000009, 0.000001, -0.000001, 0.000002, 0.000001, -0.000003, -0.000006, 0.000002, 0.000006]

4.2.2. Análisis de resultados

La implementación del algoritmo fue correcta y nos arrojaba valores muy precisos en cualquiera de los lenguajes de programación. Al tratarse del tiempo notamos que al ejecutar el programa compilado, los tiempos eran bastante buenos a comparación de los tiempos en Julia MPI.

5. Conclusiones

La sintáxis en Julia es mucho más fácil de leer y entender para un programador que en C, no obstante la diferencia en tiempos sigue siendo notable, aunado a esto hacerlo en paralelo reducía el tiempo de ejecución que en secuencial. Por lo que hemos concluido que hacerlo de manera paralela y con memoria distribuida (clúster de computadoras) fue la manera más rápida y eficiente de encontrar el resultado esperado.

Por otra parte notamos que el algoritmo PSO tenía una implementación muy natural en paralelo, esto es muy bueno para nuestro proposito ya que no todos los algoritmos son facilmente implementados de forma paralela, por ello aunque estemos utilizando muchas partículas para mejorar los resultados de la busqueda del mínimo, podemos mejorar la eficiencia de estos tal y como fue nuestra tarea de implementarlo con MPI.

Tambien es importante comentar la escalabilidad que tienen nuestras implementaciones ya ques estas no dependian del número de procesadores, al programarlo lo pensamos para ques el algoritmo se pudiera escalar en un cluster de compuradoras con mayor capacidad de procesadores, tambien aunque con las pruebas solo se busco el minimo, podemos buscar el maximo de distintas funciones. Por otra parte notamos que no importa como sea la dimension del diminio, el algoritmo es tan general que puede encontrar los mínimos o máximos en cualquier dimension.