Facultad
de
Ciencias

Paralelización de técnicas de toma de decisiones en empresas de acuicultura.

(Parallelisation of decision-making techniques in aquaculture enterprises.)

Trabajo de Fin de Máster
para acceder al

Máster Universitario de Ingeniería Informática

Autor: Mario Ibáñez Bolado

Director: José Luis Bosque Orero

Mayo-2022

Índice general

Índice de Figuras IV

Índice de códigos de programación V

Agradecimientos VI

Resumen VII

Abstract VIII

1. Introducción 1

1.1. Computación Paralela 1

1.2. Toma de Decisiones en Empresas 3

1.3. Objetivos 3

1.4. Metodología y Plan de trabajo 4

1.5. Estructura del Documento 5

2. Background 6

2.1. Conceptos Básicos 6

2.1.1. Acuicultura 6

2.1.2. Particle Swarm Optimization 8

2.2. Message Passing Interface 10

2.2.1. Inicialización de procesos 11

2.2.2. Comunicación punto a punto y llamadas colectivas 11

2.2.3. Comunicadores y grupos de procesos 15

2.3. Trabajos Relacionados 16

2.3.1. Planificación en Acuicultura 16

2.3.2. Paralelización Particle Swarm Optimization 17

3. Implementación de la Paralelización **18**

3.1. Programa Secuencial 18

3.2. Profiling 21

3.3. Paralelización del acceso a la base de datos 23

3.4. Paralelización PSO 24

3.4.1. Tipo de paralelización 24

3.4.2. Puntuación y restricciones de jaulas 26

3.4.3. División del cómputo y comunicación de resultados 26

3.4.4. Comparación y envío de los resultados 32

3.4.5. Cálculo de la velocidad y posición 33

3.4.6. Reparto de las partículas y sus valores 35

4. Evaluación **36**

4.1. Metodología 36

4.2. Experimentos de rendimiento 40

4.3. Experimentos de escalabilidad 43

4.4. Análisis de Trazas y Modelo de Predicción 49

5. Conclusiones **52**

5.1. Objetivos Conseguidos 52

5.2. Trabajos Futuros 53

Índice de figuras

3.1. Algoritmo Secuencial PSO implementado en AquiAID 20

3.2. Representación de jaulas en una partícula 20

3.3. Ejemplo de reparto de las Jaulas 27

3.4. Esquema de comunicaciones y cómputo del algoritmo PSO paralelo. 29

3.5. Ejemplos de reparto de jaulas 34

4.1. Tiempo de ejecución del programa secuencial vs paralelo 41

4.2. Eficiencia Energética del programa secuencial vs paralelo 42

4.3. Tiempo de ejecución de los distintos procesos con 20000 jaulas 44

4.4. Evolución del Speedup y Eficiencia en la escalabilidad 45

4.5. Consumo y Eficiencia en la escalabilidad 48

4.6. Traza del experimento de escalabilidad con 40 procesos 50

Índice de códigos de programación

1. Productor-Consumidor 12
2. Ejemplo Allreduce 13
3. Ejemplo Gatherv y Scatterv 14
4. Consulta del size y rank de un proceso 15
5. Creación de Grupos y Comunicadores 16

Agradecimientos

A mi familia, que me ha dado la oportunidad con su trabajo de llegar hasta donde estoy en mis estudios y de apoyarme en mis decisiones sobre mi futuro.

A mis amigos, con los que he pasado tan buenos momentos y en especial a mis amigos Dani y David. Dani por siempre estar ahí para desconectar y pasar tan buenos ratos en nuestras aficiones y a David por pasar el Máster sufriendo conmigo.

A mi madre mi padre, hermana y a mi pareja, que son los que más me han apoyado mucho en este duro Máster.

Y sobretodo agradecer a José Luis, que me presentó el proyecto y me ha guiado cada semana para avanzar, y conseguir este trabajo final. Además de guiarme para seguir con mi carrera haciendo este máster y a continuación el doctorado.

Resumen

Actualmente las empresas del sector de la Acuicultura deben de controlar una gran cantidad de factores para poder realizar su actividad de la manera más productiva. Su producción depende de factores ambientales, económicos, biológicos e ingenieriles. Esta alta complejidad, las empresas en colaboración con universidades buscan métodos para planificar la cría de peces de manera que aporte el máximo beneficio.

En este aspecto el grupo de investigación Gestión Económica para el Desarrollo Sostenible del Sector Primario (IDES) de la Universidad de Cantabria creó el software AquiAID para la gestión de empresas de acuicultura. Este software cuenta para su ejecución con un algoritmo de Inteligencia Artificial conocido como Particule Swarm Optimitation (PSO). Pero debido a su alto coste computacional es inviable su utilización en la práctica.

Este trabajo consiste en la mejora del rendimiento y del consumo energético del software AquiAID utilizando la técnica de paralelización en sistemas de memoria distribuida. Para esto se han centrado los esfuerzos en paralelizar, con el grano más fino posible, el algoritmo de inteligencia artificial que permitirá reducir su tiempo de cómputo, el cual es el mayor limitante del programa.

Se realizará una validación experimental que permitirá optimizar la paralelización y conocer los límites de la paralelización del algoritmo PSO. Estos experimentos validarán tanto el rendimiento del programa como el consumo energético, los cuales son fundamentales para la reducción en el coste empresarial.

Gracias a los cambios realizado en el programa paralelo implementado se obtienen reducciones de tiempo de hasta 40 veces el tiempo secuencial, y en la energía consumida de 20 veces respecto al original. Con estas mejoras, el software servirá a la industria de la acuicultura para realizar la gestión de la cría de peces. Todo ello gracias a la paralelización de un algoritmo de inteligencia artificial, lo cual no se ha observado actualmente en el mundo de la acuicultura.

Palabras Claves: Acuicultura, Paralelismo, Inteligencia Artificial, Toma de decisiones, Sistemas Distribuidos.

Abstract

Nowadays, aquaculture companies must control a large number of factors in order to be able to carry out their activity in the most productive way. Their production depends on environmental, economic, biological and engineering factors. Due to this high complexity, companies in collaboration with universities are looking for methods to plan fish farming in a way that brings the maximum benefit.

In this respect, the research group Gestión Económica para el Desarrollo Sostenible del Sector Primario (IDES) of the University of Cantabria created the AquiAID software for the management of aquaculture companies. This software uses an artificial intelligence algorithm known as Particule Swarm Optimitation (PSO). However, due to its high computational cost, it is unfeasible to use it in practice.

This work consists of improving the performance and energy consumption of AquiAID software using the technique of parallelisation in distributed memory systems. To this end, efforts have been focused on parallelising, with the finest possible grain, the artificial intelligence algorithm to reduce its computation time, which is the programme's main limitation.

Experimental validation will be carried out to optimise the parallelisation and to find out the limits of the parallelisation of the PSO algorithm. These experiments will validate both the performance of the programme and the energy consumption, which are fundamental for the reduction of the business cost.

Thanks to the changes made to the implemented parallel program, time reductions of up to 40 times the sequential time and energy consumption of 20 times compared to the original are obtained. With these improvements, the software will help the aquaculture industry to manage fish farming. All this thanks to the parallelisation of an artificial intelligence algorithm, which is currently unheard of in the world of aquaculture.

Key Words: Aquaculture, Parallelism, Artificial Intelligence, Decision-making, Distributed Systems.

Capítulo 1

Introducción

En este capítulo se exponen de manera genérica, los conceptos básicos que formarán parte del cuerpo de este trabajo. En un primer lugar, una explicación de lo que significa la paralelización de programas software y de su uso actual. A continuación, se definen los métodos de tomas de decisiones en las empresas, y la evolución en su uso y estudio. Y para finalizar, se remarcan los objetivos a cumplir con este trabajo, como a su vez, la metodología, plan de trabajo y la estructura del documento.

1.1. Computación Paralela

Típicamente un programa software es definido como una secuencia de instrucciones ejecutadas una tras otra siguiendo un orden marcado por el programador [1, 2]. Estas instrucciones son ejecutadas por un único procesador, pudiéndose computar únicamente una instrucción en cada instante de tiempo. Este es el paradigma tradicional de computación, con el cual, los primeros ordenadores comenzaron a trabajar. Sin embargo este modelo limita la capacidad de ejecución debido a la incapacidad física de crear procesadores cada vez más rápidos [3]. Es por esto por lo que surgió un nuevo modelo de computación, que aprovecha múltiples recursos computacionales como es la computación paralela.

La computación paralela [1, 4, 5] se basa en el uso del hardware, para ejecutar múltiples instrucciones al mismo tiempo. Para esto, las instrucciones del programa secuencial son divididas y ejecutadas en distintos componentes hardware. El uso de paralelismo, permite aumentar la eficiencia de los programas, debido que dividen el tiempo de cómputo entre los distintos componentes hardware.

Existe múltiples técnicas paralelas que aprovechan las tecnologías hardware existentes [6, 7]. Los métodos más comunes de paralelismo utilizados actualmente por los programadores de este tipo de aplicaciones son:

- Paralelismo en memoria compartida.
- Paralelismo en aceleradores hardware.
- Paralelismo en memoria distribuida.

En el primer caso, se aprovecha las tecnologías de *multicore* y *multithreading* para realizar la paralelización. Se crean threads que comparten parte de memoria para compartir resultados globales entre los threads. Este método es muy utilizado cuando los recursos computaciones se limitan a una única máquina de computo, lo cual limita la eficiencia que se puede obtener en la paralelización. En este modelo es usual, la utilización de threads o mediante librerías que reducen el tiempo de programación como es OpenMP [8].

El segundo modelo usa aceleradores hardware, que permiten ejecutar aplicaciones paralelas hasta miles de veces más rápido que en un procesador convencional [9]. Este tipo de computación paralela obtiene grandes beneficios debido al uso de hardware específico, pero se ve limitado tanto por el coste de estos aceleradores, reduciendo su escalabilidad, como por el limitado número de aplicaciones que pueden aprovechan el potencial de este hardware. Actualmente lo más común en este tipo de computación paralela es el uso de GPUs, cuya arquitectura está diseñada para un procesamiento paralelo. Para la ejecución en GPUs se usan lenguajes específicos de programación como son CUDA [10] o OpenCL [11].

Y para finalizar, el modelo de memoria distribuida hace posible la utilización de *clusters*, los cuales, son un gran conjunto de recursos computacionales que se comunican mediante una red de interconexión, permitiendo el envío de mensajes entre los distintos recursos. Este modelo ya no esta limitado a las capacidades de un único procesador o de los aceleradores hardware, sino que es posible utilizar todos los nodos de cómputo que estén conectados en una misma red de interconexión. Las ventajas de este modelo se basan principalmente en su escalabilidad, debido a que no limita el número de recursos de cómputo máximo. Además, este modelo coexiste con los dos modelos previos, permitiendo obtener las ventajas de todos ellos en una misma aplicación.

Este modelo es el más extendido en la computación de alto rendimiento (HPC, High Performance Computing) [12]. Donde se utiliza un cluster para ejecutar una o varías aplicaciones, de carácter científico-técnico al caracterizarse por su alto coste computacional [13, 14]. La programación paralela para grandes centros de cómputo, ha sido el foco de investigadores e ingenieros [15]. Esto ha permitido la creación de multitud de librerías y lenguajes de programación específicos para programar la mayoría de los programas que se ejecutan en estos entornos [16].

En concreto, para realizar computación paralela en un conjunto distribuido de computadores, nació un estándar llamado *Message Passing Interface* (MPI), que permite la ejecución paralela de un programa en un entorno distribuido y que se explicará con más detalle en el capítulo 2 de esta memoria.

1.2. Toma de Decisiones en Empresas

Actualmente, debido al desarrollo de las *Tecnologías de Información* y al desarrollo de la informática, las empresas son capaces de generar una gran cantidad de información, mucha más, de la que son capaces de procesar. Es por esto que se considera que vivimos en la época del *Big Data*, donde los métodos tradicionales de procesamiento de datos han quedado obsoletos, y las empresas tiene que buscar métodos novedosos para sacar partido a tanta información.

Para las empresas es fundamental el proceso de tratamiento de la información para convertirla en conocimiento. Con este conocimiento, los directivos de las empresas son capaces de tomar decisiones en base a datos y estadísticas empíricas [17], obteniendo un mayor beneficio para sus empresas. Por este motivo, cada vez más empresas toman parte del gran movimiento del *Big Data*, y aplican técnicas analíticas dentro de sus organizaciones [18, 19].

Esta tarea de procesamiento de la información para ofrecerle al decisor conocimiento que necesita para mejorar su toma de decisiones se conoce como *Business Intelligence* (BI, Inteligencia de Negocio) [20, 21]. El BI es un concepto que engloba desde la generación y recogida de la información, hasta el acceso sencillo e interactivo del conocimiento por parte de los decisores.

Los métodos existentes de procesamiento de información utilizados en el BI son muy distintos, y dependen tanto de la naturaleza de la información, como del procesamiento concreto que se le quiera aplicar. Estos métodos abarcan desde una simple media entre los datos, hasta la utilización de métodos complejos de *Inteligencia Artificial* para el procesado de información [22].

Debido a la complejidad que deriva todos los aspectos del BI, muchas empresas se ven necesitadas de herramientas específicas que las permiten lidiar con las necesidades de conocimiento que demandan. Es por esto que desde el grupo de investigación IDES de la Universidad de Cantabria, se creó una herramienta de soporte a las decisiones para empresas del entorno de la acuicultura, que se enmarca dentro del proyecto europeo Mediterranean Aquaculture Integrated Development (MedAID) [23] para la mejora de la competitividad de las empresas mediterráneas del sector de la acuicultura. Esta herramienta llamada AquiAID, tiene un coste computacional muy alto, y por lo tanto, unos tiempos de ejecución de días/semanas para problemas reales, por lo que no es utilizable en la práctica.

1.3. Objetivos

El objetivo principal de este trabajo es la mejora del rendimiento del software AquiAID [24], una herramienta de soporte a las decisiones para empresas del entorno de la acuicultura creada por el departamento de IDES de la Universidad de Cantabria. Esta mejora se va

a basar en la paralelización del código usando el paradigma de programación basada en paso de mensajes y ejecutándolo en un cluster de multiprocesadores. Para este objetivo es imprescindible tener en cuenta tres metas intermedias que permitan la finalización del objetivo:

- *Mejorar la escalabilidad del programa*: el limitante más importante de la herramienta es, la nula capacidad de dar soporte a ejemplos reales de empresas de acuicultura. Debido al gran tiempo computacional que conlleva la ejecución de la herramienta en ejemplos pequeños, es imposible su utilidad en experimentos reales. Por ello es fundamental, mejorar la escalabilidad del software, permitiendo dar soporte a ejemplos reales. Además, se busca un consumo energético mucho menor debido a la mejora del rendimiento del programa.
- *Paralelización del código*: la mejora de la escalabilidad, se puede conseguir gracias a la paralelización del código. Debido a la naturaleza de la herramienta, la paralelización del código se vuelve la forma más viable de mejorar del rendimiento y de consumo del programa.
- *Validación Experimental*: una vez se tenga la paralelización realizada, se pretende realizar experimentos reales que midan la mejora de la escalabilidad del programa. Con esto obtener métricas y mediciones de las capacidades de la implementación paralela, midiendo su grado de calidad y sus posibles usos prácticos en la industria.

1.4. Metodología y Plan de trabajo

El cumplimiento de los objetivos previos, ha sido realizado mediante las siguientes fases:

- *Análisis del código*: cualquier código ajeno es necesario una fase de análisis previo. Mediante un análisis conciso, se pretende comprobar las partes de código que limitan el rendimiento de la herramienta. Se realizará un análisis estático, y mediante herramientas automatizadas, esto permitirá evaluar las características intrínsecas del código, aportando información fundamental para las siguientes fases.
- *Diseño de la paralelización*: con el código analizado, se plantea de manera conceptual, la manera de paralelización del programa. Debido a la existencia de distintas alternativas de paralelización, es fundamental seleccionar la que aporte la mejor solución a la falta de rendimiento de la herramienta.
- *Implementación del programa paralelo*: en esta fase se lleva a cabo la implementación propiamente mencionada. Seleccionando las herramientas con las que se va a trabajar, se modifica el programa secuencial a su versión paralela, para disminuir el tiempo computacional de la herramienta.

- *Realización de experimentos*: con la versión final del programa, se realizan una serie de experimentos que ponen a prueba la implementación paralela en casos de empresas reales.
- *Validación de la paralelización*: los resultados de los experimentos permitirán validar la implementación, y obtener métricas y medidas que definan los puntos fuertes y débiles que la versión paralela. Con esto se podrán realizar mejoras en la paralelización y futuras vías de desarrollo.

1.5. Estructura del Documento

Este documento se divide en cinco capítulos contando con el presente *Capítulo 1* de *Introducción*:

- *Capítulo 2*: en este capítulo se explican conceptos básicos de acuicultura, del uso del estándar MPI y una recopilación de los trabajos relacionados.
- *Capítulo 3*: explicación detallada de las fases de análisis del código e implementación del programa paralelo.
- *Capítulo 4*: realización de experimentos junto con la comprobación y validación de los resultados.
- *Capítulo 5*: conclusiones finales del documento y posibles vías futuras para el desarrollo de la herramienta.

Capítulo 2

Background

En este capítulo se describen los conceptos más importantes para comprender el resto del documento. Se realiza una descripción del problema de la acuicultura junto, con la especificación de las características de los algoritmos utilizados y de la programación en entornos distribuidos con MPI. Finalmente se presentan algunos trabajos previos que tienen relación con el desarrollo de este proyecto.

2.1. Conceptos Básicos

2.1.1. Acuicultura

La acuicultura [25] es la actividad encargada de producir y engordar organismos acuáticos en su medio de vida. La acuicultura es utilizada para incrementar la producción de peces, moluscos, crustáceos y plantas acuáticas mediante la concentración de poblaciones, alimentación específica y la protección frente a los depredadores.

La historia de la acuicultura [26] viene ligada al crecimiento de las civilizaciones sedentarias hace unos cuatro mil años. Los primeros escritos pertenecen a las civilizaciones situadas en la actual China. Esta cultura cultivaba carpas en los estanques naturales permitiéndoles tener comida proteica prácticamente durante todo el año. A partir de aquí, múltiples culturas antiguas de Asia y Europa dejaron escritos de la utilización de la acuicultura para la producción de alimento, como son las civilizaciones de Egipto, Grecia, o Románica, etc.

Actualmente, ha descendido el uso de la pesca de captura [27, 28] debido a su gran impacto en el medio ambiente y su escasa eficiencia. Este tipo de pesca, aún manteniendo números similares de captura durante los últimos años, se ha visto superada por el crecimiento de la acuicultura. La acuicultura ha aumentado su porcentaje de aportación de especies marinas

desde hace treinta años, pasando únicamente de aportar el 10 % de las especies marinas globales, al actual 45 %. La acuicultura se presenta como una alternativa viable y sostenible, para la producción y venta de especies marinas. Pero la acuicultura no es sostenible intrínsecamente, sino que es necesario que se cumplan ciertas condiciones como:

- Asegurar un desarrollo ordenado de la acuicultura, así como una buena organización por parte de las autoridades y la industria.
- Gestionar el medio ambiente en beneficio de las generaciones futuras.
- Asegurarse de que hay suficientes alimentos disponibles para todos.
- Garantizar una distribución equitativa de los beneficios y los costes.
- Promover la creación de riqueza y empleo.

En la época actual la mayor parte de la acuicultura que se realiza en los países desarrollados, está llevada por empresas principalmente de carácter multinacional. En este aspecto, es importante recalcar la dificultad que conlleva la administración de una empresa de acuicultura [29]. Esto es debido a la multitud de sistemas y paradigmas a los que una empresa de este tipo tiene que hacer frente.

Cada empresa de acuicultura deberá seleccionar un tipo de sistema de producción. Existen cuatro sistemas principales de producción, desde entornos abiertos en el mar, hasta entornos cerrados en viveros. Cada sistema productivo tiene sus ventajas e inconvenientes, y llevará ligado el tipo de especies marinas a cultivar. Además este tipo de empresas tienen que tener en cuenta el cuidado de sus especies marinas, controlando todos los procesos biológicos imprescindible para su desarrollo. Deberán de monitorizar u obtener datos del desove, crecimiento, cuidados, residuos, enfermedades, genética, etc. Todos estos aspectos tendrán su debido impacto en la planificación empresarial, desde los gastos de alimentación, control de temperatura, oxígeno y humedad de las especies marinas, hasta la época de recogida, transporte, lugar y fechas de ventas y de nuevas siembras.

Debido a la gran cantidad de factores dependientes que una empresa de acuicultura tiene que afrontar, muchas empresas buscan aumentar sus beneficios con la ayuda de planificadores especializados. Estos planificadores son muy variados en la actualidad, pero una solución basada en inteligencia artificial [30] podría dar buenos resultados si se utilizan los métodos adecuados. En este trabajo se utilizará un planificador del proyecto AquiAID basado en una técnica conocida de inteligencia artificial llamada Particle Swarm Optimization (PSO, Optimización por enjambre de partículas).

2.1.2. Particle Swarm Optimization

La técnica de Particle Swarm Optimization [31] es un método metaheurístico basado en el comportamiento de ciertas especies de insectos, que actúan en forma de enjambre. Estos insectos, como pueden ser las abejas o las hormigas, buscan aleatoriamente alimento por las zonas cercanas. Cuando encuentran zonas interesantes con bastante alimento, se lo comunican al resto del enjambre. De forma similar es como actúan los algoritmos de tipo PSO.

El objetivo de estos algoritmos es maximizar o minimizar un valor para una función *f*, de la cual, no existe más posibilidad que probar con puntos aleatorios de esta función para hallar el mínimo o máximo global. Esta función *f*, es también conocida como una función de *fitness*, debido a que evalúa lo buena que es una posición en la función. La función de *fitness* es dependiente del problema, y por tanto puede ser tan compleja y tener tantas dimensiones $\overrightarrow{x}_1 ... \overrightarrow{x}_n$, como sea necesario para definir una solución al problema realista.

En el Algoritmo 1 se puede observar el pseudocódigo de los algoritmos de tipo PSO. El funcionamiento se describe en los siguientes párrafos, apoyándose en el pseudocódigo anterior.

Algoritmo 1 PSO

```
1: P ← Inicializar_matriz_población()
2: for i = 0 hasta máximo_iteraciones do
3:     for Cada partícula p en P do
4:         fp ← f(p)
5:         if fp es mejor quef(pBest) then
6:             pBest ← p
7:         end if
8:     end for
9:     gBest ← mejor_partícula_P()
10:    for Cada partícula p en P do
11:        v ← v + c1 * random_number * (pBest − p) + c2 * random_number * (gBest − p)
12:        p ← p + v
13:    end for
14: end for
15: return mejor_partícula_P()
```

1. En primer lugar, se generan partículas (individuos) con una posición $\overrightarrow{p}$ en el espacio de búsqueda $\overrightarrow{x}_1 ... \overrightarrow{x}_n$. Cada partícula se generará aleatoriamente para tener una distribución los más uniforme posible. Esto sería la labor de Inicializar_matriz_población() situada en la primera línea del algoritmo.

2. Luego se calcula la mejor posición tanto local (pBest) de la partícula correspondiente (líneas 5 y 6), como global (gBest, línea 9), en función de su valor en la función *f*. Para esto, se debe guardar un histórico de las mejores posiciones de cada partícula y realizar la comparación correspondiente.

3. El siguiente paso es actualizar el vector $\overrightarrow{v}$ mediante la inercia o movimiento de la partícula, y los dos valores previos calculados, pBest y gBest. Se dan distintas prioridades a la atracción de cada una de estas componentes mediante las constantes $c1$ y $c2$ en función de la implementación. Se les añade un componente aleatorio (*random_-number*) para mejorar el desplazamiento de la partícula (línea 11).

4. Una vez actualizado el vector $\overrightarrow{v}$, se puede cambiar la posición de la partícula $\overrightarrow{p}$ en función de la velocidad calculada. Con esto se obtiene la actualización de $\overrightarrow{p}$ hacia posiciones más cercanas al óptimo (línea 12).

5. Después se calcula el nuevo valor de f para cada una de las partículas y se repite el proceso de calcular la mejor posición local y global, y actualizar tanto la velocidad como la posición de cada partícula. Este proceso se repite de forma iterativa, hasta que se cumpla una determinada condición, como puede ser un número de iteraciones (línea 2) o diferencia entre el mejor global entre cada iteración.

6. Y para finalizar se retorna la partícula con mejor valor para f, y esta sería la mejor alternativa para el problema que se ha planteado (línea 15).

Este es el desarrollo de un algoritmo genérico de PSO. Se pueden encontrar variantes sobre este algoritmo como puede ser añadir fuerzas repulsivas, o separando las partículas en familias [32].

Es destacable que la dificultad computacional del algoritmo depende de un parámetro, el número de partículas. Este algoritmo se vuelve muy pesado computacionalmente al aumentar el número de partículas, implicando que no sea recomendable usar un gran número de partículas. Esta limitación provoca que las soluciones aportados por el algoritmo sean de peor calidad, debido a que el espacio de búsqueda recorrido será menor. Es por esto que los algoritmos PSO son bastante interesantes en la paralelización en entornos distribuidos, debido a que al estar buscando en un espacio que puede tener cientos de dimensiones, es necesario tener una gran cantidad de partículas para poder encontrar una solución factible.

Al tener una gran cantidad de partículas, los cálculos incrementan en medida al crecimiento del número de las partículas, con el correspondiente aumento del tiempo de ejecución. Además por la naturaleza propia del algoritmo es bastante propicio para la distribución de sus cálculos, puesto que por lo general las partículas son independientes entre sí, y permite realizar la mayoría de cálculos en computadores independientes.

2.2. Message Passing Interface

Message Passing Interface o también conocido por sus siglas MPI [33, 34, 35], es una especificación de programación paralela, pensada principalmente para el envió de mensajes entre computadores de memoria distribuida. El modelo específico de programación utilizado con MPI es el de inicializar un conjunto de procesos independientes que pueden ejecutar en el mismo o distinto nodo de cómputo, y que se comunican y/o sincronización mediante llamadas explícitas de comunicación. Estas llamadas, por lo general, realizan el envío de información entre la memoria de un proceso hacia el resto.

MPI no es ni un lenguaje de programación, ni una implementación, es un estándar que permite múltiples implementaciones. Muchos lenguajes implementan este estándar, desde los más clásicos como C o Fortan, o lenguajes más modernos y más abstractos como Python o Java. MPI además de definir operaciones de paso de mensaje clásicas, extiende su estándar añadiendo operaciones de acceso a memoria colectiva, creación de procesos dinámicos y operaciones de entrada y salida remotas.

Para este trabajo no ha sido necesario usar todas las funcionalidades que proporciona el estándar. Por tanto, es interesante destacar las partes de la especificación que se han utilizado de forma significativa a la realización del trabajo, en concreto se mencionará en detalle los siguientes aspectos del estándar:

- Inicialización de procesos.
- Comunicadores y grupos de procesos.
- Comunicación punto a punto y llamadas colectivas.

El trabajo realizado no se ha implementado con uno de los lenguajes básicos que define el estándar como C o Fortran. Sino que se parte de una implementación secuencial ya realizada por el grupo IDES de la Universidad de Cantabria. Para que la paralelización sea útil se ha definido como requisito no funcional que se mantenga el lenguaje de programación en Python [36]. En concreto se utilizó una implementación de MPI en este lenguaje llamada Mpi4py [37]. Es por esto que en las siguientes secciones se describe el estándar con ejemplos de código definidos en la implementación en Python.

2.2.1. Inicialización de procesos

Todo programa MPI deberá tener una primera etapa de inicialización y otra de finalización. Existen dos funciones que define el estándar para empezar la comunicación entre procesos, que son `MPI_Init` y `MPI_Finalize`.

En primer lugar,`MPI_Init` es la función que permite a todos los procesos comunicarse entre ellos, inicializando todas las estructuras y funciones necesarias para el intercambio de información sin errores, por parte de los procesos. Y finalizando siempre las comunicaciones la rutina `MPI_Finalize`, elimina todos los datos asociado al estado de las comunicaciones concluyendo así los intercambios de información entre procesos.

Entre estas dos funciones se pueden realizar cualquiera de las comunicaciones entre procesos permitidas por el estándar. En la implementación de Mpi4py, estas funciones son directamente ocultadas al programador y son automáticamente utilizadas al realizar la importación de la librería. Por tanto, al programador se le permite usar en cualquier momento funciones de comunicación.

2.2.2. Comunicación punto a punto y llamadas colectivas

Una vez inicializados los procesos estos se comunican para el intercambio de información. En el estándar se definen dos formas generales de enviar información entre los procesos, comunicación punto a punto y comunicación colectiva.

En primer lugar definir que son las comunicaciones punto a punto, las cuales son las funciones de comunicación más básicas que existen en MPI. Su funcionamiento es mediante las llamadas *comm.send(data, dst, tag)* y *comm.recv(src, tag)*, las cuales permiten a un proceso enviar los datos y a otro distinto recibirlos respectivamente.

Estas funciones permiten tanto enviar como recibir estos datos entre dos procesos. En concreto en la función *comm.send*, mediante su parámetro *data* se define la información que se envía al proceso con el rank expuesto en el parámetro *dst*. La función *comm.recv* devuelve los datos enviados por el proceso definido en *src*. El parámetro *tag*, permite a un proceso receptor diferenciar entre qué tipo de mensaje le llega.

Un ejemplo de este tipo de comunicación entre dos procesos es de productor-consumidor, donde un proceso envía datos a otro a un ritmo indeterminado. Este ejemplo se puede observar en la Función 1.

Función 1 Productor-Consumidor

```
from mpi4py import MPI
import time
import random

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

for i in range(10):
    if rank == 0: # división entre productor-consumidor
        # esperar envío entre 0 y 9 segundos.
        time.sleep(random.ranint(0, 10))
        data = random.ranint(0, 100) # generación de datos
        comm.send(data=data, recv=1, tag=0) # envío

    else:
        data = comm.recv(src=1, tag=0) # recepción
        # Procesar datos
        print("Datos recibidos en la iteración " + srt(i)": " + str(data))
```

Estas serían las funciones básicas que se encuentran en MPI para el envío de información. Pero estas funciones están limitadas a la comunicación entre dos únicos procesos. Esto limita la eficiencia de los programas debido a que la compartición de datos en la mayoría de los programas, no se limita a dos únicos procesos, sino a todos los que participan en el cómputo. El envío secuencial de datos mediante las llamadas básicas resulta, además de tedioso de programar y difícil de gestionar, es poco eficiente. Esto es así por dos motivos: por un lado, cada función de comunicación implica una sobrecarga fija, independientemente del número de procesos y del tamaño del mensaje. Por ello, cuantas más llamadas más sobrecarga se produce. Y por otro lado, las funciones colectivas implementan modelos de comunicación específicos (por ejemplo basados en árboles) que optimizan la comunicación frente a punto a punto.

Por estos motivos se crearon las llamadas colectivas. Estas llamadas permiten el envío de información entre varios procesos al mismo tiempo de manera eficiente. Las implementaciones de MPI realizan un encadenamiento de comunicaciones entre los procesos que reducen el número de mensajes que atraviesa la red de comunicaciones, mejorando en gran medida el tiempo de comunicación.

Existen multitud de llamadas colectivas en el estándar de MPI, pero como se ha comentado en secciones anteriores, sólo se describirán las funciones utilizadas en este trabajo. La primera función colectiva a explicar es *comm.Bcast(buff, root)*. Esta función realiza el envío de datos, especificado en el parámetro *buff*, enviado por un proceso *root*, al resto de procesos implicados en la comunicación. La utilidad de esta función permite enviar datos calculados por un proceso maestro, y compartirla con el resto para tener la misma variable actualizada.

La segunda función es *comm.Allreduce(sendobj, op)*. Allreduce permite enviar información definida en *sendobj* por todos los procesos y recibida por todos ellos a su vez. La gran ventaja de esta función es que mediante el parámetro *op*, es posible definir una operación que será realizada sobre los datos enviados, y la función devuelve el resultado de esta operación. Existen operaciones básicas ya definidas por el estándar MPI, pero el programador tiene la opción de definir sus propias funciones.

Un ejemplo de uso de esta función se muestra en la Función 2. En este código todos los procesos computan un dato en la variable *data*. Con todos los datos de los procesos se pretende calcular su suma. Es por esto que, en la línea 8, todos los procesos realizan la llamada colectiva `Allreduce`, con la operación de suma y la variable *data* como parámetro compartido. En la variable *result* de cada proceso se guarda el valor de la suma total de todos los procesos.

Las siguientes dos funciones, son complementarias una con la otra:

- comm.Gatherv([data, count, disp, dataType], recvbuf, root).
- comm.Scatterv([data, count, disp, dataType], recvbuf, root).

Función 2 Ejemplo Allreduce

```
1  from mpi4py import MPI
2  import random
3
4  comm = MPI.COMM_WORLD
5
6  data = random.ranint(0, 100) #dato a enviar
7
8  result = comm.Allreduce(data, MPI.SUM) #Operación suma de todos los procesos
9
10 print("Resultado de la suma: " + str(result))
```

La función Gatherv, se encarga de recoger información de todos los procesos, mediante la variable *data* y su tipo de dato *dataType*, y serán recibidos por el proceso *root* mediante la variable *recvbuf*. Además, existen las variables *count* y *disp*, especificadas como listas de enteros, que permiten diferenciar cuantos datos envía cada proceso y con qué desplazamiento son colocados los datos en *recvbuf*.

La función Scatterv, se comporta de manera similar, pero en este caso el proceso *root*, reparte los datos localizados en *data*, al resto de procesos. Siendo *count* y *disp*, respectivamente, el número de datos que recibe cada proceso y el desplazamiento desde el cual se envían los datos de *sendbuf*.

En la Función 3, se puede observar un ejemplo sencillo de funcionamiento de estas llamadas. En primer lugar, se envían al proceso con rank cero, unos datos calculados por todos los procesos, y seguidamente el proceso cero realiza una operación con ellos. Después de la operación, el proceso cero devuelve a cada uno de los procesos los datos que tenían originalmente modificados.

Función 3 Ejemplo Gatherv y Scatterv

```
from mpi4py import MPI
import random
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size() #número total de procesos

# Primera fase envío al proceso cero
data = random.ranint(0, 100) # dato a enviar
count = [] # datos enviados por cada proceso
disp = [] # desplazamiento de los datos en la recepción
root = 0 # proceso cero es el root

for i in range(size):
    count.append(1) # cada proceso envía un dato
    disp.append(i) # los datos serán colocados en orden de procesos

if rank == 0:
    recvbuf = np.empyt(size) # buffer de recepción

# envío de resultados
comm.Gatterv[data, count, disp, MPI.INT], recvbuf, root)

# El proceso cero realiza una operación con los datos
if rank == 0:
    for i in range(size):
        recvbuf[i] += 1
    sendbuf = recvbuf

recvbuf = np.empyt(1) # buffer de recepción

# Envío del proceso cero al resto de procesos
comm.Scatterv([sendbuf, count, disp, MPI.INT],recvbuf, root )
```

De la función Gatherv, existe una versión llamada Allgatherv, que se diferencia de la anterior en que no existe un proceso root, sino que todos reciben la información repartida.

2.2.3. Comunicadores y grupos de procesos

Las funciones colectivas previamente explicadas tienen la características de ser síncronas. Es decir que todos los procesos dentro de un comunicador deben ejecutarlas. Un *comunicador* es descrito por el estándar como el conjunto de procesos que pueden comunicarse entre sí, junto con todas las estructuras necesarias para realizar esta comunicación.

Existe un comunicador por defecto, donde todos los procesos pueden comunicarse punto a punto y colectivamente sin ninguna limitación, a este comunicador se le conoce como *MPI_COMM_WORLD*. Cada comunicador esta conformado por un *grupo de procesos*, los cuales son identificados unívocamente mediante un identificador conocido como *rank*. Se puede conocer tanto el tamaño del grupo de procesos como el rank de cada proceso mediante las llamadas que se observan en la siguiente Función 4:

Función 4 Consulta del size y rank de un proceso

```
1 comm = MPI.COMM_WORLD # Llamada al comunicador global o por defecto
2 size = comm.Get_size() # Tamaño del comunicador
3 rank = comm.Get_rank() # ID del proceso
```

Con la descripción previa se tiene un marco imprescindible para entender como funcionan las comunicaciones en MPI, debido a que si un proceso quiere enviar información a otro, ambos procesos deberán de usar el mismo comunicador y conocer el rank de cada uno.

El comunicador por defecto, MPI_COMM_WORLD, permite comunicar punto a punto cualquier par de procesos y realizar llamadas colectivas entre todos los procesos de la aplicación. Pero si se desea realizar una comunicación entre un grupo específico de procesos, se debe crear un nuevo comunicador e incluir en él los procesos que se quieran comunicar.

Es por esto que MPI proporciona una serie de funciones encargadas de crear grupos de procesos y comunicadores que permite realizar comunicaciones colectivas entre grupos aislados de procesos. Como suele ser habitual en MPI existen varias funciones para realizar las tareas previamente descritas, pero por simplicidad y exactitud, se explicarán solo las que se utilicen en este trabajo concreto.

Para generar nuevos grupos es interesante utilizar el método *comm.group.Incl(Sequence[int])*. Esta función creará un nuevo grupo de procesos partiendo del grupo de un comunicador previo ya definido como puede ser *comm.group*. Este método recibe una secuencia de enteros que representan los ranks del comunicador *comm*, que identifican a los procesos que generarán el nuevo grupo. Además de generar el grupo, se ordena los ranks del nuevo grupo, en función del orden de la secuencia. Esta funcionalidad es importante ya que permite seleccionar cualquier subconjunto de procesos de un comunicador y ordenar sus nuevos ranks en función de las necesidades del programador.

Con el método previo se podría crear grupos de todo tipo y cualquier forma, pero estos grupos van a necesitar un comunicador para poderse comunicar entre ellos. Para esto se utilizaría la función*comm.Create_group(newGroup)*. En este caso, la función recibe el nuevo grupo creado previamente *newGroup* y devuelve el nuevo comunicador para el nuevo grupo.

En la Función 5, se puede observar un ejemplo claro de como se realiza esta creación de grupos y comunicadores juntando todas las funciones previas.

Función 5 Creación de Grupos y Comunicadores

```
1 comm = MPI.COMM_WORLD # Llamada al comunicador global o por defecto
2 listProcess = [2, 3, 4] # lista de procesos que generan el nuevo grupo
3 newGroup = comm.group.Incl(listProcess) # generación del nuevo grupo
4 comNew = comm.Create_group(newGroup) # generación del nuevo comunicador
```

Mediante las funciones expuestas previamente y los ejemplos de código, es posible crear grupos de procesos que se pueden comunicar sin interferencias con otros procesos de manera eficiente.

Con esto se pondría fin a la explicación completa de MPI en el lenguaje de Python. Las definiciones y aclaraciones previas sientan el contexto para comprender los Capítulos 3 y 4

2.3. Trabajos Relacionados

2.3.1. Planificación en Acuicultura

Actualmente existen multitud de estudios relacionados con la planificación de empresas dedicadas a la acuicultura. Pero por lo general, están dedicados a un ámbito específico de aplicación, como puede ser el gasto energético [**?**], la planificación de flotas de recogida [38] o incluso de la cría de una única especie [39]. Los sistemas de apoyo a decisiones de las empresas de acuicultura propuestos por la academia se encuentran actualmente desfasados [40, 41] , o son excesivamente simples para un uso práctico [42].

Es por este desfase de artículos, y por el crecimiento de la acuicultura[43] que nuevos autores han propuesto fórmulas actualizadas para este tipo de empresas [44, 45]. Cabe destacar el trabajo del grupo IDES de la Universidad de Cantabria donde durante los últimos diez años han realizado múltiples trabajos en el campo de la acuicultura [46, 47, 48]. Debido a su conocimiento en la industria de la acuicultura, han aportado un software de gestión de este tipo de empresas basadas en el algoritmo de PSO [49, 50], en los cuales se basa este trabajo.

2.3.2. Paralelización Particle Swarm Optimization

A parte del sector de la acuicultura, los algoritmos PSO son utilizados en multitud de casos tanto prácticos, como académicos, debido a su versatilidad [51, 52]. Estos algoritmos pueden llegar a tener tiempos de ejecución bastante largos dependiendo de la complejidad del problema, debido a su naturaleza iterativa y de búsqueda espacial.

Este tipo de algoritmos es un buen candidato a la hora de realizar una paralelización. Esto es debido a que en líneas generales, las partículas suelen ser componentes independientes o con muy pocas dependencias entre ellas, por tanto se puede realizar una paralelización de los datos de grano fino. Debido a este componente, es posible realizar de múltiples maneras, una paralelización del algoritmo de PSO.

Existen bastantes implementaciones de PSO actualmente diseñadas e implementadas que permiten aprovechar el paralelismo de los sistemas de cómputo para obtener mejores resultados. Un ejemplo claro es MRPSO [53], que utilizan el paradigma de MapReduce [54] para realizar dicha paralelización. Aprovechan la independencia de las partículas para dividirlas entre los procesos y realizar las operaciones típicas de Map, Shuffle y Reduce.

También se han realizado paralelizaciones mediante el uso de GPUs, como en el proyecto GPU-PSO [55], que realizan una implementación en GPUs tanto local, como distribuida del algoritmo permitiendo ejecutar en múltiples GPUs al mismo tiempo. En referencia a las GPUs, se han llevado a cabo experimentos que comparan el rendimiento de este algoritmo paralelizado entre una CPU y una GPU [56]. La GPU en este escenario mediante una implementación en CUDA [57], la GPU alcanza rendimientos de hasta 17 veces más rápida que la versión en CPU.

Capítulo 3

Implementación de la Paralelización

En este capítulo se explican las contribuciones de este trabajo en el software de planificación para empresas del sector de la acuicultura desarrollado por el proyecto de AquiAID del grupo IDES. Para esta finalidad, se describirán en las siguientes secciones los cambios que se han realizado a la estructura del programa secuencial. En concreto, se detalla en las primeras tres secciones el software secuencial a paralelizar, la etapa inicial de profiling y la paralelización del acceso a la base de datos. Y en la última sección, se describe la paralelización del algoritmo PSO mediante el paradigma de memoria distribuida utilizando la librería de Mpi4py.

3.1. Programa Secuencial

El programa AquiAID, es un software desarrollado por el grupo de investigación IDES, dedicado a la optimización en la planificación de las empresas de acuicultura. El objetivo fundamental del programa es planificar el cultivo de peces, teniendo en cuenta todos los aspecto a tener en cuenta, como son los factores ambientales, biológicos y económicos. Para realizar esta tarea tan compleja de planificación, se creó este software escrito en Python, con una implementación del algoritmo PSO. Esta implementación, permite evaluar varias formas de planificación para una empresa concreta y mejorarlas con el paso de las iteraciones del algoritmo.

Una de las características que limita a este software es el tiempo de respuesta del algoritmo PSO. Cuando se ejecuta la aplicación con entornos realista de empresas de acuicultura, el tiempo de ejecución es excesivamente alto para poderse aplicar en la práctica. Por consiguiente, era necesario realizar un cambio en el programa que permitiera reducir su tiempo de respuesta, a valores viables para su aplicación práctica. Y para esto lo primero que se realizó fue un análisis del código original explicado a continuación.

Para iniciar la descripción del programa, es importante mencionar que los datos que utiliza el programa son recabados de una base de datos. La base de datos utilizada se basa en el gestor relacional de MySQL [58], uno de los más utilizados en todo el mundo. Los datos que utiliza el programa, son datos que cualquier empresa de acuicultura tiene a su disposición, como pueden ser características de sus peces, piensos, temperatura del mar, etc.

El programa secuencial recoge todos los datos mediante varias llamadas a la base de datos de MySQL. Con estos datos el programa inicializa todas las estructuras necesarias para comenzar con la fase de planificación con el algoritmo PSO.

En la Figura 3.1 se observa todos los pasos que sigue el algoritmo secuencial PSO implementado en AquiAID. Este algoritmo sigue la estructura genérica del Algoritmo 1. Pero se le añade ciertos cálculos para mejorar la eficacia de las búsquedas, aunque también, se añade a las dimensiones de las partículas cierta semántica que permitirá una mejor planificación.

En primer lugar el algoritmo genera la matriz de partículas para iniciar la búsqueda. En esta implementación concreta cada una de las partículas está forma por un número de jaulas. Las jaulas, como su propio nombre indica, representan las jaulas de peces donde se realiza un cultivo por parte de una empresa. Como se observa en la Figura 3.2, una partícula es un conjunto finito de jaulas, y las dimensiones de la partícula vienen ligadas al número de jaulas.

Esta semántica aportada a las dimensiones de las partículas, permite planificar de acorde a las necesidades de cada empresa, pudiendo variar el número de jaulas en los parámetros del programa.

Una vez inicializado las partículas y las jaulas en la matriz inicial, se continua con el siguiente paso de cálculo de la función de fitness y restricciones. En el algoritmo genérico solo se especifica la función de fitness como valor a calcular, pero en la implementación secuencial existe además el cálculos de las restricciones.

Estas restricciones son validadas para cada una de las partículas, y permiten determinar si una partícula cumple con una serie de requerimientos mínimos, como para ser considerada entre las mejores. Esto mejora en gran medida la capacidad de búsqueda del algoritmo, debido a que mediante esta comprobación, se eliminan partículas que podrían tener un mejor fitness, pero que son imposibles de realizar a nivel práctico. Esto mejora la búsqueda debido a que las posiciones de las partículas se actualizan únicamente en función de las partículas que cumplen las restricciones.

Inicializar Partículas y Jaulas

Cálculo Fitness y Restricciones

Cálculo del Mínimo Local y Global

Actualizar Velocidad y Posición

Número máximo de Iteraciones o Umbral de mejora.

Figura 3.1: Algoritmo Secuencial PSO implementado en AquiAID

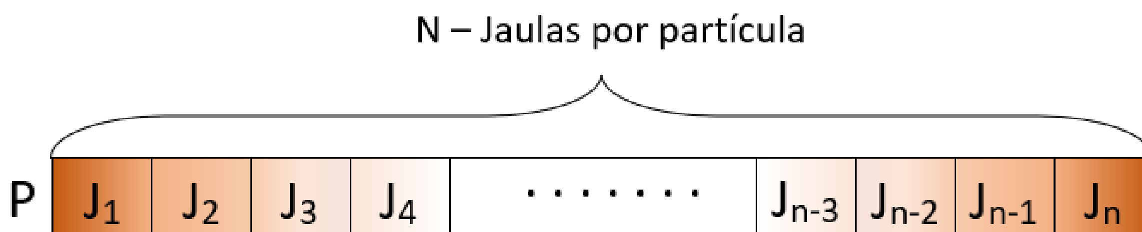


Figura 3.2: Representación de jaulas en una partícula

Una vez calculados los valores del fitness y de las restricciones, se calculará la mejor partícula local y global. Para el cálculo de la local, es necesario guardar en una matriz de las mismas dimensiones que la inicial, la mejor posición de cada partícula. Además también se guardará el mejor valor fitness de cada partícula. Una vez calculado la mejor posición de cada partícula, se calcula la mejor posición global y se guarda la posición y fitness de dicha partícula. Es necesario comentar que aquellas partículas que no cumplan con las restricciones, no podrán ser seleccionadas tanto para el cálculo local, como mucho menos la global.

Cuando se finaliza la fase anterior, se actualiza la velocidad y posición de las partículas como se menciona en el Algoritmo 1. Una vez actualizado la posición se decide si se finaliza el algoritmo o se continua con una nueva iteración.

Para esta decisión se toman en cuenta dos criterios, el primero es, si se ha alcanzado el número máximo de iteraciones que se permite en la búsqueda de nuevas soluciones. Y el segundo criterio es, si la mejoría entre el fitness de la mejor partícula entre las dos últimas iteraciones no supera un cierto umbral. Si esta última mejoría es no supera el umbral, o se llega al máximo de iteraciones el algoritmo finalizaría, si esto no sucede se volvería a la fase de cálculo de fitness y restricciones como se observa en la Figura 3.1.

El algoritmo PSO básico acabaría con todas las explicaciones previas, pero en la implementación que se está paralelizando, existe una parte final en la cual se devuelven todas las posiciones de las partículas, junto con sus puntuaciones. Esto se realiza para comprobar por parte del usuario, cómo es el estado final de las partículas, y además, se realizan ciertos cálculos simples con el conjunto entero de partículas que requieren muy poco tiempo de cálculo.

Con esta descripción del programa secuencial, se observa que la complejidad de programa reside en el algoritmo PSO. El algoritmo implementado es muy adecuado para la paralelización debido al grado fino que permite, y la relativa independencia de los cálculos realizados en las estructuras de datos. Con todo esto en mente, en las siguientes secciones se explica las fases de la paralelización del programa, iniciando con la fase de profiling.

3.2. Profiling

Antes de realizar una paralelización de un programa, es preciso realizar una fase de profiling [59]. Esta fase consiste en realizar un análisis del programa que determina cuales son las funciones o métodos que gastan mayor cantidad de tiempo en su ejecución.

Para ejecutar esta tarea se depende de las herramientas de profiling que el lenguajes aporta a los usuarios. En Python existe un profiling por defecto que permite realizar un análisis completo de la ejecución del programa que solventa las necesidades para este trabajo [60].

ncalls	tottime	percall	cumtime	percal	filename:lineno (function)
1	0.000	0.000	812.587	812.587	ACUIPSO.py:209 (optPSO)
1	0.019	0.019	812.587	812.587	pyswarmC.py:36 (pso)
3720	0.082	0.000	812.239	0.218	pyswarmC.py:21 (_obj_wrapper)
3720	0.706	0.000	812.156	0.218	ACUIPSO.py:125 (valorRelativo)
3723	6.344	0.002	676.399	0.182	FeedingStrategy.py:198 (planificar)
285007	6.127	0.000	561.085	0.002	FeedingStrategy.py:57 (alimentacionDia)

Cuadro 3.1: Resultados del profiling

En el Cuadro 3.1, se puede observar un resumen del profiling del programa de AquiAID secuencial. En este experimento se han utilizado un total de ciento veinte partículas junto con diez jaulas por cada una de ellas. Al ser este algoritmo de tipo iterativo, el tiempo aumentará de manera lineal con el número de partículas. En el cuadro existen varias columnas con distintos significados:

- *ncalls*: número de llamadas a la función.
- *tottime*: tiempo total consumido en la propia función.
- *percall*: tiempo consumido en cada llamada de a la función.
- *cumtime*: tiempo total de la función sumando las llamadas a funciones de dentro de ella.
- *percall*: tiempo consumido en cada llamada a la función sumando las llamadas a funciones de dentro de ella.
- *filename:lineno(function)*: localización del fichero donde se encuentra la función, junto con la línea donde se define y su nombre.

En esta cuadro se observan varias funciones del programa en las cuales, el tiempo invertido por ellas es bastante amplio en comparación con el tiempo total del programa. Las tres primeras funciones son llamadas previas a la función que verdaderamente gasta la mayoría del tiempo, la cual es *valorRelativo*. Esto se puede deducir gracias a observar las llamadas que realizan los métodos en el programa.

La función valorRelativo es la encargada de realizar la valoración de las partículas del algoritmo PSO, es decir, la función fitness. Esta función, en cada una de sus llamadas no gasta una gran cantidad de tiempo, solamente 0.218 segundos, pero es llamada 3720 veces, suponiendo un tiempo total consumido en valorRelativo de 812.156 segundos. Siendo 815.189 segundo el tiempo total del programa, el tiempo consumido en la función valorRelativo supone el 99.6 % del tiempo de ejecución. Esta función que calcula la puntuación o valoración de la partícula, al gastar tanto tiempo, aporta la certeza que el algoritmo PSO es el limitante claro del programa.

Por tanto, el paso fundamental para mejorar la eficiencia del programa, es reducir la carga de esta función. Para esto, la opción más efectiva, es realizar la paralelización del algoritmo PSO para repartir el trabajo de esta función. Invertir tiempo en mejorar la programación de esta función no se ve como una solución viable, debido al poco tiempo individual que gasta cada llamada, en este caso el problema está en el número abundante de llamadas, y no tanto en el tiempo consumido en cada una de ellas.

El resto de funciones en el Cuadro 3.1, son funciones que permiten el cálculo de la puntuación de la partícula que están embebidas dentro de la llamada valorRelativo.

Con los pasos anteriores se habría realizado la fase del profiling, fundamental para conocer el cuello de botella del programa. Aún con esta fase, es clave realizar un análisis del código estático para observar partes del código ocultas por el profiling que deben ser revisadas. Una de estas partes detectadas fue el acceso a la base de datos, en la cual se observó un problema cuando se diseñó la aplicación paralela.

La paralelización del acceso a la base de datos y del algoritmo PSO, se describen en detalle en las siguientes secciones.

3.3. Paralelización del acceso a la base de datos

El primer paso que se realizó fue la paralelización del acceso a la base de datos. Los datos utilizados por esta aplicación no necesitan una cantidad almacenamiento excesiva debido a que su tamaño actual no alcanza el Megabyte de información. Debido al tamaño tan limitado de los datos en el estado original de la aplicación, se realiza una única llamada a la base de datos. En esta llamada se recogen todos los datos y son guardados en memoria para ser utilizados en cualquier momento

La forma de recogida de los datos del programa secuencial se considera ineficiente si se aplica en un entorno paralelo. Si se realiza un número elevado de acceso simultáneos por parte de todos los procesos, el gestor puede tener problemas para atender todas al mismo tiempo. Esto ralentizaría a varios procesos, aumentando su tiempo de inactividad por encima de otros. Por tanto, debido a este problema con el gestor de la base de datos, se busca una solución de compromiso que permita a los procesos estar el menor tiempo parados sin saturar el acceso a la base de datos.

La solución planteada es realizar una llamada única a la base de datos por parte de un proceso concreto, y distribuir estos datos al resto de procesos, mediante funciones de comunicación. Con esta propuesta se consiguen dos ventajas, la primera es reducir al mínimo el acceso a la base de datos realizando únicamente una llamada, y la segunda es el envío distribuido de los datos entre los procesadores de manera eficiente. Gracias al uso de MPI para distribuir los datos, los procesos estarán muy poco tiempo parados esperando a recibirlos.

Con la solución previa descrita se obtiene que todos los procesos tienen los datos necesarios para poder ejecutar el resto de programa. Siendo este paso, el previo a poder empezar con la paralelización del algoritmo de inteligencia artificial basado en PSO.

3.4. Paralelización PSO

En esta sección se describe en profundidad cómo se han aplicado técnicas de paralelización al algoritmo PSO implementado por el proyecto AquiAID.

3.4.1. Tipo de paralelización

En un primer lugar, hay que analizar las estructuras de datos definidas en el programa y seleccionar las más adecuadas para realizar la partición de dominio. Es decir, qué estructuras de datos se repartirán entre los procesos, y determinarán la carga de trabajo mínima a realizar. Este análisis permitirá conocer la escalabilidad del algoritmo y la complejidad de la paralelización. En la implementación de PSO en AquiAID, existen dos estructuras diferenciadas, las partículas y las jaulas.

Las partículas son elementos típicos de los algoritmos de PSO como se define en la Sección 2.1.2. Suelen ser los elementos más comunes dónde se realiza la paralelización debido al coste computacional de su función de fitness. Su paralelización consistirá en crear procesos MPI independientes que reciben un número de partículas determinado y se encargará de calcular las puntuaciones de cada una. Después de obtener los resultados de las partículas, se pasará a una fase de comunicación donde se comparten los resultados parciales de cada proceso, y se calculan las respectivas medidas para continuar con una nueva iteración del algoritmo.

Cabe destacar que la compartición de los resultados es una dependencia entre las partículas que puede dificultar la paralelización. Aún con esta dificultad, MPI tiene las funcionalidades necesarias para recabar los resultados de manera distribuida. Esta distribución permite poder paralelizar completamente el algoritmo y obtener un planificador distribuido para una empresa de acuicultura.

Esta paralelización en base a las partículas es sencilla y rápida, permitiendo obtener resultados bastante buenos en poco tiempo de desarrollo. Pero limita la escalabilidad del algoritmo, ya que el grano de paralelismo implica en este caso una partícula por proceso. Por lo tanto, no se pueden usar más procesadores que partículas y el número de partículas suele ser de pocos centenares.

Es por esto que se buscó alguna otra estructura que permitiera una escalabilidad mayor. En la implementación del algoritmo hay otra estructura independiente en los cálculos que permite una paralelización de grano mucho más fino y que mejora en gran medido el rendimiento, y sobre todo de la escalabilidad del programa. En concreto esta estructura son las jaulas.

La definición de jaula en el algoritmo, recae a la cuestión de si el programa permite realizar esta paralelización en función de las jaulas. Para poder obtener una respuesta clara a esta pregunta, hay que analizar tres aspectos concretos:

- División en jaulas de las partículas.
- Función de fitness.
- Compartir los resultados.

En primera instancia, es importante conocer si es posible dividir las partículas en jaulas y poder repartir estas últimas entre los procesos. Esto es resuelto fácilmente debido a que entre las jaulas de una partícula, no existe ninguna dependencia que las obligue a ser indivisibles. Debido a esto es posible repartir las jaulas entre los distintos procesos.

Esta nula dependencia entre las jaulas, es en gran medida debido al segundo punto, la función de fitness. Los algoritmos PSO, calculan la función de fitness para una partícula, y esto no es distinto en la aplicación de AquiAID. Pero en esta implementación concreta la función de fitness de una partícula se divide en el cálculo de valoraciones para las jaulas de manera individual. Gracias a este cálculo individual de las jaulas es posible dividir el cálculo del fitness de una partícula entre los distintos procesos.

Como último punto, es preciso que el cálculo distribuido de las puntuaciones de las jaulas pueda ser compartido entre los procesos. Esto permite calcular el fitness de las partículas y las métricas finales de las iteraciones del algoritmo. Como en el caso de las partículas, mediante MPI es posible compartir los datos de las jaulas entre los procesos y realizar operaciones distribuidas con ellas.

Paralelizar en función de las jaulas se vuelve mucho más complejo, debido a que el algoritmo PSO define las operaciones en función de las partículas, y no a subconjuntos de las dimensiones de las partículas, como son las jaulas. Esto vuelve muy complejo la paralelización debido a que hay que refactorizar gran parte del código referente al algoritmo para que funcione mediante jaulas.

Además, se añaden operaciones de comunicación para enviar tanto las jaulas como los cálculos respectivos a éstas, para poder computar los valores de las partículas. Aún con todo lo anterior, y teniendo que cambiar gran parte del código del algoritmo y volverle mucho más complejo, la escalabilidad que permite la paralelización por jaulas mejorará en gran medida la eficiencia del algoritmo.

Esto último, junto con lo mencionado en la Sección 3.2, en la cuál se identifica que el cálculo del fitness de una partícula es la parte computacionalmente más limitante. Paralelizar mediante partículas permite dividir el tiempo de cómputo de este fitness entre el número total de partículas. Pero al dividirse la función de fitness en el cálculo por jaulas, se divide el tiempo de cómputo del fitness entre el número de jaulas, el cual siempre es superior al número de partículas.

Teniendo estas alternativas y debido a que el objetivo principal es conseguir la mayor escalabilidad del programa, la decisión es paralelizar en función de las jaula. Para esto es necesario dividir las jaulas entre los procesos, y crear todas las estructuras necesarias para realizar la paralelización del algoritmo PSO, como se describirá en las siguientes secciones.

3.4.2. Puntuación y restricciones de jaulas

Como se menciona en la sección anterior, una cuestión importante a modificar en el código es la capacidad de generar puntuaciones y comprobar las restricciones. En el algoritmo de AquiAID se computan dos datos por partícula, su puntuación o fitness y si es factible. Estos datos permiten conocer qué partículas son las más aptas, y así, el resto de partículas tendrán tendencia a acercarse a esas soluciones.

Estos cálculos son realizados mediante las dimensiones de una partícula. Esto con el modelo de paralelización de jaulas no se podría realizar y habría que modificar las funciones encargadas de esta tarea. Cuando se analizan al detalle estas funciones, los resultados se calculan recorriendo cada una de las jaulas de la partícula. Esto permite realizar la paralelización en función de las jaulas, ya que estos cálculos son independientes entre las jaulas.

Por tanto, habrá que modificar estas funciones para que calculen los valores de puntuación y restricciones de una única jaula. Para esto se eliminan las estructuras de datos que reunían y calculaban valores para varias jaulas, añadiéndolas a nuevas funciones que toman estos valores de las jaulas y generen los valores para una partícula en concreto.

Con estos cambios en las funciones de fitness y de restricción se podría empezar con la paralelización completa del algoritmo PSO.

3.4.3. División del cómputo y comunicación de resultados

Como se puede observar en el Algoritmo 1, la primera etapa definida es la generación de partículas. En la implementación de AquiAID, esta creación se realiza mediante la inicialización de una matriz. Cada fila de esta matriz representa una partícula concreta, y en las columnas están presentes las dimensiones de cada partícula. Además como se describe en la Sección 3.4.1, las partículas están representadas por jaulas, y por tanto, las dimensiones de cada partícula son la representación de las jaulas.

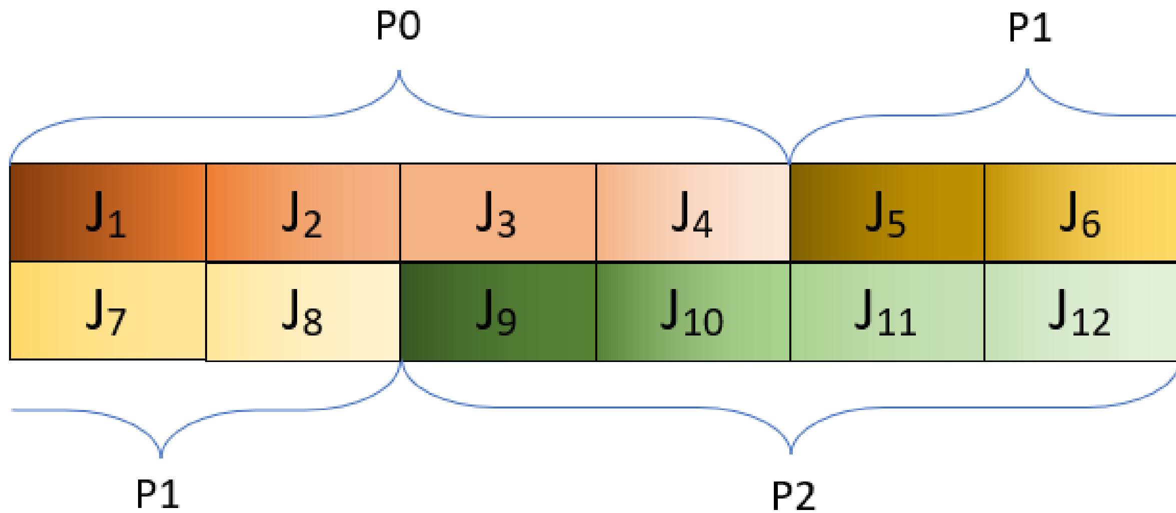


Figura 3.3: Ejemplo de reparto de las Jaulas

Esta matriz inicial tiene que ser dividida en el conjunto de jaulas, y repartida entre todos los procesos. Con esto, se garantiza un equilibrio de carga de trabajo entre los procesos, debido a su vez, que la carga computacional de las jaulas es regular, es decir, que todas realizan el mismo número de operaciones. Este reparto es de tipo estático, una vez decidido qué jaulas computan cada uno de los procesos, no se cambia este reparto a lo largo de la ejecución. Por tanto, es fundamental que el reparto de las jaulas sea lo más justo posible, ya que en este tipo de algoritmos la carga computacional depende fundamentalmente de la cantidad de datos a computar, y no de sus valores.

Para un reparto justo, la forma más habitual y sencilla de realizarlo es calcular el número total de jaulas y dividirlo entre el número total de procesos. En la Figura 3.3, se puede observar un ejemplo básico, de un reparto de dos partículas, o dos filas de la matriz. En cada una de las partículas existen seis jaulas distintas, teniendo un total a repartir de doce jaulas. En este ejemplo existen cuatro procesos que quieren computar estas jaulas. Para ser equitativo se realiza la división, y se tendría que cada proceso computará cuatro jaulas, como se observa en la figura.

Como se observa en la Figura 3.3, en el reparto los procesos recogen las jaulas en el orden en el cual están situadas en la matriz, es decir, el primero proceso realiza el cómputo de las primeras cuatro jaulas, luego el segundo proceso procesará las siguientes cuatro jaulas y así sucesivamente hasta completar la matriz completa. Mediante esta repartición se obtiene una ventaja importante, y es que sabiendo únicamente el proceso se puede conocer, cuales son las jaulas que tiene asignadas para computar.

Un caso muy probable, es que el número de jaulas totales no sea múltiplo del número de procesos. Las jaulas restantes se repartirían equitativamente entre los primeros procesos, para que la diferencia entre el número de jaulas que se computan en cada proceso sea la menor posible. Dado que el número de jaulas en una aplicación real es del orden de decenas de miles, la diferencia real entre los procesos será muy pequeña y no comprometerá el equilibrio de carga de trabajo. En el caso del ejemplo, si se aumenta el número de procesos a cinco, cada proceso le tocaría dos jaulas a procesar y sobrarían dos a repartir. Estas dos jaulas sobrantes las recibirán los procesos cero y uno, sumando un total de tres jaulas que tiene que computar cada uno de estos procesos. Aún con este reparto, se continua con la idea de repartir en orden, es decir, el proceso cero recibirá las tres primeras jaulas, el proceso uno los siguientes tres, y el resto de procesos seleccionarán de dos en dos.

Con el reparto previo se logra que la diferencia de cómputo entre los procesos llegue como máximo a una jaula de diferencia. Además, como desde el inicio se conoce el número de jaulas y el número de procesos, gracias al reparto ordenado, se sabe a su vez mediante un cálculo matemático sencillo, qué jaulas tiene seleccionadas cada proceso.

Con la lógica del reparto explicada, en la implementación paralela el proceso cero genera la matriz de partículas. En la Figura 3.4, se muestra el esquema general de la paralelización, y se puede observar cómo la primera fase de inicialización de las partículas la realiza le proceso cero. A continuación se pasará a una fase de comunicación en donde se envían las jaulas entre los procesos.

Para este envío, se utilizará la función Scatterv donde el proceso maestro se encargará de aportar los datos y repartirlos entre los procesos. Para esto hace falta calcular los vectores que definen la cantidad de datos que recibe cada proceso, así como el desplazamiento en la matriz donde se inicia el reparto por cada proceso. La creación de estos vectores es sencilla y rápida gracias a la lógica seguida en el reparto de las jaulas explicado previamente. Por tanto, una vez se generan los vectores en todos los procesos, éstos recibirán las jaulas correspondientes con las cuales, pueden empezar a generar resultados. Con el reparto de jaulas realizado, y habiendo implementado las funciones que calculan la puntuación y las restricciones de una jaula (explicado en la Sección3.4.2), se calculan estas puntuaciones y restricciones por jaula.

Una vez los procesos tienen los resultados de las jaulas, es necesario calcular el fitness de una partícula completa, realizando una fase de comunicación y sincronización como se observa en la Figura 3.4. Esta tarea es compleja debido a que un proceso puede tener o no, todas las jaulas de una determinada partícula. Si un proceso tiene todas las jaulas de la partícula, debería ser este el que computase los resultados de la partícula sin realizar ninguna comunicación. En contraposición si no tiene todas las jaulas, quiere decirse que comparte las jaulas de una partícula con otro proceso. Por tanto, para realizar el menor cómputo posible, uno de los procesos que comparte la partícula deberá de ser el encargado de computar sus resultados. En esta tarea, el resto de procesos de la partícula tendrán que enviar tanto los datos de las jaulas como los resultados parciales de cada una de ellas. Con todos los datos el proceso encargado de esta partícula podrá continuar con la evaluación de su partícula y realizar el resto de funciones.

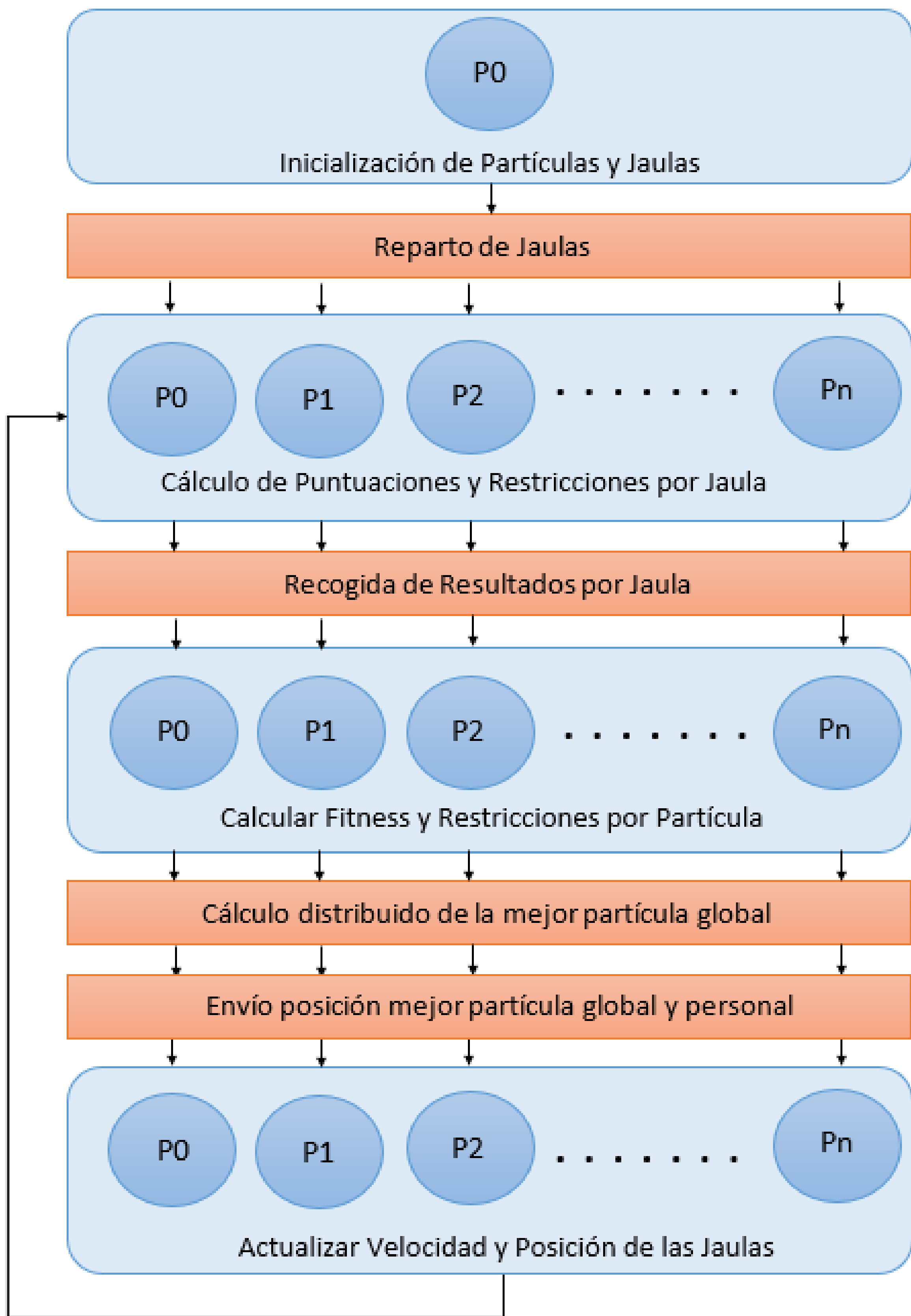


Figura 3.4: Esquema de comunicaciones y cómputo del algoritmo PSO paralelo.

Para realizar esta asignación de procesos responsables de calcular una partícula, se optó por seleccionar al proceso que contiene a la primera jaula de la partícula como encargada de realizar este cálculo. Por ejemplo, en la Figura 3.3, tanto el proceso cero como el uno, serían los encargados de calcular el fitness y las restricciones de la primera y segunda partícula respectivamente.

Sabiendo los responsables de realizar los siguientes cálculos, el resto de procesos tienen que enviar los datos de sus jaulas a los responsables para que calculen los valores para la partícula. Pero para realizar este envío de manera eficiente no solo hace falta saber quien es el responsable de calcular la partícula, sino también quienes son los otros procesos que comparten datos de la partícula. Conocer estos procesos permite utilizar las funciones de MPI de creación de grupos y de comunicadores explicadas en la Sección 2.2.3 que aumenta en gran medida el rendimiento de las comunicaciones.

Para que los procesos puedan realizar las funciones de creación de grupos y comunicadores es necesario crear una estructura de datos que almacene quienes son los participantes de esta comunicación, y cuantos datos envía cada uno.

Para generar esta estructura hay que observar en primer lugar cuándo varios procesos comparten una misma partícula. Un ejemplo de esto se visualiza en la Figura 3.3, donde el proceso cero comparte con el proceso uno las jaulas de una partícula. Además de compartir la partícula el proceso cero es a su vez responsable de su cálculo.

Esto se detecta simplemente realizando el módulo entre la jaula final del proceso y el número de jaulas por partícula. Si el resultado es cero, quiere decir que se ocupan todas las jaulas de las partículas. En caso contrario, compartirá la partícula con otro proceso. En el caso del ejemplo, el proceso cero acabaría en la jaula número cuatro, y habiendo seis jaulas por partícula, realizando la operación módulo tendríamos un restado de cuatro. Por tanto, el proceso cero comparte la partículas con más procesos.

Cuando se conoce la compartición de una partícula por parte de un proceso hay que calcular cuántos de los siguientes procesos comparten la partícula. Para esto se comprueba en que partícula inician cada proceso. En el ejemplo, el proceso uno inicia con la jaula cinco, lo cual implica que es la primera partícula. Mientras que el proceso dos inicia en la jaula nueve situada en la segunda partícula. Esto quiere decir que el proceso cero comparte partícula únicamente con el proceso uno, al ser el único proceso que tiene jaulas de su partícula final. Luego, se continuaría esta misma tarea con el proceso uno, dando lugar a que comparte partícula final con el proceso dos. Esta tarea se repite con todos los procesos para conocer todas las relaciones entre ellos.

Una vez finalizados estos cálculos se añaden los índices de los procesos que comparten partículas a la estructura que permitirá generara un grupo y un comunicador entre ellos. Esta estructura se rellena de forma ordenada permitiendo que primer elemento de la lista es el responsable del cálculo de la partícula.

La creación de grupos y de comunicador se realiza mediante las funciones de comm.group.Incl (listProcess) y comm.Create_group (newGroup) explicadas en la Sección 2.2.3. La función comm.group.Incl(listProcess) recibe una lista de procesos, que se corresponde con la estructura previa donde se tiene una lista de procesos ordenados por su identificador. Esto último es importante, ya que la función previa genera nuevos identificadores locales al grupo en función de la ordenación de la lista. Y por tanto, los responsables de calcular el fitness de las partículas siempre tendrán el identificador cero, debido a que el reparto de jaula se realizó en orden de proceso. Después de generar el grupo con la función previa, se utiliza comm.Create_group(newGroup), que crea un nuevo comunicador para permitir el envío de datos.

Una vez se genera un comunicador a partir de la estructura de datos previa, se puede aplicar para el grupo de procesos varias funciones Gatherv, explicadas en la Sección 2.2.2, que enviarán las puntuaciones, restricciones y valores de las jaulas a los procesos responsables de calcular los resultados de las partículas. Para realizar esta comunicación hace falta a su vez, calcular los parámetros para la función Gatherv, como son el desplazamiento de los datos y el número total de datos enviados por cada proceso. Estos datos deben de ser conocidos por todos y cada uno de los procesos que participen en la comunicación. Es por esto que, a la par de la generación de la estructura previa de la lista de procesos, se genera a su vez una estructura vectorial donde se guarda información referente a los procesos y a su reparto de jaulas. En esta segunda estructura se almacena la siguiente información.

- Número de partículas de las cuales el proceso es responsable de calcular sus resultados.
- Índice de la fila donde se encuentra la primera jaula designada al proceso.
- Índice de la fila donde se encuentra la última jaula designada al proceso
- Índice de la primera jaula del proceso.
- Índice de la última jaula del proceso.
- Número de jaula que tiene que enviar el proceso para calcular la partícula.
- Número de jaula que tiene que recibir el proceso para calcular la partícula.

Con todos estos datos, se pueden generar de manera sencilla los parámetros para la función Gatherv, teniendo en especial atención al penúltimo elemento que indica cuántas de sus jaulas será necesario enviar. Con esta información, se generan de manera iterativa los parámetros de número de datos enviados por cada proceso y el desplazamiento de cada conjunto de datos. El elemento root de esta comunicación será siempre el proceso cero, ya que el responsable de recibir los datos siempre tendrá el identificador cero en el grupo de procesos, como se explicó previamente.

Los procesos responsables se encargarán de recibir los datos finales y tratarlos de manera que se puedan usar con las funciones creadas para computar los resultados de las partículas como se describe en la Sección 3.4.2. Este cómputo se divide en dos fases, una en la cual se realizan los cómputos de las partículas donde el proceso responsable es dueño de todas las jaulas. Y una segunda fase en la cual, se calcula la última partícula añadiendo las jaulas externas de otros procesos. En ambos casos hay una fase de tratamiento de los datos donde se colocan los datos de manera adecuada y sencilla para el cómputo de los resultados.

Con esto se acabarían las fases de inicialización y cálculo del fitness del algoritmo PSO.

3.4.4. Comparación y envío de los resultados

Una vez los procesos responsables calculan los resultados de las partículas, se realiza la fase de comprobación y de obtención de la mejor partícula local. En primer lugar, se comprueba mediante los resultados de las partículas, que cumplen todas las restricciones. Si una partícula cumple todas las restricciones entonces se verifica si su posición actual es la mejor que ha tenido en todas las iteraciones, si es así, se reemplaza su antigua posición por la nueva mejor posición.

Después de actualizar la mejor posición de cada partícula, los procesos seleccionan individualmente la mejor partícula que poseen. El siguiente paso sería calcular la mejor partícula global, para eso mediante una llamada AllReduce 2.2.2 con la operación defina en MPI de MINLOC, se envía y calcula el mínimo de las puntuaciones de las mejores partículas de los procesos, junto además, qué proceso es el que contiene esta partícula. La partícula que tenga menor fitness será seleccionada como mejor partícula global. Para finalizar, el proceso seleccionado como dueño de la mejor partícula envía la posición de la partícula al resto de procesos mediante un Bcast 2.2.2. Con esto todos los procesos tienen la mejor partícula global del sistema. Aún así a los procesos que comparten jaulas de una partícula les faltaría obtener la posición de la partícula que comparten, ya que ésta es únicamente poseída por el responsable de calcularla. Para esto se aprovecha los comunicadores de los grupos compartidos, y el encargado de calcular los datos de la partícula, envía la posición de la partícula al resto de procesos. Y con esto quedarían realizados las dos etapas de comunicación consecutivas que se observan en la Figura 3.4.

En el algoritmo secuencial el cálculo del mínimo global de las partículas se realiza con las puntuaciones de todas las partículas. Mediante el cambio previo, se distribuye este cálculo y se aprovecha las funciones de MPI para determinar el proceso con la mejor partícula.

3.4.5. Cálculo de la velocidad y posición

El siguiente paso es calcular un vector velocidad que actualice las posiciones de las jaulas. El cálculo del vector velocidad, como se observa en el Algoritmo 1, consiste en la combinación de tres vectores diferentes:

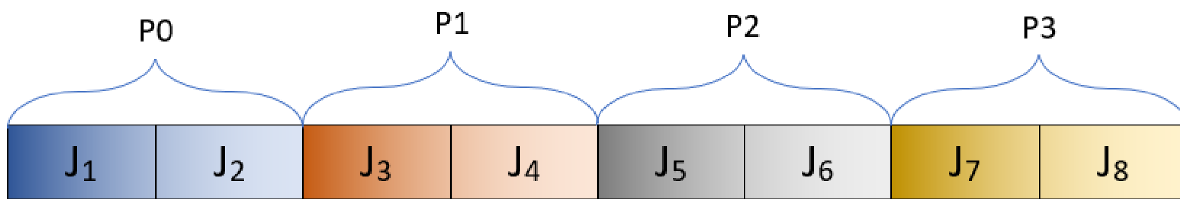
- La velocidad previa de la partícula, en el caso de la iteración inicial se inicializa la velocidad de manera aleatoria.
- La posición de la mejor partícula.
- La mejor posición de la partícula, a la cual corresponde el vector velocidad.

Estos tres vectores, en el algoritmo secuencial están implementados para ser aplicados en cada una de las partículas. Pero debido a que en el modelo distribuido se trabaja mediante jaulas, hay que modificar las estructuras de datos adecuadamente para adaptar los distintos vectores.

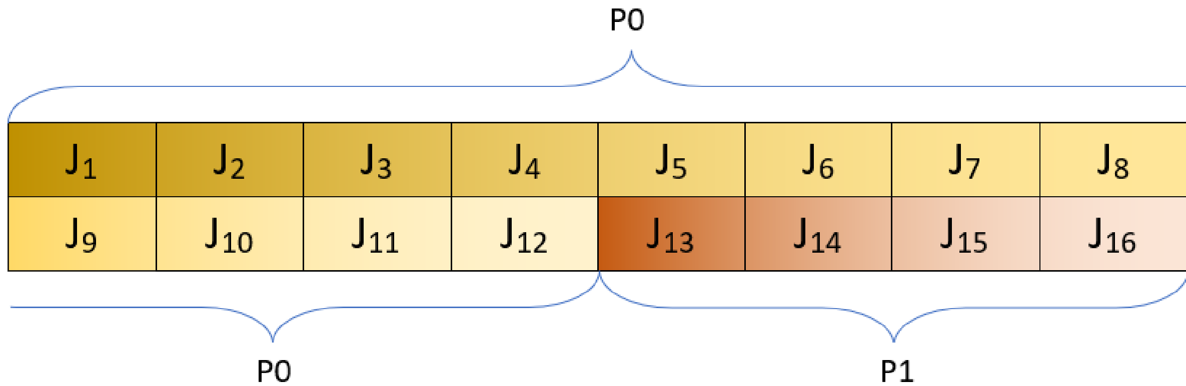
Esta tarea es complicada a la hora de la programación, debido a que las jaula que recibe un proceso, pueden estar localizadas en cualquier parte de la matriz de las partículas. Esto hace que las estructuras basadas en partículas deben ser modificadas para adecuarse a la estructura de vectores, en la cual se guardan las jaulas. Además, para mayor eficiencia, se realiza el cálculo de varios vectores de velocidad, para poder modificar todas las jaulas de un proceso al mismo tiempo, lo cual aumenta la complejidad de la programación.

Para realizar estas modificaciones de las estructuras, hay que tener varios casos en cuenta. En la Figura 3.5 se puede observar dos casos distintos de reparto de jaulas. En el caso de la figura (a) se observa como el proceso cero responsable tiene únicamente dos jaulas de una única partícula. En contraposición en la figura (b), el proceso cero responsable tiene todas las jaulas de la primera partícula y la mitad de la segunda. Se puede dar cualquier combinación de reparto atendiendo a las reglas explicadas en la Sección 3.4.3. Debido a estas distinciones en el reparto, el programa paralelo debe modificar las estructuras de datos de la manera adecuada para actualizar la velocidad y la posición.

En el caso de la mejor posición propia de una partícula, será necesario juntar todas estas posiciones en el orden correcto para poderlo aplicar a las jaulas del proceso. En el caso de la mejor partícula global puede suceder varias situaciones. Si un proceso tiene solamente un subconjunto de jaulas de una partícula, simplemente será necesario aplicar el mismo subconjunto de la mejor partícula global. Pero si las jaulas contemplan varias partículas va a ser necesario copiar varias veces la mejor partícula para poder modificar muchas jaulas al mismo tiempo. Además, no siempre las jaulas comienzan por el inicio de una partícula, y por tanto, también habrá que gestionar el comienzo de las estructuras para realizar los cálculos correctamente.



(a) Proceso responsable con escasas jaulas



(b) Proceso responsable con multitud de jaulas

Figura 3.5: Ejemplos de reparto de jaulas

Por tanto, usando la mejor partícula global, la mejor posición de cada partícula y aplicando los cambios necesarios a las estructuras, se realiza el cálculo de la velocidad de cada una de las jaulas, que conlleva a su vez el cálculo de nuevas jaulas. Esto último, conlleva a su vez la actualización.de las partículas, la cual es el objetivo final del algoritmo.

Como se observa en la Figura 3.4, con este cálculo se acabaría una iteración del bucle y se volvería a calcular el valor de las jaulas, enviar los resultados, etc. Esto se repetiría hasta el número máximo de iteraciones del bucle o hasta que se cumplan ciertos criterios. En el programa secuencial, se paraban de realizar iteraciones, cuando la mejoría entre la mejor partícula global y la anterior está por debajo de un determinado umbral.

Esto último, se mantiene en la versión paralela debido a que los criterios de parada se realizan sobre la información compartida por todos los procesos, por tanto no hay posibilidad de que varios procesos salgan del bucle, mientras que otros continúan en él.

Con todas las explicaciones de esta sección y las anteriores, se pondría fin al algoritmo PSO y a su paralelización.

3.4.6. Reparto de las partículas y sus valores

Como se menciona en la Sección 3.1, la parte final del programa consiste en retornar la posición y el fitness de todas las partículas. En secuencial, al tener las partículas en la matriz simplemente el método del algoritmo retorna la matriz de las partículas junto con sus puntuaciones. Pero en el programa paralelo creado previamente se tienen todas las puntuaciones y partículas distribuidas a nivel de jaula. Por tanto, habrá que crear esta matriz de partículas y de puntuaciones que se tenía en el programa secuencial.

En este caso, lo mejor será realizar un envío de toda la información a todos los procesos y que estos realicen las operaciones de pertinentes. El que solo un proceso realizase estas operaciones complica la paralelización, debido a que habría que refactorizar gran parte del código para que el resto de procesos no lo ejecute. Además, no mejora el tiempo de ejecución debido a que hay que esperar al último proceso a que realice los cálculos. El único inconveniente es el tiempo de comunicación que aumenta debido al envío de todos los datos a todos los procesos. Pero esto se contrarresta con el tiempo que los procesadores se quedarían sin realizar ninguna operación esperando al último proceso. Y como última ventaja, cabe destacar que mediante la observación de los resultados de todos los procesos se puede comprobar que no existen errores en el envío de las partículas y todos los procesos obtienen los mismos resultados. Y para añadir, el tiempo de programación entre realizar el envío de los datos a uno o a todos los procesos es el mismo, por tanto, la decisión tomada fue enviar los datos a todos los procesos.

En MPI para realizar las tareas previas existe la función Allgatherv 2.2.2. Para usarla habría que calcular parámetros equivalentes a los de Gatherv, pero en este caso hay que implementarlo para crear una matriz de partículas. Los procesos tienen todos los datos necesarios creados para poder calcular tanto el desplazamiento como el tamaño de los datos. Esto último es debido a que en la generación de estructuras que explicados en las Sección 3.4.3, se tienen todos los datos de todos los procesos. Al realizar la estructura de todos procesos, permite cálculos muy sencillos para saber cuantas jaulas tiene un proceso, o cuantas puntuaciones ha calculado cada uno.

Al realizar los cálculos de los parámetros únicamente es necesario llamar a la función Allgatherv, para que todos los procesos reciban las jaulas de todos los demás. Debido a que el reparto original de las jaulas era ordenado, la función Allgatherv ordena por defecto estos datos y no es necesario realizar ninguna operación más. Esta misma operación se realiza de la misma manera para las puntuaciones de las partículas.

Este sería el final de la paralelización completa del programa. En la Sección siguiente se mostrará cuales han sido los resultados de esta implementación.

Capítulo 4

Evaluación

En este capítulo se describe los experimentos realizados para comprobar la implementación paralela de AquiAID. En primer lugar, se desarrolla la metodología de experimentación, para después presentar los resultados obtenidos tanto de rendimiento como de escalabilidad. Se presentarán métricas tanto de tiempo de ejecución como de consumo de energía.

4.1. Metodología

En esta sección se define los parámetros para la realización de los experimentos que permiten medir la eficiencia de la implementación paralela, explicada en el Capítulo 3. Determinar la naturaleza de cada uno de los experimentos, permitirá realizar una medición exhaustiva de las características más primordiales del programa.

La experimentación de este programa paralelo se basa en dos conceptos claves, que son la escalabilidad y el consumo energético. Mediante la escalabilidad, se puede observar si la paralelización del algoritmo ha sido efectiva [61, 62]. Una implementación paralela deficiente no aportará una ventaja sustancial respecto a la implementación secuencial, no permitiendo aumentar la complejidad del programa. Por ello, es necesario realizar experimentos con un número de jaulas considerablemente alto, determinando métricas de rendimiento para la medición del programa paralelo.

En cuanto al consumo energético, el programa paralelo hace uso de un número mayor de recursos computacionales, lo cual puede llegar a implicar un aumento en la potencia. Es fundamental la medición del consumo energético, debido a que puede suponer un gasto económico importante que hay que tener en cuenta a la hora de implementar un sistema de soporte como AquiAID. El consumo se verá afectado tanto del número de recursos computacionales, como del tiempo de uso de estos recursos, por tanto, además del consumo, se realizan medidas ligadas a la eficiencia energética que permitirá observar, cómo de eficaz es el consumo de cada unidad de energía en cada experimento.

Además, de los dos puntos previos, se utilizarán dos métricas más. La primera métrica que permite relacionar el tiempo de ejecución y el consumo energético. Para esto se utiliza la métrica de *Energy-delay product (EDP)* [63], la cual, es utilizada para ponderar la eficiencia energética. Su cálculo consiste en la multiplicación de la energía consumida por el tiempo de ejecución, permitiendo tener una medida simple que relaciona los dos conceptos previas, permitiendo medir la eficiencia en el consumo energético. Y la segunda métrica utilizada es la Eficiencia, la cual se puede observar su definición en la Fórmula 4.1.

$$Eficiencia = \frac{Speedup}{N} = \frac{T_{secuencial}}{T_{paralelo} * N} \quad (4.1)$$

Estos son los puntos claves a la hora de realizar la experimentación de la implementación paralela. Por supuesto, se realizan también experimentos con la implementación original de AquiAID, que permitirá una comparación de los dos estados finales del programa. Para el diseño de los experimentos hay que definir varios conceptos previos como son:

- *Plataforma de ejecución*: debido a que para la evaluación de la implementación paralela es necesario la utilización de múltiples computadores. Esta limitación es fundamental, ya que dependiendo del número de computadores disponibles se podrán realizar experimentos de mayor tamaño. Es por esto, que con el permiso del departamento de Ingeniería Informática y Electrónica de la Universidad de Cantabria he podido hacer uso del cluster Tritón. Este cluster está conformado con un máximo de cinco servidores con las características observadas en el Cuadro 4.1. Con este servidor, que cuenta con una cantidad de ochenta cores físicos a la hora de ejecutar los experimentos, es posible generar hasta ochenta procesos MPI independientes a la hora de realizar los experimentos.

- *Mediciones*: al estar en un cluster como Triton, donde los usuarios lanzan sus trabajos para ser ejecutados por lo procesadores, es necesario tener alguna manera de conocer tanto el tiempo de ejecución de los trabajos, como la energía consumida por estos. Debido a las limitaciones obvias de ejecutar en un cluster, para realizar la planificación adecuada de todas las tareas enviadas al cluster, se usa el gestor de recursos Slurm [64], que garantiza un adecuada distribución de cores a las tareas. Por ello, las medidas se tienen que realizar teniendo en cuenta esta herramienta.

Nº Servidores máximos	6
Nº Servidores útiles	3
Nº de procesadores por servidor	2
Modelo CPU	Inter(R) Xeon(R) Silver 4114
Frecuencia Maxima CPU	2.20 GHz
Nº cores físicos por procesador	10
Capacidad Memoria por Nodo	32 GB
Almacenamiento	3 discos de 8TB en RAID 5
Tarjetas de Red	Intel Corporation Ethernet Connection X722 for 1GbE
Modelo de switch de red	Switch Netgear Prosafe 24 Puertos Gigabit más 4 puertos SFP+
Sistema Operativo	CentOS

Cuadro 4.1: Componentes de Triton

Es por esto, que Slurm tiene incorporado una base de datos donde guarda toda la información relacionada con la ejecución de programas. En concreto, se puede realizar consulta de los datos de consumo energético y tiempo de ejecución de cada trabajo lanzado. Por tanto, con la información recabada mediante Slurm de energía y tiempo, es posible realizar los análisis y las métricas pertinentes para la evaluación del programa.

- *Número de Jaulas y Partículas*: con el entorno de ejecución caracterizado, es fundamental definir los parámetros iniciales del programa. Para la aplicación de AquiAID, es necesario indicar dos parámetros iniciales que determinan el comportamiento del programa, que son el número de jaulas y de partículas. Ambos parámetros han sido explicados en detalle en la Sección 3.4.1. El aumento de cualquiera de esto dos parámetros implica un en la cantidad de trabajo (operaciones) a realizar, traduciéndose a su vez en un aumento en el tiempo de ejecución de manera lineal. Por esta razón, es fundamental un diseño previo del número de jaulas y partículas, para realizar experimentos útiles para cada una de las validaciones a realizar.

  Es necesario un estudio de estas dos variables, para adecuar los experimentos a casos reales que puedan ser contemplados por empresas del sector. En primer lugar, el número de partículas viene determinado por la precisión que se pretende obtener. Al aumentar el número de partículas, se incrementa el espacio de búsqueda donde encontrar soluciones al problema, pero aumenta el tiempo de cómputo. Hay multitud de estudios referentes a esta variable en función de la naturaleza del problema [65, 66, 67, 68]. La conclusión de estos estudios es que un número adecuado de partículas suele rondar entre veinte y cincuenta partículas. Pero estudios más recientes [69], contemplan la necesidad de aumentar el número de partículas acercando el número al centenar, dependiendo sobretodo del tamaño del problema. Además, como la paralelización realizada se basa en el número de jaulas, el número de partículas es interesante fijarlo, para tener experimentos lo más homogéneos, reduciendo así las variabilidades que se pueden dar debido al desequilibrio en el cálculo del fitness. Por tanto, el número de partículas seleccionado en todos los experimentos es de cien partículas.

Y en cuanto al número de jaulas, este valor en casos reales puede llegar a ser muy variable, desde productores locales con una o dos jaulas, hasta grande empresas de la acuicultura con decenas de ellas. El número de jaulas afecta a las dimensiones del problema, y por tanto, a su tiempo de cómputo y a la convergencia del algoritmo. Pero debido a que el objetivo de estos experimentos no es la valoración de la precisión del algoritmo, sino el efecto del tamaño del problema a la paralelización el número de jaulas será dependiente del objetivo que tengan los experimentos y de los parámetros a evaluar. Gracias a esto, es posible modificar el tamaño del problema, sabiendo que procesos pueden verse afectados por estos cambios, y así predecir el comportamiento del programa.

Con los aspectos previos definidos, se pueden realizar los experimentos que se explicarán en detalle en las siguientes secciones. Los experimentos son cargados en la cola de Slurm, mediante un script de bash, que permite automatizar el proceso de generación de los experimentos. Cabe destacar que los experimentos son repetidos en cinco ocasiones para, mitigar las variaciones inesperadas que se produzcan en las ejecuciones. Aún así las desviaciones encontradas entre las ejecuciones son muy pequeñas, por lo que los resultados son muy robustos.

Recalcar, que se han realizado ejecuciones independientes para obtener las trazas de ejecución en algunos casos significativos y poder analizar así, problemas como el desequilibrio de carga de trabajo o las comunicaciones [70]. Las trazas permiten recabar la mayoría de información del estado de los procesos durante la ejecución del programa paralelo. Estas trazas son realizadas mediante la herramienta Extrae [71], la cual inserta código a la hora de la ejecución del programa, para extraer toda la información posible de los procesos MPI.

Extrae [72] permite la obtención de información de diferentes librerías y lenguajes, para este caso particular, es posible obtener datos de un programa paralelo en Python cuando se utiliza la librería de Mpi4py. La inserción de código realizada por Extrae, aumenta el tiempo de ejecución del programa paralelo de manera considerable, esto provoca que no es posible utilizar las mediciones de tiempo ni de energía de estos experimentos.

Aún así, las trazas recogidas aportan una gran cantidad de información relevante para comprobar la ejecución del programa. En este aspecto, Extrae es capaz de medir todos los tiempos tanto de ejecución como de comunicaciones de todos los proceso. Esto permite observar y medir, las iteraciones entre los procesos, y las posibles limitaciones que puede tener el programa paralelo.

Extrae finaliza con la salida de un fichero de datos muy complejo, que se vuelve ilegible para la mayoría de usuario de la herramienta, es por esto que se usó la herramienta Paraver [73]. Este programa, recoge la salida de la traza de Extrae y permite el visualizado de la ejecución de todos procesos. Esto simplifica la tarea de analizar la ejecución del programa, debido a que es posible observar en una figura gráfica todas las comunicaciones, tiempos de espera, y tiempos de ejecución de todos los procesos de forma ordenada en el tiempo.

Además, Paraver es capaz de la lectura del fichero de traza y analizar todos los tiempos de comunicación y ejecución. Paraver cuenta con una cantidad considerable de ejemplos que pueden ser utilizados por los propios usuarios, usando las trazas que los usuarios han creado. Esto permite crear gráficas y métricas numéricas que permiten un análisis mucho más sistemático del comportamiento del programa. Con estas herramientas es posible crear métricas específicas que valoren objetivamente los comportamientos de los procesos, y el rendimiento final del programa completo.

Una vez realizados los experimentos llega la fase de recogida y comprobación de los resultados. La recogida de la información en los experimentos sin trazas, se realiza de manera automatizada leyendo la base de datos de Slurm, y los ficheros de salida de la ejecución de cada trabajo [74, 75]. Se vuelcan los datos en un fichero de texto estilo csv, y son analizados mediante scripts de Python para realizar medidas y gráficas que se explicarán en cada una de las secciones. Los experimentos con trazas son recogidos y analizados manualmente mediante la herramienta de Paraver.

Mediante el análisis se Extrae el comportamiento del programa en las distintas situaciones en las cuales se ejecuta. Con ello se llegará a conclusiones de cómo es la calidad de la implementación paralela y la mejora que supone respecto al programa secuencial. Para este análisis, se realizarán gráficas y figuras explicativas de los resultados que permitan de manera sencilla entender lo sucedido en la paralelización.

Y para finalizar, el último paso ha sido la realización de un modelo simple de predicción de ejecución del programa. Mediante los datos recabados en los experimentos es posible realizar un modelo, que permita de manera sencilla calcular el tiempo que va a tardar el programa en función del número de cores utilizados y el número de jaulas y partículas. Esto tiene una utilidad práctica para cualquier empresa que desee utilizar este software, debido a que es posible tener una estimación de tiempos de cómputo del programa, o más interesante, estimar el número de cores necesarios para realizar una planificación en un tiempo determinado.

4.2. Experimentos de rendimiento

Los primeros experimentos realizados comparan los mismos problemas utilizando distinto número de cores, es decir, utilizando el mismo número de jaulas y de partículas se realizan experimentos con distinto número de cores. Esto permite comparar si la implementación paralela ha tenido efecto tanto en el tiempo de ejecución como en la energía. Esta comparación se puede observar en las Figuras 4.1 y 4.2, en estas gráficas se representa en el eje de abscisas el número de jaulas totales utilizados en cada experimento, y en el de ordenadas el tiempo de ejecución en segundos, la energía consumida en kilojulios y la métrica EDP en megajulios por segundos respectivamente. En ambas figuras, se muestran tres líneas que representan el número de cores utilizados en cada experimento. La línea roja representa la ejecución con el programa secuencial, y la azul y verde representa los experimentos con la implementación paralela con 60 y 80 cores respectivamente.

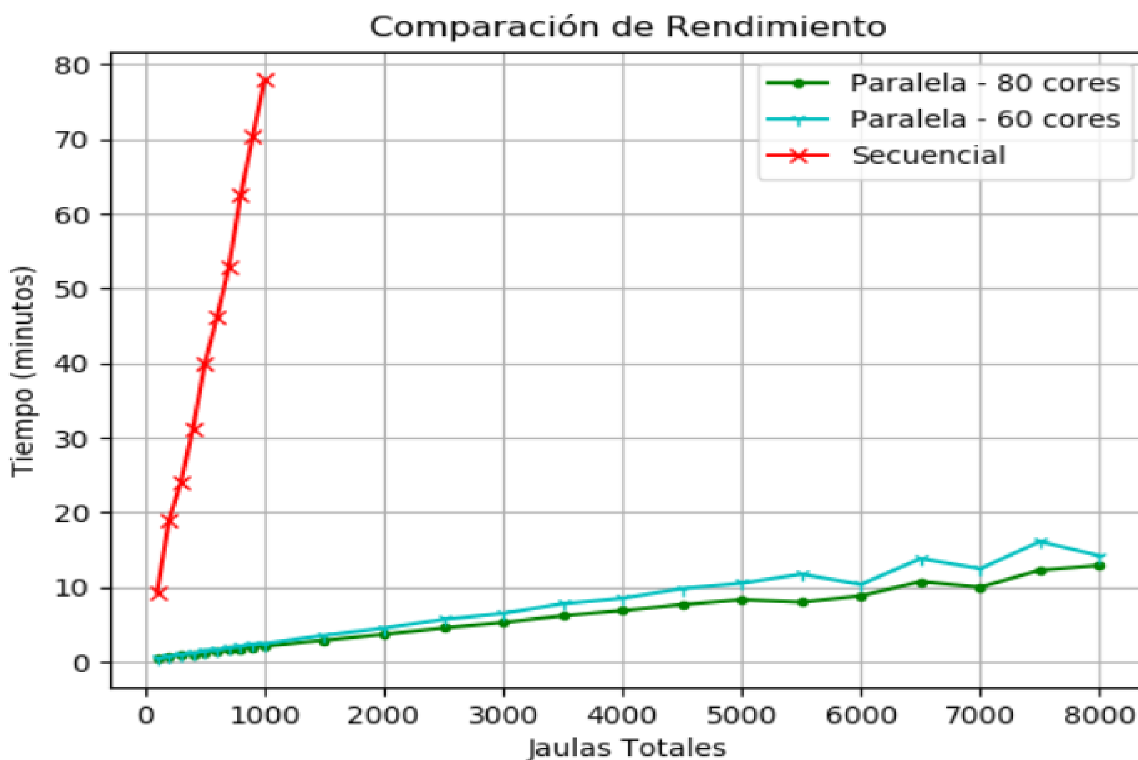
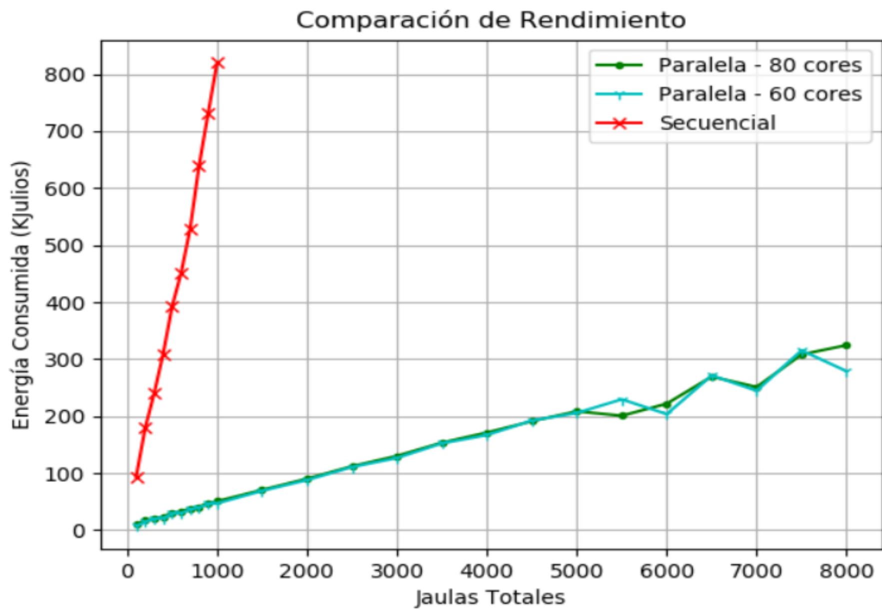


Figura 4.1: Tiempo de ejecución del programa secuencial vs paralelo

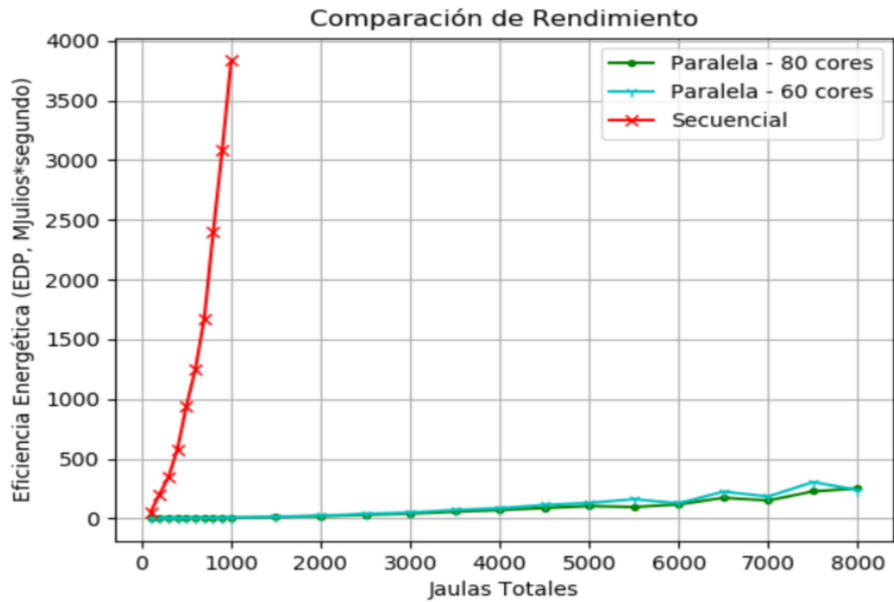
En la Figura 4.1, se observa la mejora de la implementación paralela en el tiempo de ejecución. El programa secuencial tiene un rendimiento muy deficiente comparado con el programa paralelo. En el mismo tiempo que el programa secuencial realiza una ejecución con 100 jaulas, el programa paralelo con 60 y 80 cores es capaz de realizar experimentos con 4000 y 6000 jaulas respectivamente. Esto permite realizar planificaciones que durarían días en el programa secuencial, a únicamente un par de horas si se ejecuta de manera paralela con 60 u 80 cores.

En general el aumento del tiempo se realiza de manera lineal, con respecto al número de jaulas, exceptuando pequeñas variaciones con los ejemplos paralelos con más de 5000 jaulas. Esto puede ser debido a que con ejemplos más grandes las variaciones en los repartos de jaulas y partículas se vuelven más importantes e influyen en el tiempo de ejecución. Y para finalizar hay que añadir, que las diferencias entre los dos tiempos de ejecución paralelos no disminuye de manera proporcional al aumentar el número de cores, esto se detallará en profundidad en el siguiente Sección 4.3, donde se realizará un estudio profundo de la escalabilidad.

Este primer paso, demuestra la capacidad de paralelización que tiene el programa de AquiAID, y en específico, el algoritmo de PSO utilizado. Una vez visto la mejora en el tiempo de respuesta del algoritmo, es importante comprobar el gasto energético que tiene la implementación paralela. Debido al uso de varios de computadores, es importante la medición de energía, ya que puede suponer un enorme gasto para las empresas que implementen AquiAID.



(a) Energía Consumida del programa secuencial vs paralelo



,

(b) Energía Consumida del programa secuencial vs paralelo

Figura 4.2: Eficiencia Energética del programa secuencial vs paralelo

En la Figura 4.2(a), se muestra el consumo energético de las tres ejecuciones. Se observa un comportamiento distinto al presentado en el tiempo de ejecución. En la figura se observa como el consumo energético de la implementación secuencial tiene una correlación clara con el tiempo de ejecución. En cada uno de los experimentos secuenciales, el tiempo de ejecución aumenta junto a la energía en la misma proporción, lo cual tiene sentido al ser una ejecución secuencial y lo único que cambia es el tiempo.

En cuanto a los ejecuciones paralelas podemos ver un comportamiento muy particular. En primer lugar, lo más singular es que no existe una ganancia tan superior con el programa secuencial como en el caso del tiempo de ejecución. Esto es debido a que la ejecución en varios cores, implica que la energía consumida es proporcional al número de cores activos aumentando en gran cantidad la potencia consumida. Donde antes en el tiempo de ejecución, se podría realizar ejemplos 40 y 60 veces mayor, ahora con la misma energía, se podrían realizar ejemplos con 20 veces más jaulas. Esto sigue siendo una gran mejora respecto al secuencial, porque además de realizar ejemplos más grandes con la misma energía, se realiza en tiempos de ejecución mucho menor.

A partir de esto, se puede observar que la energía consumida por los experimentos con 60 y 80 cores son muy similares en más de la mitad de la ejecución. Esto es debido a que aunque con 80 cores la potencia consumida es mayor, se contrarresta con el tiempo de ejecución menor que tiene, y en el cómputo final de la energía, prácticamente ambos ejemplos consumen muy parecido. En cuanto a la energía consumida con los experimentos con más de 5000 jaulas, se puede observar desviaciones donde a veces se consume más con 60 cores y otras con 80. Esto se debe a la variación vista en el tiempo de ejecución en los mismos ejemplos, estas pequeñas variaciones vistas anteriormente tienen la misma naturaleza presenciada en el tiempo de ejecución.

Hay que recalcar la Figura 4.2(b), donde se observa la eficiencia energética de los experimentos de rendimiento. Aquí se examina la gran diferencia entre el programa secuencial y paralelo, al juntar en esta métrica la energía y el tiempo, se extrae que el consumo energético en el secuencial es desperdiciado en comparación con el programa paralelo. Se observan los mismos comportamientos que en los casos anteriores, donde prácticamente el paralelo es entre 40 y 60 veces mejor en eficiencia, y existen las variaciones en los ejemplos grandes vistos en las gráficas anteriores.

4.3. Experimentos de escalabilidad

Comprobado que la versión paralela mejora la versión secuencial, tanto en tiempo de ejecución como en consumo de energía. Ahora es interesante realizar un análisis de la escalabilidad del problema, es decir, como evoluciona el tiempo y la energía del programa cuando añadimos recursos computacionales. Con esta prueba, se obtienen resultados con respecto a la calidad del balanceo de carga y a los límites en el aumento de recursos. En este modelo se basa el siguiente experimento, donde se fija el número de jaulas y partículas, y se van añadiendo cores para observar la evolución del tiempo y la energía consumida.

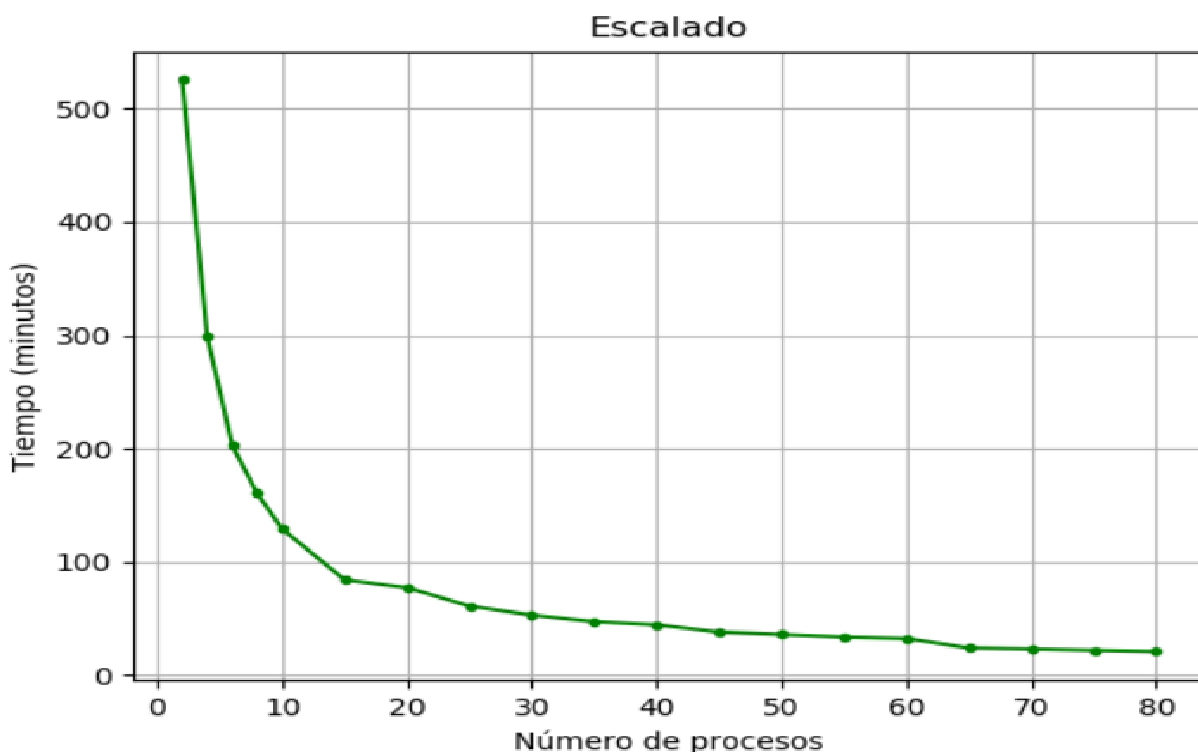


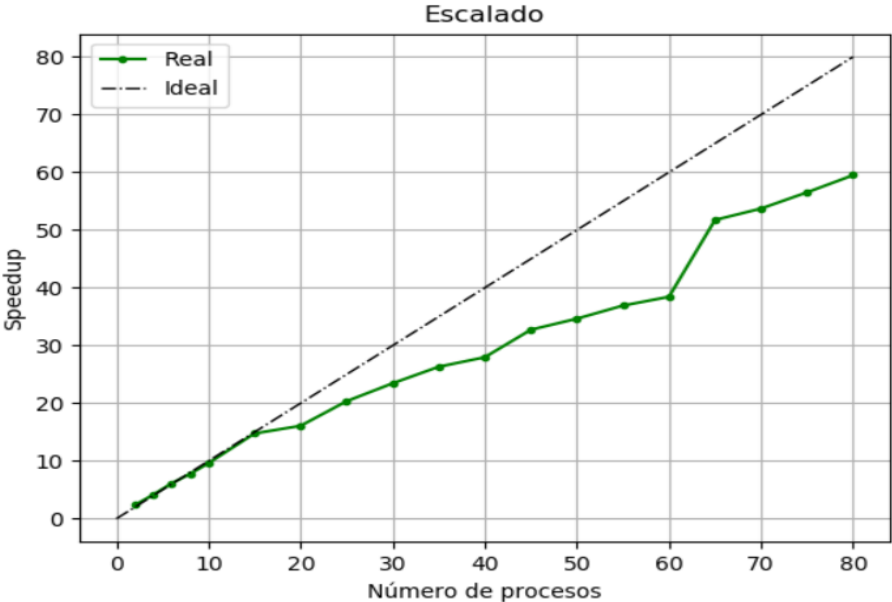
Figura 4.3: Tiempo de ejecución de los distintos procesos con 20000 jaulas

En la Figura 4.3, se representa en el eje de abscisas el número de procesos, y en el eje de ordenadas está definido el tiempo de ejecución en cada experimento. En este caso se han utilizado un total de 200 jaulas por partícula, dando lugar a un total de 20000 jaulas totales. Este número de jaulas llega a ser representativo de una empresa grande de acuicultura, donde es necesario planificar una gran cantidad de jaulas al mismo tiempo para obtener el mayor beneficio posible.

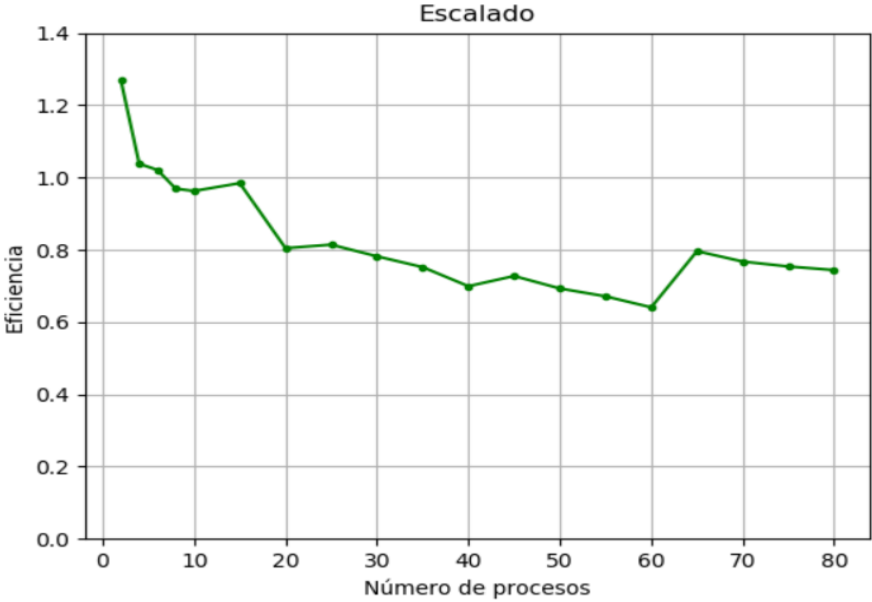
En la Figura 4.3, se puede observar el escalado del programa al aumentar el número de cores. En este caso, se observa el comportamiento habitual de una paralelización correcta, en primer lugar el descenso del tiempo de ejecución se realiza de manera exponencial, pero después se va atenuando este descenso poco a poco hasta que llega un punto concreto donde el aumento de recursos no mejora significativamente el tiempo de ejecución.

Y este es el comportamiento de la implementación paralela, utilizando hasta 15 cores, el descenso en el tiempo de ejecución es muy significativo, pero desde los 20 cores hasta los 65 cores se produce una desaceleración de la bajada de tiempo, si bien se sigue mejorando lentamente. Y a partir de los 65 cores el tiempo no disminuye de manera significativa, esto es debido, a que la carga computacional de la fase paralela del programa se ha vuelto tan pequeña, que la parte secuencial no paralelizada empieza a ganar mucho peso en el tiempo de ejecución.

En este experimento debido a que se utiliza un número de jaulas lo suficientemente grande y estático, es posible comparar de manera numérica la mejoría del programa paralelo contra el secuencial. En concreto se puede medir el speedup del programa paralelo respecto al secuencial, esto es lo que se observa en la Figura 4.4.



(a) Speedup de los experimentos de escalabilidad



,

(b) Eficiencia en los experimentos de escalabilidad

Figura 4.4: Evolución del Speedup y Eficiencia en la escalabilidad

En la Figura 4.4(a) se muestra el incremento del speedup cada vez que se añaden nuevos recursos, el speedup ideal sería el que corresponde con la recta y=x representada en el gráfico. Analizando la recta real del experimento, en los primeros 15 cores el aumento del speedup es proporcional al aumento de los cores, mezclándose con la recta real. A partir de este número de cores, el speedup sigue aumentando pero con menor inclinación, lo que implica, que se separa de la ideal, y esto se debe a que se empiezan a notar los efectos de las comunicaciones en el tiempo de respuesta del programa. Aún así, el speedup sigue aumentando de manera continua, lo cual indica que la paralelización sigue reduciendo el tiempo de computo. En el caso de 65 cores hay un aumento muy significativo del speedup, esto se debe a que al usarse un nuevo nodo, parte de la carga total de trabajo es recibida por el nuevo nodo, que al no usar todos los cores no satura las estructuras arquitectónicas compartidas por los cores, como las caches, los buses de memoria, QPI, etc, lo que concluye en una reducción en la computación. Esta hipótesis se ve reafirmada cuando se observa este efecto, pero en menor medida en los ejemplos de 25 y 45 cores, con una subida del speedup al utilizar un nuevo nodo.

Para más detalle en la evaluación del rendimiento, la Figura 4.4(b) muestra la eficiencia global del algoritmo en función del aumento de cores. Esta eficiencia es calculada como el speedup dividido entre el número de cores utilizados, lo que permite observar la tendencia del speedup, si se aumentan el número de cores. En primer lugar, cabe destacar que los primeros ejemplos con muy pocos cores se observar una eficiencia mayor que uno, lo cual implica que se produce una situación conocida como speedup super-lineal. Esto se produce cuando el speedup es mayor al número de cores utilizados, o su equivalente, cuando la eficiencia es mayor que uno, y es debido a que en la paralelización, la jerarquía de memoria se ve beneficiada con la partición de datos, debido a que es posible guardar la mayoría de datos en las caches. Esto reduce drásticamente el acceso a memoria principal, reduciendo consigo la latencia en la recuperación de los datos y por tanto, realizando el cómputo mucho más rápido.

A partir de este suceso, para el resto de procesos es observar un descenso gradual de la eficiencia, algo común ya a que al aumentar el número de recursos computacionales, el speedup ganado empieza a ser menor. Esto es debido a que la parte paralela empieza a tener menor peso computacional, y la secuencial empieza a tener ganar un porcentaje mayor del tiempo de cómputo. Existen ciertos momentos en la eficiencia donde se ven comportamientos irregulares.

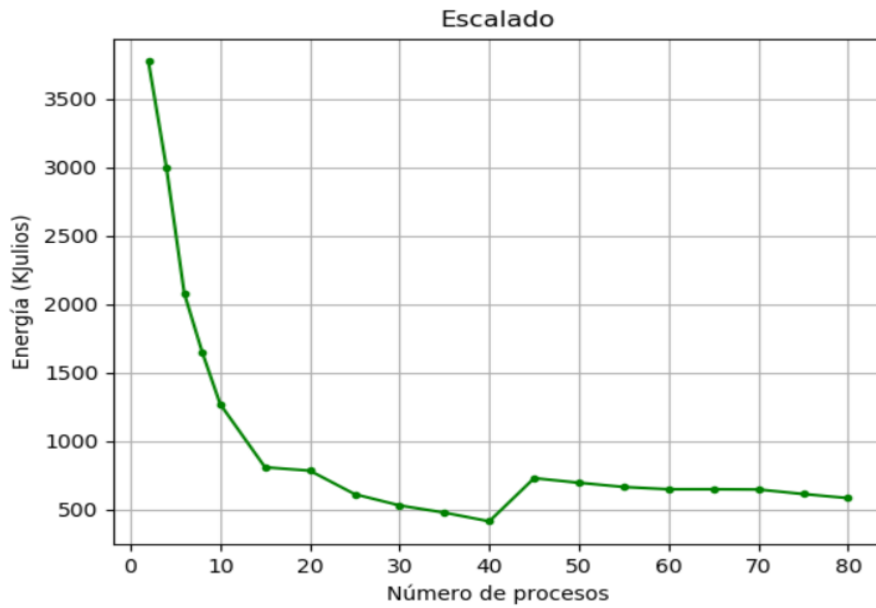
En primer lugar, entre 10 y 15 cores se observa un aumento en la eficiencia, lo cual puede deberse a la utilización de un segundo procesador. Debido a que con 15 cores se está utilizando el segundo procesador del servidor, incrementando la caché L3 total utilizada y los canales de acceso a memoria. Esto implicaría una mejora en la eficiencia que se aprecia en los experimentos de escalabilidad. En los experimentos donde pueden suceder efectos similares con más cores, se ven reducidos estos efectos a causa de que este suceso solo se produce en uno de los servidores y no en todos los utilizados, lo cual mitiga su importancia en los resultados.

La siguiente situación se observa con 15 y 20 cores la eficiencia baja de 1 hasta 0.8 respectivamente, esto puede ser debido, a que el uso de todos los cores provoca que se saturen los canales de acceso a la memoria principal, lo que implica contención en el acceso a los datos. Esto explica esta baja tan brusca que se presenta en esta parte de la gráfica. Con más procesos se aprecia este descenso gradual con alguna imperfección, hasta llegar a los valores de 60 y 65 cores. Con estos procesos se aprecia lo explicado con la Figura 4.4(a), donde el añadido de un nodo mejora significativamente el speedup, y a su vez la eficiencia. Como conclusión del análisis del tiempo en el escalado, es importante mencionar que la eficiencia conseguida en cada uno de los experimentos es muy alta, nunca bajando del 0.6. Esto permite concluir que la paralelización del algoritmo permite un alto escalado del programa consiguiendo siempre mejoras muy amplias con respecto al programa secuencial.

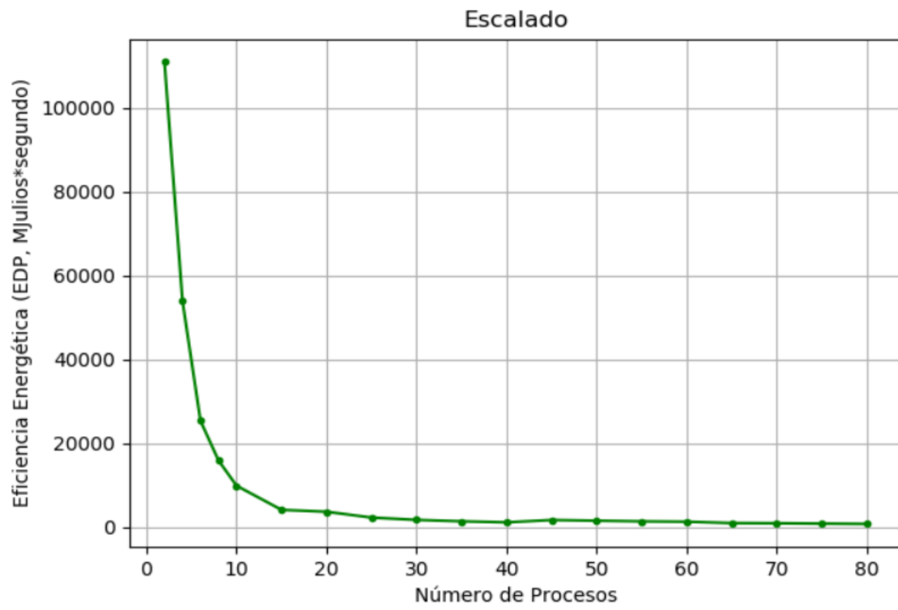
El siguiente apartado que atender es el consumo energético, y comprobar si tiene un comportamiento igual o parecido al tiempo de ejecución. Como se observa en la Figura 4.5(a), el comportamiento de la energía es muy semejante al del tiempo de ejecución. Lo destacable está en el paso de 40 a 45 cores, donde aumenta la energía consumida de 500 Kjulios hasta 750 KJulios. Esta diferencia puede ser debida a que el encendido de un nodo más provoca un aumento del consumo energético mucho mayor, y además al no mejorar en gran medida el tiempo de ejecución, el consumo no desciende.

La eficiencia mostrada en la Figura 4.5(b), concluye con que el consumo y el tiempo de ejecución tiene un comportamiento similar formando en la eficiencia la curva más usual en este tipo de escalado. Al seguir el tiempo de ejecución y el consumo energético un patrón similar de escalado, permite concluir que el algoritmo paralelizado escala de manera correcta, aún con una gran cantidad de jaulas como sucede en este experimento.

Con estos datos, se concluye que el escalado de la paralelización sigue todos los comportamientos habituales de una implementación correcta de una aplicación paralela. Esto permite realizar ahora una análisis más conciso del detalle de la aplicación, para observar los puntos de mejora que puede tener la implementación paralela.



(a) Energía consumida de los distintos procesos con 20000 jaulas



,

(b) Eficiencia energética de los distintos procesos con 20000 jaulas

Figura 4.5: Consumo y Eficiencia en la escalabilidad

4.4. Análisis de Trazas y Modelo de Predicción

En esta sección se pretende realizar una análisis en profundidad de todos los aspectos que influyen a la hora de ejecutarse un programa paralelo. Lo primero a mostrar en esta sección, es cómo se ejecuta el programa paralelo en el cluster y que posibles observaciones se puede realizar al respecto.

Para esta observación se obtienen ciertas trazas de los experimentos previos, y se realizan utilizando la herramienta Extrae, y se recogen y analizan los resultados con Paraver. Con esto se obtienen diagramas como el mostrado en la Figura 4.6, donde se pude observar como se ejecuta el algoritmo paralelo.

En primer lugar, se observa una etapa grande de cómputo debido a que es la parte de inicialización de las estructuras necesarias para planificar. Luego se observa una parte de comunicación bastante amplia correspondiente al envío de datos que el proceso cero lee, de la base de datos y difunde al resto de nodos. A partir de ahí, se sigue con una pequeña parte de cómputo y se empieza a realizar el algoritmo PSO.

Lo primero en realizarse es el envío de los datos de partida, que se correspondería con la primer franja amarilla, que es algo más grande que el resto. Esta gran parte de comunicaciones solo se realiza en esta ocasión y por tanto no limitará en exceso la ejecución del programa.

A partir de estas fases iniciales, se repiten constantemente el patrón de cómputo y comunicaciones. Este patrón corresponde con el cálculo de jaulas y partículas, explicada en la Sección 3.4, donde se calculan sus nuevos valores, para después realizar una comunicación de los resultados. Como se observa, estos tiempos de comunicación de las partículas y jaulas son muy cortos, y representan una parte muy pequeña del tiempo total. Esto hecho se repite en todas las iteraciones del algoritmo, treinta para este ejemplo. Una vez finalizado el algoritmo la última fase de comunicación se debe a la recogida final de todas las partículas en todos los procesos. El reparto final como sucede en los casos anteriores, tiene un impacto leve tiempo final del programa.

La conclusión a la que se puede llegar es que en los tiempos de comunicación no existe un gran cuello de botella que reduzca la eficiencia de la paralelización. Si se debe de tener en cuenta que debido a la propia naturaleza del algoritmo PSO las comunicaciones se repiten en multitud de ocasiones, pero independientemente del tipo de paralelización, es complicado reducir el número de comunicaciones.

Para finalizar el trabajo, se quiere contribuir a la implantación en las empresas del software AquiAID mediante un modelo de predicción de tiempos de ejecución del programa paralelo. Es interesante para una empresa de acuicultura que vaya a adoptar el software paralelo desarrollado en este trabajo, cuánto tiempo es necesario para planificar los recursos disponibles de la empresa.

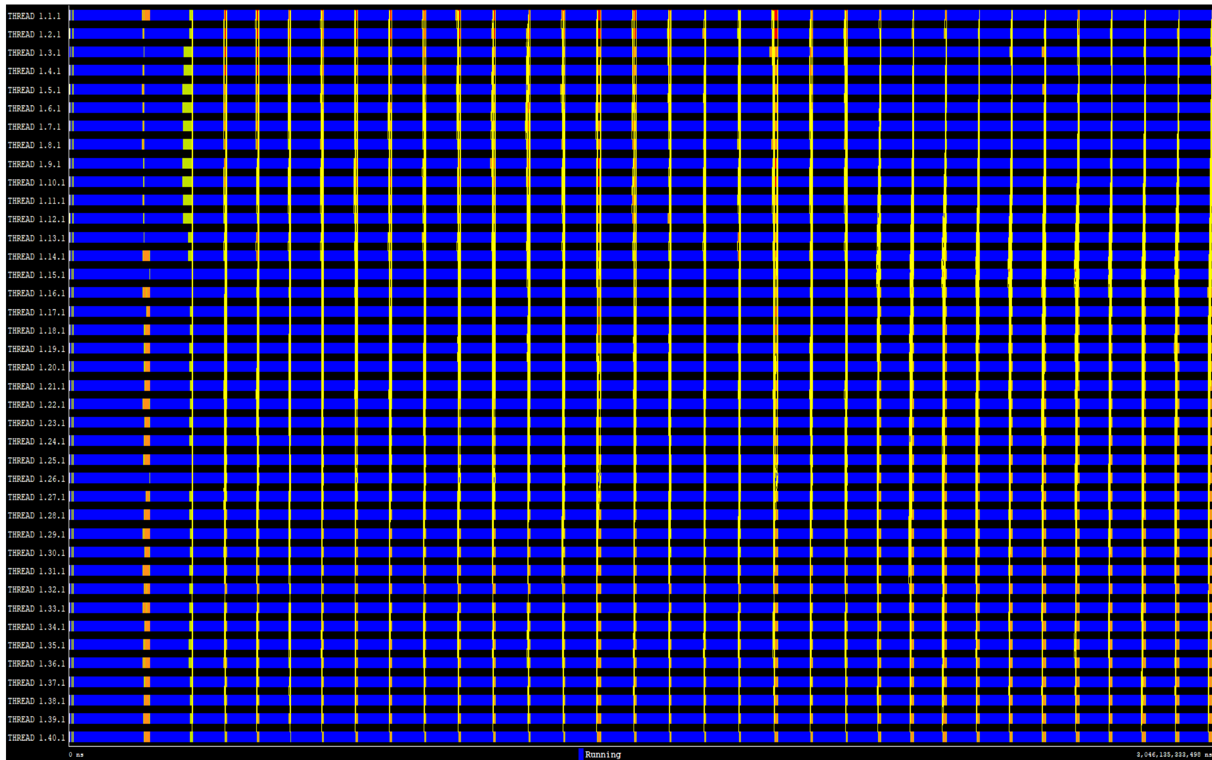


Figura 4.6: Traza del experimento de escalabilidad con 40 procesos

Es por esto que para completar las aportaciones al software de AquiAID, se ha creado un modelo sencillo de predicción del tiempo de ejecución del programa paralelo desarrollado en este trabajo. Con ello una empresa puede calcular de forma rápida, una aproximación del tiempo que llevaría realizar una planificación completa de los recursos disponibles.

Este modelo ha sido creado mediante los datos del experimento de la Sección previa 4.3. Además, destacar que este modelo es realizado con la plataforma explicada en el Cuadro 4.1, y por tanto, en otro tipo de arquitectura los resultados variarán , pero el objetivo de este modelo es aportar una aproximación del tiempo de respuesta, y no un modelo completo del comportamiento del programa en el cluster.

El modelo esta creado en base a buscar una relación entre los datos de la Figura 4.3. En concreto, para encontrar una relación matemática entre los distintos datos, de las múltiples opciones que existen, el modelo que se ajusta de manera más precisa a los datos es el ajuste potencial. Este ajuste se basa en una fórmula del tipo $P = a * x^b$, la cual es utilizada en multitud en casos prácticos como en aceleraciones de vehículos, fricción del aire en objetos en movimiento, etc.

El ajuste potencial es idóneo para este ejemplo, debido a que permite modelar tanto el aumento del tamaño del problema, como la variación del número de cores con una misma fórmula muy sencilla. En este caso como se observó en la Figura 4.3, el aumento de cores reduce el tiempo de ejecución de manera bastante precisa mediante una función de la forma x^b, siendo b un factor negativo para representar el descenso del tiempo en función del aumento

de cores. Y para el tamaño del problema, como se observa en la Figura 4.1, el tamaño aumenta el tiempo de respuesta de manera lineal en todos los casos, y por tanto, multiplicando las variables de jaulas y partículas por un factor fijo se tendría representado el tamaño del problema en la fórmula. Por consiguiente el modelo de referencia obtenido se representa en la Fórmula 4.2

$$T = 2{,}7855 * j * p * c^{-0{,}848} \tag{4.2}$$

donde las distintas variables representan:

- T: tiempo de ejecución aproximado.
- j: número de jaulas por partícula.
- p: número de partículas utilizadas.
- c: número de cores utilizados.

Este sería el modelo predictivo final que permitiría a las empresas tener una referencia en el tiempo de ejecución del programa paralelo. Además, se podría utilizar para calcular el número de cores necesario para un tiempo de ejecución buscado, muy útil a nivel empresarial para tener una aproximación de la inversión a realizar.

Capítulo 5

Conclusiones

En este capítulo final se mencionan los objetivos conseguidos con este trabajo, y a su vez las posibles mejoras que se podrían implementar en la paralelización del algoritmo AquiAID.

5.1. Objetivos Conseguidos

En esta sección se pretende argumentar los objetivos de la Sección 1.3 que se han cumplido a la hora de realizar este trabajo:

- *Mejorar la escalabilidad del programa*: este objetivo es claramente el más importante para los usuarios de AquiAID, y se ha completado con creces. Los experimentos del Capítulo 4 dan clara muestra de que, actualmente el programa es capaz de realizar casos reales. Con la versión paralela es posible realizar ejemplos entre 50 o 60 veces mayores a los conseguidos con la versión original. Además, de que el consumo energético no se vuelve un limitante a la hora de la ejecución paralela debido a la gran eficiencia energética obtenida con el programa paralelo. Con estas mejoras es posible que una empresa utilice este software de planificación para gestionar la cría de peces, sin la necesidad de esperar días para obtener un resultado. Esto hace al programa útil para el uso con el que nació, y permite tanto a las empresas como a los investigadores, seguir mejorando el largo camino de detalle y mejora de las aplicaciones de gestión de acuicultura.
- *Paralelización del código*: el algoritmo de PSO es uno de los problemas que permite una paralelización sencilla, pero debido a su implementación en el software de AquiAID, esta paralelización se ha podido realizar con un grano mucho más fino, aumentando las capacidades del software. Gracias al diseño expuesto en el Capítulo 3, se puede observar cómo se ha realizado esta implementación paralela de grano fino, complicando las paralelizaciones generales del algoritmo PSO. Aún con este grado de complejidad,

se ha conseguido obtener un programa que otorga grandes rendimientos en entornos distribuidos. Por todo esto, es posible concluir que la paralelización del código se ha realizado de manera correcta y muy optimizada.

- *Validación experimental*: como se ha mencionado en reiteradas ocasiones en el Capítulo 4, se pretendía realizar ejemplos con un número elevado de jaulas para demostrar que la implementación es capaz de ejecutar este tipo de experimentos. Y como se demuestra en este mismo capítulo es posible ejecutar de manera precisa experimentos con una cantidad considerable de jaulas para una empresa de acuicultura. Además, se demuestra mediante un análisis profundo que la implementación paralela ejecutada en estos entornos con gran número de jaulas, es capaz de tener un gran rendimiento tanto en cómputo como en energía.

Todos los objetivos planteados al inicio del documentos han servido para la mejora del programa de AquiAID. Actualmente el programa es capaz de ejecutarse en un entorno distribuido, aprovechando la capacidad de paralelización de estos sistemas obteniendo así resultados a la planificación de una empresa de acuicultura.

5.2. Trabajos Futuros

Existen ciertas posibles mejoras que se le pueden aplicar al software de AquiAID para mejorar la eficacia y eficiencia del programa. Las posibles mejoras son las siguientes:

- *Paralelización en GPU*: otro paradigma de paralelización de algoritmos es la utilización de GPUs. Las GPUs son mucho más potentes para el procesamiento paralelo que los cores normales, debido a su arquitectura específica de paralelización de datos. Es por esto, que un cambio posible a realizar es implementar este programa para la ejecución del algoritmo PSO en una GPU. En este proyecto no fue posible realizar esta implementación debido a la falta de medios, y sería muy importante comprobar que implementación paralela aporta más rendimiento, si con cores o con GPUs.

- *Balanceo dinámico*: en la mayoría de implementaciones paralelas de PSO, no es necesario realizar un balanceo dinámico, debido a que las partículas suelen tener una carga computacional muy parecida. Esto no es tan cierto en AquiAID, debido a que solo ciertos procesos cargan con esta carga de trabajo mientras que otros no lo hacen. Sería posible ir repartiendo este trabajo de cálculo de las partículas dinámicamente entre los distintos procesos. Además, esto permitiría la ejecución eficiente en sistemas paralelos heterogéneos, en los cuales, unos cores fueran más potentes que otros, o algunos nodos dispusiesen de GPUs y otros no.

Bibliografía

[1] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[2] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M Vidal. *Introducción a la programación paralela*. Thompson Paraninfo, 2008.

[3] John M Shalf and Robert Leland. Computing beyond moore's law. *Computer*, 48(12):14–23, 2015.

[4] George S Almasi and Allan Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., 1994.

[5] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[6] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[7] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

[8] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[9] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

[10] Cuda zone, Feb 2022.

[11] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[12] Kevin Dowd, Charles R Severance, and Michael Kosta Loukides. High performance computing. 1998.

[13] Emilio Castillo, Cristobal Camarero, Ana Borrego, and José Luis Bosque. Financial applications on multi-cpu and multi-gpu architectures. *J. Supercomput.*, 71(2):729–739, 2015.

[14] Oscar David Robles, José Luis Bosque, Luis Pastor, and Angel Rodríguez. Performance analysis of a CBIR system on shared-memory systems and heterogeneous clusters. In *Seventh International Workshop on Computer Architectures for Machine Perception (CAMP 2005), 4-6 July 2005, Palermo, Italy*, pages 309–314. IEEE Computer Society, 2005.

[15] Michael J Quinn. Parallel programming. *TMH CSE*, 526:105, 2003.

[16] Raúl Nozal, José Luis Bosque, and Ramón Beivide. Enginecl: Usability and performance in heterogeneous computing. *Future Gener. Comput. Syst.*, 107:522–537, 2020.

[17] Manuel Mora, Guisseppi A Forgionne, and Jatinder ND Gupta. *Decision making support systems: achievements, trends and challenges for*. IGI Global, 2002.

[18] Jason Papathanasiou, Nikolaos Ploskas, et al. Multiple criteria decision aid. *Methods, Examples and Python Implementations*, 136, 2018.

[19] Carlos Romero. *Teoría de la decisión multicriterio: conceptos, técnicas y aplicaciones.* Number 338 ROM. 1993.

[20] David Roi Hardoon and Galit Shmueli. *Getting started with business analytics: insightful decision-making*. CRC Press, 2013.

[21] Cindi Howson. *Business Intelligence: Estrategias para una implementación exitosa*. McGraw-Hill Interamericana, 2010.

[22] Galit Shmueli, Nitin R Patel, and Peter C Bruce. *Data mining for business intelligence: Concepts, techniques, and applications in Microsoft Office Excel with XLMiner*. John Wiley and Sons, 2011.

[23] Maite Cidad, Irene Peral, Saioa Ramos, Bernardo Basurco, Antonio López-Francos, Ana Muniesa, Marianna Cavallo, Jose Perez, Cristóbal Aguilera, Dolors Furones, et al. Assessment of mediterranean aquaculture sustainability. 2018.

[24] Aquiaid. `https://www.ides.unican.es/aquiaid-2/`. Accedido: 06-03-2022.

[25] ¿qué es la acuicultura? `https://www.observatorio-acuicultura.es/conocenos/que-es-la-acuicultura`. Accedido: 04-03-2022.

[26] Acuicultura: definición, historia, importancia y clasificación. `https://www.aquahoy.com/el-acuicultor/34373-acuicultura-definicion-historia-importancia-clasificacion`. Accedido: 04-03-2022.

[27] Acuicultura. `https://www.fao.org/aquaculture/es/`. Accedido: 04-03-2022.

[28] Austin Stankus. State of world aquaculture 2020 and regional reviews: Fao webinar series. *FAO Aquaculture Newsletter*, (63):17–18, 2021.

[29] James H Tidwell. *Aquaculture production systems*. John Wiley & Sons, 2012.

[30] Patrick Henry Winston. *Artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[31] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[32] Saptarshi Sengupta, Sanchita Basak, and Richard Alan Peters. Particle swarm optimization: A survey of historical and recent developments with hybridization perspectives. *Machine Learning and Knowledge Extraction*, 1(1):157–191, 2019.

[33] Mpi documents. `https://www.mpi-forum.org/docs/`. Accedido: 04-03-2022.

[34] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[35] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[36] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[37] Lisandro Dalcin and Yao-Lung L Fang. Mpi4py: Status update after 12 years of development. *Computing in Science & Engineering*, 23(4):47–54, 2021.

[38] Henrik Theodor Ramm and Alexander Wallem Berge. Fleet scheduling of service vessels used in a more exposed norwegian aquaculture industry. Master's thesis, NTNU, 2017.

[39] Run Yu, PingSun Leung, and Paul Bienfang. Optimal production schedule in commercial shrimp culture. *Aquaculture*, 254(1-4):426–441, 2006.

[40] Beverley M Wilson, Timothy L Shaftel, and Russell M Barefield. A mathematical programming approach to production decisions in the emerging aquaculture industry. *Decision Sciences*, 22(1):194–205, 1991.

[41] Timothy L Shaftel and Beverley M Wilson. A mixed-integer linear programming decision model for aquaculture. *Managerial and Decision Economics*, 11(1):31–38, 1990.

[42] Run Yu, PingSun Leung, Lotus E Kam, and Paul Bienfang. A decision support system for scheduling partial harvesting in aquaculture. In *Decision Support Systems in Agriculture, Food and the Environment: Trends, Applications and Advances*, pages 406–419. IGI Global, 2010.

[43] Xiaowei Zhou. An overview of recently published global aquaculture statistics. *FAO Aquaculture Newsletter*, (56):6, 2017.

[44] Francisco Vergara-Solana, Marcelo E Araneda, and German Ponce-Díaz. Opportunities for strengthening aquaculture industry through multicriteria decision-making. *Reviews in aquaculture*, 11(1):105–118, 2019.

[45] Kanokpatch Wonginyoo, Pachara Chatavithee, and Jumpol Vorasayan. Integration of process planning and scheduling for aquaculture. *Veridian E-Journal, Silpakorn University (Humanities, Social Sciences and arts)*, 11(4):374–389, 2018.

[46] José Fernández-Polanco and Ladislao Luna. Factors affecting consumers'beliefs about aquaculture. *Aquaculture Economics & Management*, 16(1):22–39, 2012.

[47] José Fernández-Polanco, Simone Mueller Loose, and Ladislao Luna. Are retailers'preferences for seafood attributes predictive for consumer wants? results from a choice experiment for seabream (sparus aurata). *Aquaculture Economics & Management*, 17(2):103–122, 2013.

[48] Ignacio Llorente and Ladislao Luna. Bioeconomic modelling in aquaculture: an overview of the literature. *Aquaculture International*, 24(4):931–948, 2016.

[49] Manuel Luna, Ignacio Llorente, and Angel Cobo. Aquaculture production optimisation in multi-cage farms subject to commercial and operational constraints. *Biosystems Engineering*, 196:29–45, 2020.

[50] A Cobo, I Llorente, and L Luna. Swarm intelligence in optimal management of aquaculture farms. In *Handbook of Operations Research in Agriculture and the Agri-Food Industry*, pages 221–239. Springer, 2015.

[51] Yuhui Shi et al. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 81–86. IEEE, 2001.

[52] Riccardo Poli. An analysis of publications on particle swarm optimization applications. *Essex, UK: Department of Computer Science, University of Essex*, 2007.

[53] Andrew W McNabb, Christopher K Monson, and Kevin D Seppi. Parallel pso using mapreduce. In *2007 IEEE Congress on Evolutionary Computation*, pages 7–14. IEEE, 2007.

[54] Cheng-Tao Chu, Sang Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Ng. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19, 2006.

[55] Narjess Dali and Sadok Bouamama. Gpu-pso: parallel particle swarm optimization approaches on graphical processing unit for constraint reasoning: case of max-csps. *Procedia Computer Science*, 60:1070–1080, 2015.

[56] Jaspreet Kaur, Satvir Singh, and Sarabjeet Singh. Parallel implementation of pso algorithm using gpgpu. In *2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)*, pages 155–159. IEEE, 2016.

[57] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[58] Mysql. `https://www.mysql.com/`. Accedido: 06-03-2022.

[59] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[60] The python profilers. `https://docs.python.org/3/library/profile.html`. Accedido: 04-03-2022.

[61] Luis Pastor and José Luis Bosque. An efficiency and scalability model for heterogeneous clusters. In *2001 IEEE International Conference on Cluster Computing (CLUSTER 2001), 8-11 October 2001, Newport Beach, CA, USA*, pages 427–434. IEEE Computer Society, 2001.

[62] José Luis Bosque and L. P. Perez. Theoretical scalability analysis for heterogeneous clusters. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*, pages 285–292. IEEE Computer Society, 2004.

[63] James H Laros III, Kevin Pedretti, Suzanne M Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. Energy delay product. In *Energy-Efficient High Performance Computing*, pages 51–55. Springer, 2013.

[64] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[65] Ioan Cristian Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information processing letters*, 85(6):317–325, 2003.

[66] Daniel Bratton and James Kennedy. Defining a standard for particle swarm optimization. In *2007 IEEE swarm intelligence symposium*, pages 120–127. IEEE, 2007.

[67] Yuhui Shi and Russell C Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1945–1950. IEEE, 1999.

[68] Zhang Li-Ping, Yu Huan-Jun, and Hu Shang-Xu. Optimal choice of parameters for particle swarm optimization. *Journal of Zhejiang University-Science A*, 6(6):528–534, 2005.

[69] Adam P Piotrowski, Jaroslaw J Napiorkowski, and Agnieszka E Piotrowska. Population size in particle swarm optimization. *Swarm and Evolutionary Computation*, 58:100718, 2020.

[70] José Luis Bosque, Pablo Toharia, Oscar David Robles, and Luis Pastor. A load index and load balancing algorithm for heterogeneous clusters. *J. Supercomput.*, 65(3):1104–1113, 2013.

[71] Extrae instrumentation package. `https://tools.bsc.es/extrae`. Accedido: 07-04-2022.

[72] Michael Wagner, Germán Llort, Estanislao Mercadal, Judit Giménez, and Jesús Labarta. Performance analysis of parallel python applications. *Procedia Computer Science*, 108:2171–2179, 2017.

[73] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. Citeseer, 1995.

[74] Esteban Stafford and José Luis Bosque. Performance and energy task migration model for heterogeneous clusters. *J. Supercomput.*, 77(9):10053–10064, 2021.

[75] Esteban Stafford and José Luis Bosque. Improving utilization of heterogeneous clusters. *J. Supercomput.*, 76(11):8787–8800, 2020.