TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DE COMPUTADORES

Estratexias de tolerancia a fallos nun algoritmo paralelo de colonia de formigas

Estudante: Miguel Blanco Godón

Dirección: Patricia González Gómez

María José Martín Santamaría

A Coruña, setembro de 2022.

A meus pais e avós, que sempre me axudaron e deron todo o necesario para acadar os obxectivos da miña formación.

Agradecementos

Ás titoras deste traballo por axudarme e guiarme na realización do mesmo. Ao grupo de Arquitectura de Computadores da Universidade da Coruña por darme acceso ao equipamento necesario para a realización deste traballo.

Resumo

A resolución de problemas combinatorios de complexidade NP aínda é un problema aberto debido ao alto tempo de computación requirido para obter solucións de boa calidade. Dentro deste conxunto, o problema do viaxante - *Travel Salesman Problem (TSP)* - é un dos máis estudados pola comunidade científica debido á grande cantidade de aplicacións no mundo real que podemos atopar. Co obxectivo de obter solucións óptimas de xeito eficiente, unha das estratexias máis prometedoras é o uso de algoritmos de colonia de formigas - *Ant Colony Optimization (ACO)* - e variacións deste baseados no uso de metaheurísticas. Non tardaron en aparecer solucións paralelas para este algoritmo para intentar acelerar a computación da solución aínda máis. Non obstante, o aumento do número de recursos utilizados nas aplicacións e sistemas paralelos, conleva ao aumento na taxa de fallos aos que están expostos estas aplicacións tan demandantes, polo que utilizar técnicas de tolerancia a fallos vólvese fundamental. O obxectivo deste proxecto é estudar diferentes estratexias de tolerancia a fallos para unha versión paralela asíncrona do ACO, utilizando as extensións de MPI que proporciona o framework *User Level Failure Mitigation (ULFM)*. Esta proposta foi avaliada utilizando dous problemas TSP distintos extraídos da libraría TSPLib, obtendo tolerancia dende 1 a $N-1$ fallos (sendo N o número de procesos MPI) cun overhead moi pequeno na maior parte dos casos.

Abstract

Solving combinatorial problems of NP complexity is still an open problem due to the high computational time required to obtain good quality solutions. Within this set, *Travel Salesman Problem (TSP)* is one of the most studied by the scientific community due to the large number of applications in the real world that we can find. With the aim of obtaining optimal solutions efficiently, one of the most promising strategies is the use of ant colony algorithms - *Ant Colony Optimization (ACO)* - and variations of it based on the use of metaheuristics. Parallel solutions to this algorithm soon appeared in an attempt to speed up the computation of the solution even more. However, the increase in the number of resources used in parallel applications and systems leads to an increase in the rate of failures to which these demanding applications are exposed, so using fault tolerance techniques becomes essential. The objective of this project is to study different fault tolerance strategies for an asynchronous parallel version of the ACO, using the MPI extensions provided by the *User Level Failure Mitigation (ULFM)* framework. This proposal was evaluated using two TSP problems taken from the

TSPLib library, obtaining tolerance from 1 to $N-1$ failures (being N the number of MPI processes) with a very small overhead in most cases.

Palabras chave:

- ACO
- MPI
- ULFM
- Tolerancia a fallos
- Computación paralela

Keywords:

- ACO
- MPI
- ULFM
- Fault tolerance
- Parallel computing

Índice Xeral

1 Introdución **1**

1.1 Antecedentes e obxectivos do proxecto 1
1.2 Recursos e fases de desenvolvemento 2
1.3 Estrutura da memoria 3

2 Estado do arte e conceptos previos **4**

2.1 Clases de complexidade 4
2.2 O problema do viaxante 5
2.2.1 Solución exacta 5
2.2.2 Solucións aproximadas 6
2.2.3 Aplicacións 7
2.3 Ant Colony Optimization (ACO) 7
2.4 Message-Passing Interface (MPI) 10
2.4.1 Implementacións de MPI 10
2.4.2 Conceptos básicos de MPI 11
2.5 Paralelización do algoritmo ACO 12
2.6 Alta dispoñibilidade e tolerancia a fallos 16
2.6.1 Garantía de funcionamento 16
2.6.2 Estratexias para aumentar a GdF 18
2.6.3 Alta dispoñibilidade 19
2.6.4 Modelización de fallos en sistemas informáticos 19
2.6.5 Métricas 19
2.7 User Level Failure Mitigation (ULFM) 20
2.7.1 Novos códigos de erros 21
2.7.2 Novas funcións de control e propagación de erros 21

3 Planificación e custos **23**

3.1 Planificación temporal do proxecto 23

3.1.1 Planificación inicial 23

3.2 Avaliación de custos 24

3.3 Seguimento do proxecto e esforzo real 24

4 Implementación **27**

4.1 Instalación de ULFM e compilación con tolerancia a fallos 27

4.2 Fluxo de execución dun programa tolerante a fallos en ULFM 30

4.2.1 Detección de fallos en MPI 30

4.2.2 Procesamento dos fallos con UFLM 31

4.3 Estudo das solucións xenéricas de ULFM 31

4.3.1 MPI_ERRORS_ARE_FATAL 31

4.3.2 MPI_ERRORS_RETURN 32

4.4 *Handlers* customizados desenvolvidos polos usuarios 33

4.4.1 FT_ABORT_ON_FAILURE_HANDLER 34

4.4.2 FT_IGNORE_ON_FAILURE_HANDLER 36

4.4.3 FT_RESPAWN_ON_FAILURE_HANDLER 39

4.4.4 FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER 44

4.5 Adaptación do código para o uso dos *handlers* customizados 45

4.5.1 Inicialización da libraría e dos *handlers* 45

4.5.2 Consideracións xerais 48

5 Resultados experimentais **50**

5.1 Sistema utilizado nas probas 50

5.2 Metodoloxía utilizada nas probas 51

5.2.1 Inxección de fallos 53

5.2.2 Saída de datos e tratamento estatístico 54

5.3 Análise de resultados 55

5.3.1 Análise de resultados para o problema *lin318* 56

5.3.2 Análise de resultados para o problema *rat783* 58

6 Conclusións e liñas futuras **63**

6.1 Conclusións sobre a tecnoloxía subxacente 64

6.2 Relación coa titulación 65

6.3 Liñas futuras 65

Bibliografía **67**

Índice de Figuras

2.1 Funcionamento da estigmerxia. 8
2.2 Relación causa-efecto das amezadas da GdF dun sistema 17

3.1 Diagrama de Gantt da planificación inicial do proxecto 25

4.1 Fluxo de execución dun programa tolerante fallos 31

5.1 Arquitectura de Pluton 51
5.2 Dispersión dos resultados por experimento para o problema *lin318* 61
5.3 Dispersión dos resultados por experimento para o problema *rat783* 62

Índice de Táboas

2.1 Explosión combinatoria cando se tenta resolver o TSP mediante forza bruta . . 6

3.1 Planificación inicial. Valores estimados 23
3.2 Valores reais de esforzo no proxecto 26

5.1 Especificacións técnicas da cabina 0 utilizada para as probas 52
5.2 Características dos *benchmarks* utilizados nas probas 55
5.3 Resultados para os experimentos con parada por calidade (óptimo=49029) para o problema *lin318* 58
5.4 Resultados para os experimentos con parada por calidade (óptimo=8806) para o problema *rat783* 59

Capítulo 1

Introdución

ESTE capítulo expón unha breve descrición do proxecto, incluíndo os seus obxectivos principais, así como as fases de desenvolvemento do mesmo e o formato desta memoria.

1.1 Antecedentes e obxectivos do proxecto

A metaheurística denominada Optimización de Colonia de Formigas (Ant Colony Optimization – ACO) [1] é un dos métodos empregados na resolución de problemas combinatorios de complexidade NP [2]. Existen multitude de variantes do algoritmo, pero neste traballo o obxectivo é estudar a incorporación de estratexias de tolerancia a fallos nunha versión paralela concreta que usa MPI para a comunicación asíncrona entre as diferentes colonias que cooperan no algoritmo. O código resultante aplicarase á resolución do popular Problema do Viaxante (Travel Salesman Problem – TSP) [3], comunmente utilizado como conxunto de probas para os novos algoritmos asociados á resolución de problemas combinatorios. O TSP presenta unha grande cantidade de aplicacións no mundo real, o que o converteu nun dos problemas máis estudados pola comunidade científica mundial. A maioría destes problemas precisan de moito tempo de computación para chegar a boas solucións, por iso as versións paralelas dos métodos de resolución están gañando popularidade. Non obstante, o aumento do número de recursos utilizados nas aplicacións e sistemas paralelos conleva ao aumento na taxa de fallos aos que están expostos estas aplicacións tan demandantes, polo que utilizar técnicas de tolerancia a fallos vólvese fundamental.

Para incorporar tolerancia a fallos faremos uso de ULFM (User Level Failure Mitigation) [4][5]. ULFM é a iniciativa máis recente proposta polo Grupo de Traballo en Tolerancia a Fallos do Foro de MPI para habilitar a tolerancia a fallos no estándar MPI [6]. ULFM inclúe novas semánticas para a detección de fallos e revogación e reconfiguración de comunicadores, o que permite a implementación de aplicacións MPI resilientes, é dicir, capaces de detectar e reac-

cionar a fallos sen deter a súa execución.

Para acadar este obxectivo partirase do código paralelizado do ACO aplicado ao TSP implementado en linguaxe C polo Grupo de Arquitectura de Computadores (GAC) da Universidade da Coruña (UDC) [7][8]. Este, á súa vez, parte da implementación secuencial do ACO desenvolvido por Marco Dorigo e Thomas Stützle [1].

1.2 Recursos e fases de desenvolvemento

O proxecto foi desenvolvido e probado no clúster *Pluton* [9] da Universidade da Coruña, administrado polo Grupo de Arquitectura de Computadores (GAC), tamén da Universidade da Coruña. Utilizouse Git como ferramenta de control de versións no repositorio indicado en [10].
As fases de desenvolvemento do proxecto foron as seguintes:

1. **Estudo do ACO:** estudo detallado do ACO aplicado ao problema do TSP, principalmente a través de [1] e [8].

2. **Estudo de ULFM:** estudo do funcionamento da solución experimental de tolerancia fallos do Grupo de Traballo en Tolerancia a Fallos do Foro de MPI para habilitar a tolerancia a fallos no estándar MPI. Para isto dispúxose da (moi escasa) documentación preliminar da proposta para o estándar, así como material expositivo presentado polo grupo no congreso *SC'20* [11].

3. **Implementación dunha solución tolerante a fallos:** presentación, partindo do código paralelo para o ACO aplicado ao TSP desenvolvido polo GAC, dunha posible solución tolerante a fallos e implementación da mesma, tentando que sexa o máis desacoplada do problema en cuestión, para que poida ser fácilmente reutilizada por outros problemas semellantes.

4. **Proba de funcionamento:** conxunto de probas deseñadas para comprobar o nivel de tolerancia a fallos da solución implementada.

5. **Repetición de *3* e *4* para todas as alternativas:** proposta de outras posibles solucións e estudo, implementación e proba das mesmas.

6. **Avaliación exhaustiva de resultados:** conxunto de probas deseñadas para comprobar o impacto no rendemento da aplicación destas técnicas a diversos problemas do TSP.

7. **Documentación de resultados:** proceso de elaboración desta memoria.

Cada unha destas fases foi executada en orde cronolóxico. Como se pode observar, para o desenvolvemento do sistema seguiuse un ciclo de vida incremental, onde en cada incremento se presentou unha nova solución tolerante a fallos do mesmo.

1.3 Estrutura da memoria

Este documento está dividido en seis capítulos (incluíndo este), co seguinte contido:

- **Capítulo 1**: breve resumo dos obxectivos, recursos e fases do proxecto.
- **Capítulo 2**: introducción a conceptos previos necesarios para a comprensión deste proxecto. Máis concretamente, trátanse certos temas de complexidade computacional, o problema do viaxante, os algoritmos ACO, MPI e ULFM e a enxeñaría da fiabilidade.
- **Capítulo 3**: planificación do proxecto e custos. Trátase tanto as estimacións iniciais como o seguemento do proxecto e os valores reais finais.
- **Capítulo 4**: explicación detallada das solucións desenvolvidas.
- **Capítulo 5**: avaliación dos resultados experimentais e explicación da metodoloxía aplicada nas probas realizadas.
- **Capítulo 6**: conclusións finais extraídas do desenvolvemento deste proxecto e posibles liñas futuras de traballo.

Capítulo 2

Estado do arte e conceptos previos

NESTE capítulo describiranse os conceptos e tecnoloxías necesarias para a comprensión do traballo realizado.

2.1 Clases de complexidade

Defínense como problemas clase P [12] aqueles problemas que se poden resolver cunha máquina de Turing determinista nun número de pasos acotados superiormente por unha función polinómica da forma $x = O(n^k)$.

Defínense como problemas de clase NP [12] aqueles problemas que se poden resolver cunha máquina de Turing non determinista nun número de pasos acotados superiormente por unha función polinómica da forma $x = O(n^k)$.

Calquer problema NP pódese resolver cunha máquina de Turing determinista, pero non nun tempo polinómico con respecto ao número de pasos, xa que a única diferenza entre unha máquina determinista e unha non determinista é que a primeira evalúa un camiño en cada paso, mentres que a segunda é capaz de executar varios camiños ao mesmo tempo.

Problemas clase NP-hard

Defínese como un problema NP-hard [12] aquel que é polo menos tan complicado de resolver coma un problema NP, é dicir, que se conseguimos resolver un problema deste tipo (H) nun número $O(n^k)$ de pasos usando certo algoritmo A, entón poderemos resolver calquer problema NP nun número polinómico de pasos facendo unha redución dese problema en H e aplicando o algoritmo A.

Problemas clase NP completos

Defínese un problema X como un problema NP-completo se cumpre as dúas condicións seguintes:

- $X \in NP$.
- calquera problema $Y \in NP$ é reducible a X nun número de pasos polinomial.

2.2 O problema do viaxante

O problema do viaxante (*Travel Salesman Problem, TSP*, en inglés) é un problema de optimización formulado no século XIX polo matemático Thomas Kirkman, pero non foi propiamente estudado ata os anos 30 do século XX. Este problema pertence á clase de complexidade NP-completo, polo que tamén é un problema NP-hard.

Formalmente o problema defínese como segue [3]:

”Para un conxunto finito de puntos P, atopar a ruta máis curta que conecta todos os puntos, sabendo que para dous puntos calquera $p_i \in P$ e $p_{i+1} \in P$, a distancia entre p_i e p_{i+1} é coñecida de antemán.”

onde no caso do TSP, os puntos serían as cidades que terían que ser visitadas polo viaxante.

Se tentamos resolver o problema con N cidades, teremos $N!$ posibles rutas. Con todo, no problema non se definen restriccións sobre en qué cidade ten que comezar ou en cal debe de rematar o viaxante, polo que só teríamos que avaliar un camiño partindo dunha cidade $c_i \in C$, sendo C o conxunto total de cidades. Deste xeito podemos reducir o número de posibles rutas a avaliar nun factor de N. Como tampouco importa a dirección na que se mova o viaxante, tamén reducimos o número posible de rutas a avaliar por un factor de 2, polo que só habería que calcular

$$\frac{N!}{2N} = \frac{(N-1)!}{2} \tag{2.1}$$

rutas.

2.2.1 Solución exacta

A solución exacta poderíase obter por forza bruta, calculando todas as rutas, cunha complexidade computacional $O\left(\frac{(N-1)!}{2}\right)$. Para problemas arbitrariamente grandes, esta solución non é valida, debido que se produce unha explosión combinatoria. Isto fai o problema intratable mediante forza bruta excepto para conxuntos de cidades moi pequenos, como se pode

Cidades a visitar	Rutas totales a calcular
4	3
5	12
10	181440
20	$6.08 \cdot 10^{16}$
50	$3.04 \cdot 10^{62}$
100	$4.67 \cdot 10^{155}$

Táboa 2.1: Explosión combinatoria cando se tenta resolver o TSP mediante forza bruta

observar na táboa 2.1. As outras solucións exactas inclúen o uso de algoritmos de ramificación e poda, así como da programación lineal, pero en ningún caso se consegue evitar a explosión combinatoria e a solución de grandes conxuntos de puntos pode tardar varios anos de tempo de CPU.

2.2.2 Solucións aproximadas

Debido a que os problemas son intratables para cando temos unha grande cantidade de puntos, optouse por obter solucións aproximadas, cambiando o obxectivo a acadar: en vez de obter o mínimo camiño, tentarase obter o camiño máis próximo posible ao mínimo cun tempo razoable, o que obrigará a facer concesións tanto en tempo de cómputo como en precisión.

Algoritmos de heurísiticas construtivas

Unha heurística construtiva é un método heurístico onde a solución comeza sendo un conxunto baleiro que se extende repetidamente ata atopar unha solución completa. Un exemplo de algoritmo que usa este método é o algoritmo do veciño máis próximo (*Nearest Neighbour Algorithm*, NNA). Este é un algoritmo voraz que se basea en escoller sempre como seguinte cidade a visitar aquela que estea máis cerca con respecto ao punto fronteira entre os visitados e os non visitados. Esta peculiaridade do algoritmo permite obter solucións rápidamente, pero pouco óptimas: de media, devolve un camiño 25% mais longo do óptimo, e en moitos casos a distribución das cidades provoca que o sistema devolva o peor camiño posible [13].

Algoritmos de tipo ACO

Método heurístico que xera boas solucións mediante o uso de colonias virtuais de formigas que emulan o funcionamento dos procesos biolóxicos de estigmerxia vencellados ás feromonas das formigas. Este algoritmo é o utilizado neste proxecto e explícase con profundidade

máis adiante neste capítulo.

2.2.3 Aplicacións

O TSP é utilizado como benchmark de algoritmos de optimización, xa que é un problema NP-hard e está moi estudado, dando lugar a conclusións xeralmente extrapolables a outros problemas de optimización semellantes.

Ademais do anterior, este problema (ou variantes do mesmo) presentan un gran número de aplicacións no mundo real. Algúns exemplos son os seguintes:

- **Loxística:** a distribución de recursos pódese reducir a atopar o camiño ou xeito máis barato, rápido ou eficiente de chegar ás localizacións de distribución, debido a que a propia distribución debe realizarse físicamente. Neste caso, teríamos unha aplicación directa do problema, onde só teríamos que engadir certas restriccións (prioridades á hora de visitar unha localización antes ca outra, camiños de peso variable en función de factores externos, etc) para adaptalo ao problema real.

- **Industria electrónica:** cada vez buscamos obter circuítos electrónicos máis compactos e baratos. A interconexión dos distintos compoñentes no PCB (*Printed Circuit Board*) pode dar lugar a dispositivos máis grandes e caros se non se obteñen solucións de enrutamento entre os compoñentes válidas. Hoxe en día, o software para a fabricación destes dispositivos permite o enrutado automático dos camiños buscando minimizar o uso de material e, por ende, o tamaño do dispositivo. Este enrutado pódese ver como un problema do viaxante, considerando a PCB como unha malla de elementos, onde cada pin ou punto de interconexión dos compoñentes se pode considerar como unha cidade. O enrutado final obteríase mediante a resolución de múltiples TSP, 1 por conxunto de puntos a interconectar.

- **Xenética:** en problemas de secuenciación, poderíamos considerar cidades os fragmentos de ADN a secuenciar, e a distancia entre eses fragmentos sería a similitude entre os nucleótidos de cada fragmento.

2.3 Ant Colony Optimization (ACO)

O algoritmo de optimización por colonia de formigas (ACO) é un algoritmo de optimización cun enfoque probabilístico proposto por Marco Dorigo na súa tese *Optimization, Learning and Natural Algorithms, 1992* [14].

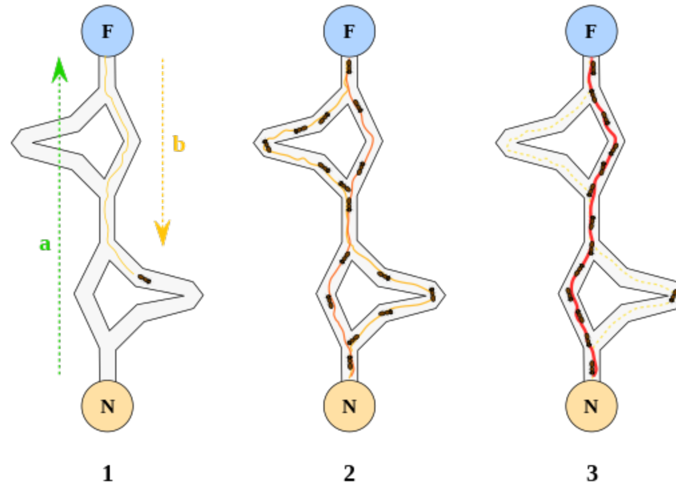

Figura 2.1: Funcionamento da estigmerxia.

Bases do algoritmo

A técnica baséase na replicación artificial do comportamento das colonias de formigas, posto que estes grupos en conxunto conseguen atopar os camiños mínimos entre os formigueiros e os puntos de exploración ou nos que se alimentan. Para permanecer a menor cantidade de tempo posible nun entorno hostil, as formigas desenvolveron o mecanismo da *estigmerxia*. Este é un método de marcaxe biolóxica a través de feromonas, que son depositadas nas superficies polas que pasan os individuos dunha colonia. Os individuos veranse fortemente atraídos ás concentracións de feromonas dunha superficie, polo que canta máis acumulación de feromona, máis formigas pasarán por un camiño. Mediante este método, obterán iterativamente os camiños máis curtos entre dous puntos, xa que a densidade de feromonas dunha superficie aumentará co paso das formigas, e os camiños máis curtos chegarán a picos de feromonas antes, posto que as formigas realizarán o percorrido en menor tempo. Esta estratexia é un perfecto exemplo de como realizar tarefas cooperativas sen necesidade dunha planificación previa ou un poder central. Na figura 2.1 amósase un exemplo do funcionamento da estigmerxia.

Implementación

A implementación de ACO baséase en realizar buscas locais onde en cada paso se van actualizando as feromonas, para simular o paso dos elementos da colonia polos diferentes camiños [1] [8] [7].

```
void do_aco(...)
{
    while (!is_termination_condition_met(...))
    {
        construct_ant_solutions(...);
        apply_some_search_method(...); // optional
        update_pheromones(...);
    }
}

void construct_ant_solutions(...)
{
    for (int k = 0; k < n_ants; k++)
    {
        do_mark_all_cities_as_unvisited(...);
        // place ants on the same initial city
        place_ant(...);
        while (!are_all_cities_visited(...))
        {
            choose_next_move_and_move(...);
        }
        // compute tour length for every ant
        do_compute_tour_length(...);
    }
}

void apply_some_search_method(...)
{
    // implementation dependant
}

void update_pheromones(...)
{
    do_compute_pheromone_evaporation(...);
    evaporate_pheromone_deposit(...);
    do_apply_new_pheromone_deposit(...);
}
```

Como se pode observar no pseudocódigo anterior, este algoritmo está composto fundamentalmente por tres procedementos:

- `construct_ant_solutions`: xestiona unha colonia de formigas as cales visitan estados adxacentes do problema considerado movéndose a través de nodos veciños aos previamente visitados. Os nodos represéntanse en memoria como un grafo. O movemento das formigas calcúlase mediante un proceso de decisión estocástico en función

das trazas de feromonas e da información heurística asociada a cada nodo (ou conxunto de nodos) do grafo asociado ao problema a resolver. Cando unha formiga remata de calcular o seu camiño, avalía a solución obtida para decidir canta concentración de feromonas será depositada e a distrubución da mesma a través do camiño calculado.

- `apply_some_search_method`: opcionalmente, pódese executar algún método de busca local para tentar mellorar os camiños calculados no procedemento anterior.
- `update_pheromones`: encárgase de modificar a concentración de feromonas. Esta pode aumentar, pois as formigas depositan feromonas nos nodos e conexións que visitan, ou diminuír, debido á evaporación de feromonas. Téntase simular a evaporación que ocorriría naturalmente no proceso biolóxico equivalente. Neste caso, a evaporación consiste en reducir a concentración de feromonas en función dun parámetro configurable polo usuario. Canto maior sexa a evaporación, máis se benefician os camiños máis transitados con respecto aos camiños menos percorridos.

2.4 Message-Passing Interface (MPI)

MPI (*Message-Passing Interface*) é a especificación dunha libraría de paso de mensaxes para un modelo de programación paralelo que permite a comunicación de datos entre os espazos de memoria de dous procesos a través de operacións cooperativas entre ambos [6]. Mediante esta especificación conséguese que os programas implementados con MPI sexan portables entre distintos sistemas, así como escalables en función do tamaño do sistema a utilizalos. Por isto MPI é actualmente o estándar no campo da computación de altas prestacións (*High Performance Computing, HPC*) en sistemas de memoria distribuída.

2.4.1 Implementacións de MPI

As implementacións de MPI consisten xeralmente nun conxunto de rutinas que poden ser directamente utilizadas nos linguaxes *C*, *C++* e *Fortran*, así como calquera outra linguaxe capaz de interactuar coas bibliotecas dos antes mencionados, como *C#*, *Java* e *Python*. As implementacións de MPI agrúpanse en librarías, contendo todo o necesario para que o protocolo funcione en calquera sistema soportado. As máis coñecidas son *MPICH* [15] e *OpenMPI* [16]. Diferentes versións das librarías poden ser compiladas para diferentes arquitecturas, sistemas e configuracións, asegurando a interoperabilidade do código entre sistemas heteroxéneos, así como garantindo o máximo rendemento ao compilalas directamente para o hardware que interesa utilizar.

2.4.2 Conceptos básicos de MPI

Defínense aquí algúns conceptos básicos de MPI que se usarán na descrición da solución proposta:

Grupos

Un grupo en MPI é un conxunto ordenado de identificadores de procesos. A orde dos procesos será definida polo rango dos mesmos.

Comunicadores

Un comunicador é un obxecto que encapsula toda a información necesaria para a comunicación dun conxunto de procesos. Pódense comunicar múltiples conxuntos de procesos mediante o uso de *intra-comunicadores*, os cales son comunicadores entre dous conxuntos disxuntos de procesos (sen solapamento de procesos entre os conxuntos). En MPI hai un comunicador por defecto, `MPI_COMM_WORLD`, o cal comunica a todos os procesos que se lanzan na execución do programa. Os usuarios da libraría poden crear, destruír e asociar comunicadores entre os procesos libremente a través das primitivas MPI definidas para tal efecto, pero o manexo dos múltiples comunicadores é responsabilidade do usuario.

Rango dun proceso

O rango dun proceso é un número enteiro que identifica os procesos que se executan dentro dun comunicador. Ese número será único para cada proceso do comunicador e estará comprendido no rango $[1, N-1]$, onde N é o número de procesos asociados a un grupo. En caso de que algún proceso pertenza a máis dun grupo, terá un rango propio en cada un dos grupos, e non necesariamente terán o mesmo valor.

Operacións punto a punto

As operacións punto a punto son aquelas nas cales se realiza un trasvase de datos entre dous procesos en MPI que pertenzan ao mesmo comunicador. Para estas operacións será obrigatorio especificar qué proceso será o que envíe os datos e cal será o que os reciba. Esta especificación realizarase mediante o rango. Tamén se lle pode asociar unha etiqueta (*tag* ou *message id*) a cada mensaxe, para poder discernir os distintos mensaxes de distintas ”conversas”. Un proceso só procesará as mensaxes cuxa etiqueta se corresponda coa configurada na operación de recepción. O resto das mensaxes serán enviadas a un buffer, e procesaranse cando as etiquetas coincidan.

Operacións colectivas

As operacións colectivas son aquelas que involucran a todos os procesos dun comunicador. Defínen un proceso raíz (*root*) que se encargará de coordinar a operación, típicamente será o que envíe e/ou reciba os datos necesarios para levar a cabo a mesma.

Operacións bloqueantes

As operacións bloqueantes son aquelas que non permiten a un proceso seguir o fluxo do programa ata que a comunicación definida na operación se complete. Isto pode levar a problemas como o interbloqueo dos procesos en caso de que algo falle na comunicación.

Operacións non bloqueantes

As operacións non bloqueantes son aquelas que permiten a un proceso seguir o fluxo do programa antes de que a comunicación definida na operación se complete. Con estas operacións non se van a producir bloqueos, pero queda como responsabilidade do usuario a comprobación da validez dos datos transmitidos na operación.

Tipos de datos en MPI

Os tipos de datos son obxectos que conteñen a información (xeralmente o tamaño en bytes de cada tipo, dependente da arquitectura) sobre as características físicas dos datos a enviar nas operacións en MPI. Estes están definidos como constantes en MPI e son da forma *MPI_...*. Cada implementación de MPI terá uns valores distintos destes tipos de datos dependendo da arquitectura do sistema e o hardware que execute o programa MPI. Por ese motivo é necesario utilizar estes tipos de datos para que os programas sexan portables.

2.5 Paralelización do algoritmo ACO

O código [7] do algoritmo ACO foi paralelizado polo departamento de Arquitectura de Computadores (GAC) da Universidade da Coruña. A parelización aplicada ten un enfoque híbrido, utilizando tanto MPI como *OpenMP* [17]. Ao contrario ca con *MPI*, que se encarga da comunicación entre procesos, *OpenMP* encárgase da paralelización mediante o uso da memoria compartida por múltiples fíos de execución dentro do mesmo proceso.
A paralelización con *OpenMP* fíxose aproveitando as opcións de compilación do linguaxe *C*, a través do uso de sentenzas *#pragma* para acelerar a computacion nos bucles. Cando o compilador dectecta estas sentenzas, busca dependencias en cada iteración do bucle, así como entre iteracións, para intentar o máximo número de operacións posibles concorrentemente. Tamén

intenta separar o espazo de computación, de xeito que cada fío só deba realizar o cálculo para un conxunto reducido de datos. Máis concretamente, aplicouse nos bucles que constrúen o conxunto solución e realizan as buscas locais, pois presentan unha complexidade igual ou inferior a $O(n^2)$, sendo n o número de elementos do espazo de busca. Se ben pode que a complexidade non diminúa co uso da parelización, si que se vai dividir o espazo, n, polo que o tempo real de computación, t_c, podería verse reducido ata $\frac{t_c}{p}$, sendo p o número de fíos involucrados. A continuación pódese ver un exemplo co pseudocódigo da implementación realizada [8]:

```
void construct_solutions(...)
{
#pragma omp parallel for private(k,step) schedule(guided)
    for ( k = 0 ; k < n_ants ; k++) {
        step=0;
        /* Mark all cities as unvisited */
        ant_empty_memory( &ant[k] );
        /* Place the ants on same initial city */
        place_ant( &ant[k], step);

        while ( step < n-1 ) {
      step++;
      neighbour_choose_and_move_to_next( &ant[k], step);
        }
        step = n;
        ant[k].tour[n] = ant[k].tour[0];
        ant[k].tour_length = compute_tour_length( ant[k].tour );
    }
}

void local_search(...)
{
    #pragma omp parallel for schedule(guided)
    for ( k = 0 ; k < n_ants ; k++ ) {
  switch (ls_flag) {
        case 1:
      two_opt_first( ant[k].tour );    /* 2-opt local search */
            break;
        case 2:
      two_h_opt_first( ant[k].tour );  /* 2.5-opt local search */
            break;
        case 3:
      three_opt_first( ant[k].tour );  /* 3-opt local search */
            break;
        default:
```

```
36         fprintf(stderr,"type of local search procedure not correctly
       specified\n");
37         exit(1);
38     }
39     ant[k].tour_length = compute_tour_length( ant[k].tour );
40   }
41 }
```

Deste xeito, conseguimos unha paralelización efectiva en tempos de cálculo multicolonia (supoñemos para efectos deste traballo que 1 proceso representa unha colonia de inviduos para a execución do algoritmo).
Por outra banda, no tocante a MPI temos un enfoque multicolonia. Neste caso, tentarase acelerar a converxencia do algoritmo mediante a compartición dos mellores resultados acadados por cada colonia individual coas outras. Como cada colonia pode funcionar coma un ente illado, a paralelización lévase a cabo mediante operacións non bloqueantes para evitar perder tempo ou bloquear colonias tras cada iteración do algoritmo. Máis concretamente, realízanse envíos e recepcións asíncronos para transmitir o mellor resultado de cada colonia. Despois, cada colonia actualizará os seus valores en función dos resultados obtidos polas outras e, de ser mellores, recalculará as novas concentracións de feromonas a partires dos datos obtidos polas colonias veciñas. Para este problema en concreto préstase moi ben este enfoque asíncrono da paralelización, posto que cada colonia só debe saber o mellor camiño global, así como que non require comunicacións punto a punto específicas (non importa saber de qué colonia veciña chega qué resultado para obter o valor óptimo final). A continuación amósanse as funcións nas que se ven involucradas as comunicacións MPI. En concreto, utilizouse unha función de envío non bloqueante cada vez que se atopa unha nova solución prometedora, e outra función para comprobar a recepción de novas solucións dende outras colonias.

```
1  /**
2   Routine to send the best solution found to the rest of Colonies
3   **/
4  void sendBestSolutionToColonies ( void )
5  {
6      int     i,j;
7      int     rc;
8
9      /* Send best solution found (tour) to the rest of the colonies
       */
10
11     for( i=0 ; i<NPROC ; i++ )
12        if ( i != mpi_id){
13            MPI_Isend(best_so_far_ant->tour, n+1, MPI_LONG, i,
14                      3000, MPI_COMM_WORLD, &Srequest);
```

```
15         }
16
17     for ( j=0 ; j<n+1 ; j++ )
18             best_global_tour[j] = best_so_far_ant->tour[j];
19     best_global_tour_length = best_so_far_ant->tour_length;
20
21     if (mpi_id==0 && cc_report) fprintf(cc_report,"%ld \t
   %f\n",best_global_tour_length,elapsed_time(REAL));
22
23 }
24
25
26 /**
27  Routine to check if there are pending Tour messages from Colonies
28  **/
29 void listenTours()
30 {
31     int      i, j;
32     int      flag;
33     long int     recv_tour_length;
34
35     /* Loop to listen best solutions from colonies */
36     for(i=0; i<NPROC ; i++) {
37
38       if(i!=mpi_id){
39
40         flag = 1;
41
42         while ( flag == 1 ) {
43
44           MPI_Test(&request[i][n_try], &flag, &status);
45
46             if (flag==1) {
47
48               recv_tour_length = compute_tour_length(
   &global_tour[status.MPI_SOURCE][0] );
49
50               if ( recv_tour_length < best_global_tour_length) {
51                 best_global_tour_length = recv_tour_length;
52                 for ( j=0 ; j<n+1 ; j++ )
53                     best_global_tour[j] =
   global_tour[status.MPI_SOURCE][j];
54
55                 if (mpi_id==0 && cc_report) fprintf(cc_report,"%ld
   \t %f\n",best_global_tour_length,elapsed_time(REAL));
56                 write_report();
```

```
57                 }
58
59              /* Prepare to receive more solutions */
60                 MPI_Irecv(&global_tour[status.MPI_SOURCE][0], n+1,
    MPI_LONG,
61                        status.MPI_SOURCE, 3000, MPI_COMM_WORLD,
    &request[i][n_try]);
62
63              }
64        }
65      }
66    }
67 }
```

Como xeralmente ocorre ao aplicar estas solucións, hai que obter un balance entre número de comunicacións e rendemento para evitar que tanto a sobrecarga producida polo establecemento das conexións como a propia latencia da rede de interconexión dos nodos opaque (ou incluso chegue a contrarrestar) os beneficios producidos pola propia paralelización.

Para o propósito deste Traballo de Fin de Grao (TFG), prescindiuse do uso da paralelización inter-colonia (*OpenMP*), para obter un entorno MPI puro onde poder probar a aplicación da tolerancia a fallos (a tolerancia a fallos debería funcionar independente do uso de *OpenMP* xa que os fallos son a nivel de proceso, polo cal é un problema que afecta á paralelización multicolonia únicamente).

2.6 Alta dispoñibilidade e tolerancia a fallos

Para poder estudar as diferentes estratexias de tolerancia a fallos no ACO, definiranse algúns conceptos e métricas da enxeñaría da fiabilidade para poder avaliar as solucións propostas.

2.6.1 Garantía de funcionamento

A garantía de funcionamento (GdF) dun sistema é unha propiedade que permite aos usuarios do mesmo confiar xustificadamente no servizo que este proporciona [18]. Que o sistema sexa confiable implica que este ten que ser quen de evitar as avarías máis graves e/ou frecuentes, as cales non son toleradas polos usuarios. A garantía de funcionamento engloba catro campos:

- **Dispoñibilidade:** ou capacidade dun sistema de estar operativo cando se necesita.

Figura 2.2: Relación causa-efecto das amezadas da GdF dun sistema

- **Fiabilidade:** ou capacidade dun sistema para funcionar correctamente durante un período de tempo.
- **Mantenibilidade:** ou capacidade dun sistema para facilitar o seu mantemento ou reparación.
- **Inocuidade:** ou a capacidade dun sistema de operar sen provocar consecuencias catastróficas.

A GdF estivo tradicionalmente vencellada a sistemas críticos onde un fallo pode provocar consecuencias catastróficas, como pode ser a industria aeroespacial ou a enerxía nuclear.

Ameazas á GdF dun sistema

Existen tres ameazas á garantía de funcionamento dun sistema, as cales están interrelacionadas mediante unha relación de *causa-efecto*:

- **Fallos**: defecto ou imperfección no hardware ou software dun sistema. Estes poden ser permanentes, se se deben a un fallo de fabricación ou deseño; intermitentes, se aparecen e desaparecen rápidamente durante un período curto de tempo (por exemplo un bug no software que só afecte a determinados casos límite); ou transitorios, no caso que deriven de causas externas como un corte do suministro eléctrico. Cada tipo de fallo require un tratamento distinto. Por exemplo, un fallo permanente pode non ser solventable nunca sen remplazar o sistema ou rediseñalo; un intermitente poder ser corrixido mediante unha actualización do compoñente afectado; e os transitorios non se poden corrixir, só se poden tomar medidas de prevención para intentar mitigar os efectos dos mesmos.
- **Erros**: desviación ou corrupción do estado interno do sistema.
- **Avarías**: manifestacións dos erros como mal funcionamento do sistema. Poden ser parciais, se o sistema aínda pode realizar algunha das súas funcións; ou totais, se os sistema queda inservible.

Tal e como se amosa na figura 2.2, un fallo provocará un erro. Ese erro propagarase polo sistema, posiblemente en forma de outros erros, ata que se manifestará ao usuario como unha avaría.

No contexto deste proxecto consideraremos que os fallos poderán ser físicos, causado por exemplo pola caída dun nodo, o fallo dun core dalgún procesador ou a caída das comunicacións entre dous nodos; ou por software, como podería ser a terminación manual dalgunha colonia nun nodo por algún ente externo ao sistema. En ambos casos, un fallo manifestarase coma un erro na libraría de MPI, o cal causaría unha avaría: o fallo do programa. Este TFG pretende estudar diferentes estratexias para que, ante a ocorrencia dun fallo, o erro resultante en MPI non dea lugar a unha avaría, continuando a execución e evitando que o usuario se decate do problema.

2.6.2 Estratexias para aumentar a GdF

As estratexías pódense dividir en dous grupos, dependendo de se estas se aplican antes ou despois da detección dos fallos.

Prevención de fallos

A prevención de fallos consiste na utilización de métodos e técnicas que minimicen a introdución de fallos durante o desenvolvemento do sistema, como o análise exhaustivo de requerimentos, a aplicación de boas prácticas de deseño, procesos de validación e verificación exhaustivos e a inclusión de mecanismos de control de calidade.

Tolerancia a fallos

A tolerancia a fallos consiste na introducción de mecanismos no deseño do sistema que permitan evitar a aparición de avarías a pesar de que se produzan fallos. Dependendo de en qué punto se aplique esta tolerancia a fallos, haberá dúas estratexias distintas:

- **Técnicas de enmascaramento de fallos:** técnicas que evitan que os fallos produzan erros. Para isto, o sistema debe ser deseñado coa capacidade de detectar o fallo e evitar que o mesmo corrompa o seu estado interno.

- **Técnicas de detección e recuperación ante erros:** técnicas que detectan a ocorrencia de erros e devolven ao sistema a un estado libre de erros. Neste caso espérase a que se manifeste o fallo, e evítase a propagación do mesmo a través das capas do sistema.

Neste proxecto incídese nas estratexias de tolerancia a fallos, debido a que a prevención debe ser tomada en conta á hora de desenvolver o sistema. Por outra banda, as técnicas de tolerancia a fallos son un engadido ao sistema orixinal co obxectivo de aumentar a súa garantía de funcionamento.

2.6.3 Alta dispoñibilidade

A alta dispoñibilidade é a capacidade que ten un sistema para realizar a súa función sen interrupción durante un período superior ao que se podería esperar, baseándonos nas fiabilidades individuais dos compoñentes que a compoñen [18]. No caso que nos compete, sería a capacidade do sistema de conseguir solucións a problemas os cales requiran tempos de computación maiores aos tempos de fiabilidade dos compoñentes do entorno de execución por separado: fallos de hardware en elementos de proceso como procesadores e tarxetas gráficas, caídas de suministro en partes do sistema, fallos de conexión ou caída da rede de interconexión entre nodos; así como calquer fallo en software, tanto no propio software de usuario como no entorno de sistema operativo e librarías a utilizar.

2.6.4 Modelización de fallos en sistemas informáticos

Defínese *fiabilidade* dun sistema no instante t, $R(t)$, como a probabilidade de que o sistema opere sen avarías no intervalo [0, t], supoñendo que operaba correctamente no instante 0. Este valor estará comprendido sempre entre 0 e 1 [18].
A *taxa de avarías*, λ, é o número esperado de avarías por unidade de tempo. Como un sistema informático posúe unha compoñente física e outra lóxica, debemos estudar a fiabilidade do mesmo en conxunto.

Fallos en hardware

O hardware segue a *lei da fiabilidade exponencial*, posto que nos dispositivos electrónicos a taxa de avarías ao longo da súa vída útil considérase constante, e esta verase incrementada únicamente debido ás fallas por desgaste, as cales ocorrerán fora da vida útil do sistema.
Considerando λ=cte, podemos definir a fiabilidade no hardware do sistema como $R(t) = e^{-\lambda t}$. Canto maior sexa a taxa de fallos, maís pronunciada será a curva, obtendo menos fiabilidade para o mesmo instante de tempo t_i.

2.6.5 Métricas

Finalmente, defínense tres métricas para avaliar a fiabilidade do sistema:

- ***Mean Time To Failure (MTTF)***: tempo medio ata a ocorrencia da primeira avaría dende que o sistema se pon en funcionamento.

$$MTTF = \frac{1}{N} \sum_{i=1}^{N} t_i \quad (2.2)$$

sendo t_i o tempo ata a primeira avaría do sistema i-ésimo e N o número de subsistemas ou compoñentes do sistema. Úsase como métrica para sistemas non reparables. Se λ=cte, $MTTF = \lambda^{-1}$.

- ***Mean Time To Repair (MTTR)***: tempo medio necesario para a reparación dun sistema.

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} t_i \tag{2.3}$$

sendo t_i o tempo de reparación do sistema i-ésimo e N o número de subsistemas ou compoñentes do sistema. É unha métrica da mantenibilidade do sistema.

- ***Mean Time Between Failure (MTBF)***: tempo medio entre avarías dun sistema.

$$MTBF = \frac{T}{\sum_{i=1}^{N} n_i} \tag{2.4}$$

onde n_i é o número de avarías do sistema i-ésimo, N o número de subsistemas ou compoñentes do sistema e T o tempo operativo total.

2.7 User Level Failure Mitigation (ULFM)

A especificación *User Level Failure Mitigation (ULFM)* é unha extensión experimental de MPI desenvolvida polo Grupo de Tolerancia a Fallos do Foro de MPI a partires do estándar MPI 3.1. Na actualidade atópase baixo avaliación do comité de estandarización de MPI. Esta especificación establece o conxunto mínimo de cambios que debe conter calquer aplicativo ou libraría MPI para incluír técnicas de tolerancia a fallos e construír novas formas de toleracia a fallos [4][11].
O obxectivo principal desta extensión do estándar é a especificación de clases de erro e interfaces que permintan ao usuarios continuar as comunicacións MPI sinxelas despois de que ocorra algún fallo, así como reconstruír obxectos MPI segundo sexa necesario para restaurar as capacidades de MPI de levar a cabo comunicacións máis elaboradas, como no caso de arranxar comunicadores para poder realizar comunicacións colectivas. Non é o obxectivo de ULFM incluír mecanismos para a recuperación de datos perdidos debidos a fallos de procesos. Isto é responsabilidade do usuario. O comportamento por defecto da implementación de MPI para o caso de que falle un proceso pode ter dúas vías:

- calquera operación MPI na que interveña un proceso fallido deberá devolver un código de éxito ou lanzar un erro [4], evitando bloquearse indefinidamente.

- calquera operación MPI que non involucre procesos que fallen completarase normalmente, a menos que se vexa interumpida polo usuario a través da funcionalidade proporcionada por MPI para tal efecto.

Os erros teñen caracter local, e MPI non garante que outros procesos sexan notificados de ese erro. A propagación asíncrona de erros tampouco está garantida polo estándar, polo que os usuarios deberán proceder con cautela cando se intente delimitar o conxunto de procesos nos cales un fallo lanzou un erro.

2.7.1 Novos códigos de erros

En ULFM engadíronse os seguintes códigos de erro:

- `MPI_ERR_PROC_FAILED`: a operación non se puido completar debido a un fallo nun proceso.
- `MPI_ERR_PROC_FAILED_PENDING`: a operación non se puido completar debido a un fallo nun proceso, pero a petición segue pendente e a operación pode que se complete máis tarde.
- `MPI_ERR_REVOKED`: o obxecto usado para a comunicación, o comunicador, foi revogado.

2.7.2 Novas funcións de control e propagación de erros

En ULFM engadíronse novas funcións, as cales podemos clasificar en función do uso como:

Funcións de notificación

As funcións de notificación sirven para discernir qué procesos se decatan dos erros locais. Estas funcións non rompen o funcionamento de MPI, posto que as comunicacións seguirán activas no comunicador.

- `MPIX_Comm_failure_ack`: proporciona aos usuarios un xeito de recoñecer todos os fallos notificados localmente nun comunicador.
- `MPIX_Comm_failure_get_acked`: devolve o grupo de procesos que se recoñeceu que fallaron nun comunicador.

Funcións de propagación

As funcións de propagación encárganse de propagar os erros e convertilos en globais. Destaca a función `MPIX_Comm_revoke`, a cal interrumpe toda comunicación activa ou

futura en todos os rangos no comunicador desexado. Todas esas operacións interrumpidas lanzarán o erro `MPI_ERR_REVOKED`.

Funcións de recuperación

As funcións de recuperación son as encargadas de aranxar os mecanismos de comunicación rotos e devolver ao sistema a un estado estable para que poida seguir o funcionamento normal do programa. Destacan:

- `MPIX_Comm_shrink`: crea un novo comunicador (idéntico en todos os rangos) excluíndo o grupo de procesos que fallaron.

- `MPIX_Comm_agree`: o obxectivo é acordar unha etiqueta común sobre o grupo de procesos non fallidos do comunicador. Utilízase para comprobar a consistencia do grupo de procesos, xa que se os valores da etiqueta son distintos, a comunicación fallou no grupo. É util despois do *shrink* dun comunicador para verificar que as novas conexións funcionen.

Con estes engadidos e xunto co uso de manexadores de erros (*error handlers* no seu termo en inglés, que é o que usaremos neste documento), cos que xa contaba MPI, é posible implementar solucións tolerantes a fallos. Con todo, como o estándar aínda non está aprobado, é unha tecnoloxía nada trivial e a documentación é moi escasa, o que pode provocar un aumento nos tempos de desenvolvento de aplicativos tolerantes a fallos.

Capítulo 3

Planificación e custos

A planificación dun proxecto software é unha parte crítica para a consecución exitosa do mesmo. Unha boa planificación permitirá realizar o traballo en menos tempo, con máis calidade e menor custo. Neste capítulo, exporanse as estimacións realizadas antes de comezar o proxecto, a planificación xeral do proxecto e os problemas atopados en cada unha das fases durante o seguemento do proxecto.

3.1 Planificación temporal do proxecto

O tempo límite para a realización do proxecto foi prefixado entre o 1 de febreiro de 2022 e o 28 de xuño de 2022. Delimitáronse 5 fases para a súa consecución: estudo de material bibliográfico, instalación e configuración do entorno, implementación das solucións tolerantes a fallos, validación das solucións tolerantes a fallos e documentación.

3.1.1 Planificación inicial

A planificación do proxecto fíxose tomando como base a guía docende da materia. Segundo esta, o TFG debería levar unhas 275 horas de traballo. En base a iso, divideuse o tempo entre os meses de febreiro e xuño en cada unha das 5 tarefas. Para cada mes consideráronse 22 días hábiles (deixando libres os fins de semana e festivos), e entre 1 e 3 horas de traballo por

Tarefa/Fase	Data de inicio	Data de fin	Horas/día	Días totais	Horas Totais
Estudo da bibliografía	01/02/2022	28/02/2022	2	22	44
Instalación e configuración do entorno	01/03/2022	04/03/2022	1	4	4
Implementación	07/03/2022	29/04/2022	3	40	120
Validación	02/05/2022	31/05/2022	3	22	66
Documentación	01/06/2022	28/06/2022	2	22	44

Táboa 3.1: Planificación inicial. Valores estimados

día. Como se pode ver na táboa 3.1, a maior carga de traballo concéntrase na implementación e nas probas, posto que concentran o 67% do esforzo adicado ao proxecto. A suma total de horas estimadas para a consecución do proxecto é de 278 horas. Na figura 3.1 amósase un diagrama de Gantt coa planificación inicial do proxecto.

3.2 Avaliación de custos

No tocante aos custos, os únicos recursos necesarios son un computador persoal con conexión a internet para poder operar no clúster Pluton remotamente e a bibliografía.

Como os recursos necesarios para a consecución do proxecto son de uso público para os membros da comunidade universitaria que soliciten acceso aos mesmos ou poden ser atopados libremente en liña, o custo de recursos é un custo fixo de 0€. Para o custo dos recursos humanos tómanse o salario promedio dun egresado de enxeñaría informática en España, 11 €/h [19]. Con todo, o custo total estimado do proxecto serían 3058€.

3.3 Seguimento do proxecto e esforzo real

Xa dende a primeira fase do proxecto houbo atrasos. Debido á falta de documentación, a comprensión do material previo levou maís do previsto. Isto fixo que a fase de configuración comezara semana e media máis tarde. Non tardaron en atoparse problemas graves para a execución e compilación co entorno que proporcionaba o clúster. Para mitigar estes problemas adicáronse 3 horas ao día a esta tarefa, que alongouse durante 13 días, ata que se conseguiu unha configuración válida que permitira compilar e executar con ULFM. Isto debeuse principalmente á falta de documentación, xa que o proxecto aínda non forma parte do estándar. Máis concretamente, a execución con tolerancia a fallos non funcionou ata que se incluíron na compilación uns flags que non estaban documentados (eses flags obtivéronse por proba e erro a partir de respostas dadas pola comunidade en múltiples foros de MPI) e ata que non se configurou unha variable de entorno en execución.
Con este xa amplo historial de contratempos, decidiuse ampliar a 3 horas diarias o esforzo, pasando a considerar todos os días do mes, con fins de semana e festivos incluídos para intentar axustarse á planificación e ter un ”fin de curso” menos cargado de traballo. A implementación presentou novos retos, derivados do non determinismo intrínseco ao problema, sumado ao non determinismo que presenta MPI e sumado ao non determinismo que presentaba o feito de matar procesos para obligar a que o sistema lanzara a tolerancia a fallos. Esta falta de determinismo dificultaba a labor de depuración, obtendo resultados moi dintintos entre sucesivas execucións. A todo isto hai que engadirlle a falta dun mecanismo de *debugging* para esta tecnoloxía, o que obrigaba a facer un extenso *logging* para poder seguir o fluxo de execución.

Figura 3.1: Diagrama de Gantt da planificación inicial do proxecto

Tarefa/Fase	Data de inicio	Data de fin	Horas/día	Días totais	Horas Totais
Estudo da bibliografía	01/02/2022	10/03/2022	2	31	62
Instalación e configuración do entorno	011/03/2022	22/03/2022	3	11	33
Implementación	23/03/2022	11/07/2022	3	110	330
Validación	12/07/2022	10/08/2022	3.5	28	98
Documentación	11/08/2022	31/08/2022	3	20	60

Táboa 3.2: Valores reais de esforzo no proxecto

Con todo, esta solución presentaba problemas, debido a que o acceso dos diferentes procesos aos ficheiros para tal efecto facían que a propia trazabilidade non fose exacta. Ademáis volvíase pouco tratable, pois para unha execución con moi poucos procesos e varios segundos, o sistema xeraba varios centos de MegaBytes de datos. Isto levou a rematar o desenvolvemento o día 11 de xullo, polo que o proxecto ía xa varios meses atrasado.
Para abordar o testing decidiuse subir media hora máis o esforzo diario. O testing tamén levou moitas máis horas das estimadas previamente, debido a que as probas tardaban varias horas en realizarse e os resultados non eran os esperados, principalmente polo uso de problemas nos cales o algoritmo caía en mínimos locais ou por estar medindo con criterios de parada pouco adecuados. Despois dun mes remataron as probas, e adicouse o restante do mes de agosto á realización desta memoria cun esforzo medio de 3 horas por día.
Finalmente, o proxecto foi completado en 583 horas, repartidas en 8 meses, o que duplica o estimado inicialmente. A contabilidade completa pódese observar na táboa 3.2.

Capítulo 4

Implementación

NESTE capítulo exporanse os detalles do desenvolvemento dun sistema tolerante a fallos con MPI sobre a linguaxe C, particularizando os detalles para o problema do viaxante. Con todo, o código desenvolvido fíxose o máis xenérico posible, de xeito que os manexadores de erros poidan ser utilizados por calquera outro aplicativo coa menor necesidade de adaptación posible. Como xa se verá, a inclusión da tolerancia a fallos en MPI non é trivial, o que obrigará a facer cambios nos códigos fonte, potencialmente nas comunicacións e na arquitectura de certos procedementos. Todos os pasos de configuración dependerán do entorno utilizado e deberán ser adaptados en función das tecnoloxías e versións a utilizar.

4.1 Instalación de ULFM e compilación con tolerancia a fallos

A instalación de ULFM é un proceso longo debido á falta de binarios precompilados nos repositorios públicos de Linux. Isto débese a que aínda non forma parte do estándar de MPI, e polo cal non ven incluído cos paquetes dispoñibles. Para a instalación, o xeito máis sinxelo sería a compilación local dos ficheiros fonte que se poden descargar dende o repositorio oficial do grupo encargado da confección de ULFM[1]. A versión de OpenMPI utilizada para o desenvolvemento deste traballo foi a versión 4.02, posto que foi a última versión estable coa que se conseguiu que funcionara ULFM. Probouse inicialmente coa reciente 5.0, pero non se chegaban a executar os handlers, polo que se optou polo *downgrade* ata unha versión máis estable.

Malia ter que instalar OpenMPI manualmente, pódese crear un script que faga todo o traballo xunto. A compilación é moi longa posto que ten que descargar varios paquetes e compilar todo o sistema, pero non é difícil. As dependencias necesarias son mínimas, sendo as básicas para a compilación de C: un compilador e un automatizador. No meu caso, decanteime por GCC e Make, posto que son as ferramentas coas que xa estaba familiarizado, e tamén porque

[1] https://fault-tolerance.org/

son ferramentas open source. Un exemplo de script de instalación sería o seguinte:

```
# ter coidado de descargar a versión mpi + ulfm
git clone https://bitbucket.org/icldistcomp/ulfm2.git
cd ulfm2/
./autogen
# en --prefix, o directorio de instalación; --with-ft para engadir
    ULFM
./configure --prefix=\$HOME/mpi402 --with-ft
make install
```

O proceso de instalación tarda derredor dunha hora. En caso de que remate sen fallos, xa deberíamos poder utilizar ULFM. Para poder compilar e linkar correctamente, debemos engadir ao path os directorios de `../mpi402/bin` e `../mpi402/lib`. Para que non haxa problemas de versións, deberíamos utilizar o xestor de módulos - neste caso *Lmod* - para desactivar todas as outras versións de OpenMPI instaladas no sistema. O seguinte paso sería a compilación e execución dos códigos tolerantes a fallos. Para iso, é necesaria a modificación dos scripts de compilación para facer o linkado estático das librarías de ULFM. No seguinte script pódense ver os directorios proporcionados para a compilación a través da opción de compilación `-I`; e no linkado coa opción `-L`. Neste caso particular, tamén se desactivou o uso de *OpenMP* comentando a opción `-fopenmp` e `-lpthread`. No caso particular do entorno dispoñible no clúster Plutón foi necesaria a adición da flag de compilación `-fuse-ld=gold` para que GCC utilizara `gold`, o cal é un *linker* alternativo a `ld`.

```
VERSION=MPI-OMP
WITH_ACO=-DFT_ACO
#HANDLER = -DFT_ERRORS_ARE_FATAL
#HANDLER = -DFT_ERRORS_RETURN
#HANDLER = -DFT_ABORT_ON_FAILURE
#HANDLER = -DFT_IGNORE_ON_FAILURE
HANDLER = -DFT_RESPAWN_ON_FAILURE
#HANDLER = -DFT_ELASTIC_RESPAWN_ON_FAILURE
#KILL_POLICY = -DKILL_AT_50
#KILL_POLICY = -DKILL_AT_25_AND_50
KILL_POLICY = -DKILL_ALL_BUT_ONE
OPTIM_FLAGS=-O
WARN_FLAGS=-Wall -Wextra #-Werror
# -D_FT_ACO_ to compile with Fault tolerance API
#CFLAGS=$(WARN_FLAGS) $(OPTIM_FLAGS) -I ~/mpi402/include/ -fopenmp
    -D$(WITH_ACO) -D$(HANDLER)
CFLAGS=-fuse-ld=gold $(WARN_FLAGS) $(OPTIM_FLAGS) -I
    ~/mpi402/include/ $(WITH_ACO) $(HANDLER) $(KILL_POLICY)
CC=mpicc
```

```
18 # To change the default timer implementation, uncomment the line
       below
19 #TIMER=dos
20 TIMER=unix
21 #LDLIBS=-lm -L/home/miguel.blanco/mpi402/lib -fopenmp -lmpi
       -lpthread
22 LDLIBS=-lm -L/home/miguel.blanco/mpi402/lib -lmpi
23 all: clean acotsp
24 clean:
25         @$(RM) *.o acotsp
26 acotsp: acotsp.o parallel.o TSP.o utilities.o ants.o InOut.o
       $(TIMER)_timer.o ls.o parse.o ft_aco.o
27 acotsp.o: acotsp.c
28 TSP.o: TSP.c TSP.h
29 ants.o: ants.c ants.h
30 InOut.o: InOut.c InOut.h
31 utilities.o: utilities.c utilities.h
32 ls.o: ls.c ls.h
33 parse.o: parse.c parse.h
34 parallel.o: parallel.c parallel.h
35 # Fault tolerance API
36 ft_aco.o: ft_aco.c ft_aco.h
37 $(TIMER)_timer.o: $(TIMER)_timer.c timer.h
38 dist : DIST_SRC_FILES=*.c *.h README *.tsp Makefile gpl.txt
39 dist : all
40         @(mkdir -p ../ACOTSP-$(VERSION)                  \
41         && rsync -rlpC --exclude=.svn $(DIST_SRC_FILES)
       ../ACOTSP-$(VERSION)/ \
42         && cd ..        \
43         && tar cf - ACOTSP-$(VERSION) | gzip -f9 >
       ACOTSP-$(VERSION).tar.gz \
44         && rm -rf ./ACOTSP-$(VERSION)
           \
45         && echo "ACOTSP-$(VERSION).tar.gz created." && cd $(CWD) )
```

Coa compilación solucionada, só restaría a creación de scripts de execución. Neste caso engadiremos os executables da libraría recién instalada ao path, e débese configurar unha variable de entorno da mesma, necesaria para que se aplique a tolerancia a fallos. A variable `MPI_FAULT_CONTINUE` debe ter o valor 1. Deste xeito, cando o sistema reciba o fallo dun proceso, non matará aos outros inmediatamente, senón que dará a posibilidade ao sistema de executar o código do *error handler* definido por defecto (en caso de habelo). Se non houbese *error handler*, a execución continuaría, pero o fluxo do programa non seguiría posto que en cada comunicación MPI lanzaríase un fallo, e como o problema non se resolve, o sistema entraría nun sumidoiro de erros concatenados.

```
#!/bin/bash
#
#SBATCH --job-name=ft_aco
#SBATCH -N 1
#SBATCH -n 16
#SBATCH -c 1

export PATH="$PATH:/home/miguel.blanco/mpi402/bin"
export
    LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/miguel.blanco/mpi402/lib"
export I_MPI_FAULT_CONTINUE=1
mpirun -np 16 ./acotsp -i rat783.tsp -r 1 -t 1000 -x -o 8806
```

Por último, o programa executaríase como calquera outro programa MPI, co lanzador utilizado pola implementación de MPI escollida (neste caso `mpirun`, da libraría *OpenMPI*).

4.2 Fluxo de execución dun programa tolerante a fallos en ULFM

O fluxo de execución dun programa tolerante a fallos está composto de tres etapas interconectadas: execución normal, detección do erro e proceso do erro. En caso de que non se poida procesar o fallo (por exemplo, porque só se estaba executando un proceso) chegarase a un estado sumidoiro, que será a finalización anormal do programa MPI. Na figura 4.1 amósase un diagrama coas relacións entre etapas do fluxo de execución dun programa tolerante a fallos en ULFM.

4.2.1 Detección de fallos en MPI

En MPI, a notificación dun fallo non invalida a comunicación. Subsecuentes fallos seguirán saíndo se as operacións sobre un comunicador non se poden completar, pero as comunicacións *peer-to-peer (P2P)* entre procesos vivos seguirán funcionando. A detección do fallo dase cando un proceso intenta establecer conexión con outro nun estado non válido. Isto causa que esa operación MPI lance un erro para o *sender*. Para o caso no que o proceso co estado inválido sexa o *sender*, o *receiver* poderá entrar nun estado denominado *deadlock*. Nese estado, estará continuamente esperando polos datos do *sender*, o que causará que o programa quede bloqueado. Para evitar isto, débese de interrumpir sempre a comunicación nun comunicador ante un fallo se estamos a utilizar operacións bloqueantes. No caso de usar operacions non bloqueantes, non sería necesario, posto que simplemente detectaría o *buffer* de datos da conexión baleiro. En MPI, a detección de fallos é local, polo que diferentes procesos poden ver diferentes códigos de erro.

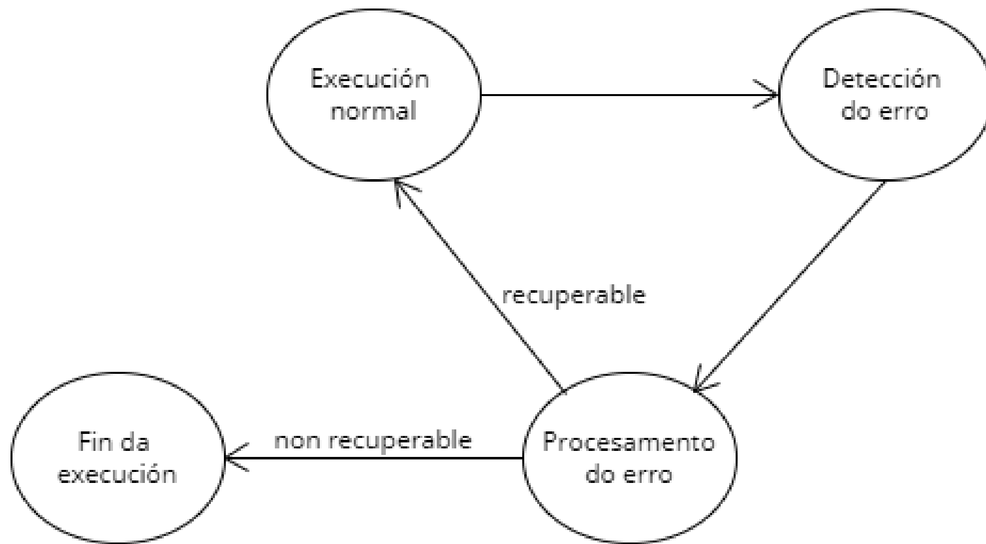

Figura 4.1: Fluxo de execución dun programa tolerante fallos

4.2.2 Procesamento dos fallos con UFLM

Cada vez que unha operación MPI non se pode completar exitosamente, lanza un erro. Ese erro será servido a través dun *error handler*, que é unha rutina que se executará despois de que se lance o erro. Un *error handler* debería conter toda a lóxica necesaria para facer que o programa poida tolerar os fallos segundo o paradigma escollido. Hai que ter en conta que as implementacións de MPI non garanten que esta funcionalidade estea presente, malia formar parte do estándar dende a versión 2. A todo isto hai que engadirlle que as implementacións que si dispoñen deste mecanismo non garanten valores de temporización, é dicir, que garanten que o sistema sempre executará a rutina definida polo *handler*, pero será nun período de tempo arbitrario. Esta falta de determinismo na execución pode ser un problema á hora de desenvolver e depurar programas tolerantes a fallos, así como pode dificultar a súa execución en casos límite.

4.3 Estudo das solucións xenéricas de ULFM

ULFM define dous *error handlers* propios. Na seguinte sección analizaranse a capacidade dos mesmos para engadir resiliencia á aplicación.

4.3.1 MPI_ERRORS_ARE_FATAL

É o *handler* por defecto asociado ao comunicador `MPI_COMM_WORLD` despois da inicialización de `MPI_init`. Debido a isto, se o usuario decide non controlar os *error handlers*,

todos os erros serán tratados como irrecuperables, forzando a invocación deste *handler*. Cando se invoca, causa que todos os procesos finalicen a súa execución. Ten o mesmo efecto ca chamar a `MPI_Abort`. A finalización é forzada, e neste caso non se espera a que se liberen bloqueos derivados de posibles conexións, ou bloqueos derivados de operacións con varios fíos, o que pode conlevar a perda de información ou corrupción de datos en caso de que o aplicativo estea a utilizar ficheiros ou recursos compartidos.
Este *handler* non nos da ningunha posibilidade de construír un código tolerante a fallos. Ademáis, ao rematar a execución dunha forma non segura, os resultados efectivos serían os mesmos ca se se producira un fallo de segmento, polo que incluso se poderían por en perigo datos compartidos utilizados polo aplicativo.

4.3.2 MPI_ERRORS_RETURN

Este *handler* ten como único propósito evitar que MPI obrigue á terminación do programa cando hai un fallo. Internamente a rutina asociada non fai absolutamente nada máis ca devolver o código de erro ao usuario. Unha posible implementación semellante sería:

```
/* ... */
#define MPI_ERRORS_RETURN errors_return_handler;
/* ... */
MPI_Errhandler errors_return_handler;
/* ... */
int MPI_init(...)
{
    /* ... */
    MPI_Comm_create_errhandler(&errors_return_handler,
    &mpi_errors_return);
    /* ... */
}

int mpi_errors_return(MPI_Comm * comm, int* err, ...)
{
    return *err;
}
```

Dado que case todas as operacións de MPI devolven algún código de erro, o usuario pode optar por xestionar os fallos manualmente no código principal, comprobando os códigos de erros e executando algunha rutina de recuperación en caso de obter resultados distintos de `MPI_SUCCESS`. O propósito principal deste *handler* é permitir que o usuario emita mensaxes de erro definidas polo usuario e realice accións non relacionadas co MPI (como limpar os *buffers* de comunicacións) antes de que un programa remate a súa execución, pero depende moito do tipo de aplicativo. Podemos atopar diferentes usos para o handler en función do

tipo de operacións que se usen, e en base a iso, adaptar o programa para dar certo nivel de tolerancia:

- nos programas onde se estean a utilizar operacións colectivas bloqueantes, este *handler* poderá ser utilizado para o seu propósito orixinal, é dicir, para intentar liberar memoria, recursos compartidos ou realizar algún tipo de *checkpointing*. O *checkpointing* consiste en almacenar información da execución que permita, en caso de fallo, recuperar a execución no punto onde se realizou o *checkpoint*. Ao mesmo tempo o *handler* pode utilizarse para escribir ficheiros de *log*, e dicir, ficheiros con información para realizar unha trazabilidade do erro.

- nos programas nos que se utilicen comunicacións non bloqueantes ou comunicacións bloqueantes punto a punto pódese utilizar para seguir a execución, posto que en MPI os erros son locais, e as comunicacións entre dous *endpoints* válidos seguirán a funcionar nese comunicador.

Para os casos que se acaban de definir, podemos observar as fortalezas e debilidades desta ferramenta. Por unha banda, permite un control absoluto cando se trata de erros recuperables, posto que poderías implementar unha estratexia de tolerancia a fallos *ad-hoc* para cada chamada a MPI no código; non obstante, xeralmente é máis conveniente e eficiente definir *handlers* propios para solventar estes erros. Por outra banda, para os casos nos que non sexan recuperables os erros (comunicacións colectivas bloqueantes), levaríamos ao sistema a un estado de interbloqueo, xa que o uso deste *handler* evitaría que se abortara o programa, pero como o comunicador non é revogado en ningún momento, causará dous problemas: os procesos que estean esperando por datos que deberan ser enviados por un proceso que fallou permanecerán bloqueados indefinidadamente. Pola contra, os procesos que envíen datos a procesos fallidos lanzarán o erro, polo que entrarán nun bucle no que se executará a rutina definida polo *handler* ata que o planificador do sistema operativo decida finalizar o programa (ben por límite de tempo ou por orde do usuario, unha vez vexa esta situación nos *logs*).

4.4 *Handlers* customizados desenvolvidos polos usuarios

Unha vez abordadas as características das solucións máis básicas fornecidas por ULFM, creouse unha libraría cun conxunto de ferramentas (*handlers* + funcións auxiliares) para brindar diferentes niveis de tolerancia a fallos en función das características da aplicación:

- aplicativos cuxa arquitectura non permita (ou non compense) a continuación ante un fallo. Neste caso trataranse os fallos como fatales, pero deixa a porta aberta para mecanismos externos de recuperación.

- aplicativos que poidan funcionar cun número menor de procesos, reconfigurando automáticamente e, de xeito transparente ao usuario, o mapa de comunicacións do sistema.
- aplicativos que deban funcionar cun número específico de procesos por motivos de particionamento do problema a resolver, ou por motivos derivados da aplicación específica dalgún patrón de paralelismo, e nos cales a inicialización de novos procesos non sexa excesivamente custosa.

Con estas tres categorías, debería poder brindar certo nivel de tolerancia a fallos á maior parte dos sistemas sen necesidade dun gran esforzo de implementación. Con todo, sempre será necesario adaptar as solucións ao problema específico. Nas seguintes subseccións descríbense os *handlers* desenvolvidos para ter en conta as tres categorías consideradas.

4.4.1 FT_ABORT_ON_FAILURE_HANDLER

O obxectivo deste *handler* é proporcionar un mecanismo de recuperación para os aplicativos que por diversas circunstancias non poidan beneficiarse doutro. O mesmo debe permitir ao programa ser terminado do xeito máis inocuo posible. Para iso, o *handler* intenta finalizar o programa, como se se finalizara unha execución normal sen erros de MPI. Complementando o *handler*, hai un procedemento auxiliar que nos permite configurar unha rutina de limpeza. Nesta rutina, o usuario deberá manexar a memoria para evitar perdas de memoria, desfacer calquera conexión por rede, pechar os ficheiros abertos e demais tarefas para evitar corrupción de datos e malversación de recursos de cómputo. Internamente, o *handler* invocará esa rutina e rematará coa invocación de `MPI_Finalize`. Isto faise para que sexa o propio MPI o que se encargue de liberar os recursos necesarios.

```
1  static void (* ft_cleanup_function) (void *) = NULL;
2
3  static void * ft_cleanup_params = NULL;
4
5  void FT_set_cleanup_function(void * function, void * parameters)
6  {
7      ft_cleanup_function = function;
8      ft_cleanup_params = parameters;
9  }
10
11 int FT_set_cleanup_params(void * parameters)
12 {
13     if (ft_cleanup_function == NULL) {
14         return FT_FAILURE;
15     }
16     ft_cleanup_params = parameters;
```

```
    return FT_SUCCESS;
}

void FT_abort_on_failure(MPI_Comm * comm, int * err, ...)
{
    char error_info[MPI_MAX_ERROR_STRING];
    int error_info_length, rank, size;

    MPI_Comm_rank(*comm, &rank);
    MPI_Comm_size(*comm, &size);
    /* ... */
    MPI_Error_string(*err, error_info, &error_info_length);
    log_error(...);
    /* ... */
    if (ft_cleanup_function != NULL) {
        (* ft_cleanup_function)(ft_cleanup_params);
    }
    /* ... */
    MPI_Finalize();
    exit(EXIT_FAILURE);
}
```

Como se pode observar no pseudocódigo, defínense variables de entorno para controlar a rutina de limpeza. O uso desta é opcional. Hai tamén procedementos para configurar diferentes parámetros para a rutina. Só se pode asignar unha rutina por handler, o que equivale a unha rutina por comunicador. Non obstante, pódense enviar calquera conxunto de datos como parámetros para a rutina, polo que poderíase obter máis variedade implementando a rutina coma un switch. Por exemplo:

```
void cleanup_function(void * args)
{
    /* ... */
    my_cleanup_params* params = (my_cleanup_params *) args;
    switch (params->error_type)
    {
        case ERROR_TYPE_0:
            /* ... */
            break;
        case ERROR_TYPE_1:
            /* ... */
            break;
            .
            .
            .
        default:
            /* ... */
```

```
18      }
19      /* ... */
20  }
```

Deste xeito pódese implementar unha lóxica de erros complexa, válida para calquera casuística posible no código. Podemos dicir que podería substituír fácilmente ao *handler* MPI_ERRORS_RETURN para o caso de erros non recuperables con operacións colectivas bloqueantes, obtendo o mesmo nivel de personalización con menos traballo no código principal do programa. O único cambio sería crear un conxunto de rutinas de limpeza, estruturas de datos para o paso de parámetros, códigos de error internos da aplicación e configurando o comunicador para que utilice FT_ABORT_ON_FAILURE_HANDLER. A vantaxe sería que non habería que tocar nada no código orixinal (exceptuando a configuración do handler).
Ademáis de utilizalo para facer a limpeza, podemos utilizalo como ferramenta de *checkpointing* antes de rematar a execución. Isto permitiranos volver a lanzar o proceso en varias tandas partindo dos datos gardados durante o *checkpointing*, de xeito que na nova execución busque os datos nos ficheiros de control e aforrando tempo de computación. Neste último caso poderíamos consideralo unha ferramenta de recuperación ante desastres. Se o erro provoca un desastre, como por exemplo unha interrupción total que supoña a migración da aplicación a outra infraestrutura, a execución podería ser continuada dende o punto de *checkpoint*.

4.4.2 FT_IGNORE_ON_FAILURE_HANDLER

O obxectivo deste *handler* é proporcionar tolerancia a $N - 1$ fallos (sendo N o número de procesos en execución), permitindo que o sistema siga funcionando con menor nivel de paralelismo e remapeando as comunicacións punto a punto para que estas sigan funcionando ante un fallo. Para conseguir este comportamento, o *handler* tentará illar os grupos de procesos fallantes dos que non fallan, e volver a construír os obxectos de comunicación necesarios para que o sistema se poida comunicar de novo. Para que este *handler* poida funcionar correctamente, é necesario que o mesmo teña acceso aos punteiros que controlan os valores locais de rango dos procesos e o valor global de número de procesos en cada grupo. Disponse tamén de funcións auxiliares para o manexo deses punteiros, coma a rutina FT_set_respawn_data.

```c
char ** gargv;
int * global_rank, *global_procs;

void FT_set_respawn_data(char ** argv, int * mpi_id, int * NPROC) {
    gargv = argv;
    global_rank = (int *) mpi_id;
    global_procs = (int *) NPROC;
}
```

Esta rutina utilízase como configuración de datos para os *handlers* FT_IGNORE_ON_FAILURE_HANDLER, FT_RESPAWN_ON_FAILURE_HANDLER e FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER. Neste caso, só é necesario que os parámetros do rango (mpi_id) e o número de procesos (NPROC) sexan non nulos.

No pseudocódigo listado a continuación pódense ver os compoñentes principais do *handler*.

```c
void FT_ignore_on_failure(MPI_Comm * comm, int * err, ...)
{
    /* ... */
    char error_info[MPI_MAX_ERROR_STRING];
    int error_info_length, size, rank, number_of_dead;
    MPI_Group group_f, group_c;
    int ranks_gf, ranks_gc;
    /* ... */
    MPI_Error_string(*err, error_info, &error_info_length);
    /* ... */
    log_error(...);
    MPI_Comm_size(*comm, &size);
    MPI_Comm_rank(*comm, &rank);
    MPIX_Comm_failure_ack(*comm);
    MPIX_Comm_failure_get_acked(*comm, &group_f);
    MPI_Group_size(group_f, &number_of_dead);
    /* ... */
    MPI_Comm new_comm;

    repair(*comm, &new_comm);

    MPI_Comm_group(*comm, &group_c);
    MPI_Comm_rank(*comm, &shrinked_rank);
    MPI_Comm_size(*comm, &shrinked_size);
    /* ... */
    *global_rank = (int) shrinked_rank;
    *global_procs = (int) shrinked_size;
    /* ... */
}
```

```
30
31 void repair(MPI_Comm * comm) {
32     MPI_Comm * scomm;
33     int ns, nc;
34     if ( *comm != MPI_COMM_NULL) {
35         scomm = (MPI_Comm *) malloc(sizeof(MPI_Comm));
36         /* ... */
37         MPIX_Comm_shrink(*comm, scomm);
38
39         MPI_Comm_size(*scomm, &ns);
40
41         MPI_Comm_size(*comm, &nc);
42
43         MPIX_Comm_revoke(*comm);
44         /* ... */
45         if (MPI_COMM_WORLD != *comm) {
46             int rc = MPI_Comm_free(comm);
47             if (rc == MPI_SUCCESS) {
48                /* ... */
49             } else {
50                 char error_info[MPI_MAX_ERROR_STRING];
51                 int error_info_length;
52                 MPI_Error_string(rc, error_info,
    &error_info_length);
53                 log_error(...);
54             }
55         }
56
57         *comm = *scomm;
58     }
59 }
```

En primeiro lugar utilízanse as funcións de notificación MPIX_Comm_failure_ack para que a libraría marque o proceso que a chama coma notificado, é dicir, que recoñeceu a existencia de erros (a nivel local) no comunicador. Seguidamente utilízase a función MPIX_Comm_failure_get_acked para obter o grupo de procesos que se recoñeu como fallidos no comunicador.

A rutina *repair* encárgase de executar as funcións de propagación e recuperación. Primeiramente execútase a función MPIX_Comm_shrink, a cal crea on comunicador novo para o grupo de procesos vivos. Posteriormente realízase a revogación do comunicador vello, interrompendo todas as conexións abertas. Realízase nesta orde para poder executar operacións sobre rangos no *handler* as cales fallarían cun comunicador revogado. Se o comunicador usado non é MPI_COMM_WORLD, destrúese o comunicador para liberar recursos do sistema. Finalmente actualízase o novo comunicador no punteiro global do comunicador para que o

cambio sexa transparente ao usuario, obtense o grupo correspondente ao novo comunicador creado, e recalcúlanse todos os rangos para cada proceso. Opcionalmente poderíase utilizar a función MPIX_Comm_agree para comprobar a consistencia do grupo, pero neste caso optouse por non utilizalo por eficiencia. A falta do uso podería provocar que algún proceso quedara nun estado inconsistente e que lanzara o erro MPI_COMM_REVOKED. Neste caso habería que volver a realizar o proceso de notificación, propagación e recuperación pero, como só se daría en certos casos e MPIX_Comm_agree é unha operación colectiva bloqueante, optouse por evitala neste caso.
Ante un erro, este *handler* reducirá en 1 o número total de procesos asociados ao grupo e creará un novo comunicador. Ao rematar, todas as conexións deberían poder ser restauradas e é responsabilidade do usuario facer que estas poidan funcionar cun número de procesos variable. Esta ferramenta é util para fornecer de tolerancia a fallos a programas nos cales o número de procesos non sexa moi relevante, xa sexa debido a que o particionado dos datos da paralelización non é fixo ou porque se trate dun sistema con control distribuído (coma o caso do ACO).

4.4.3 FT_RESPAWN_ON_FAILURE_HANDLER

O obxectivo deste *handler* é brindar tolerancia a fallos a aplicativos que deban funcionar cun número específico de procesos, ou nos cales se queira maximizar a eficiencia acaparando a maior parte de recursos posibles. A arquitectura deste *handler* é moi semellante á do anterior: consta da propia rutina e dunha auxiliar para configurar as propiedades de reinicio do sistema, así como os punteiros para os valores de control de MPI. A rutina de configuración FT_set_respawn_data xa foi explicada na sección anterior. Neste caso o primeiro argumento non poderá ser nulo, senón que deberá conter o comando de inicio dunha instancia do aplicativo.

```
1  void FT_respawn_on_failure(MPI_Comm * comm, int * err, ...)
2  {
3      char error_info[MPI_MAX_ERROR_STRING];
4      int error_info_length, size, rank, number_of_dead;
5      MPI_Group group_f, group_c;
6      int ranks_gf, ranks_gc;
7
8      MPI_Error_string(*err, error_info, &error_info_length);
9      log_error(...);
10     /* ... */
11
12     MPIX_Comm_failure_ack(*comm);
13     MPIX_Comm_failure_get_acked(*comm, &group_f);
```

```
14      MPI_Group_size(group_f, &number_of_dead);
15      /* ... /
16      MPI_Comm new_comm;
17
18      respawn(*comm, comm);
19
20      /* ... */
21      MPI_Comm_group(*comm, &group_c);
22      MPI_Comm_rank(*comm, &shrinked_rank);
23      MPI_Comm_size(*comm, &shrinked_size);
24      *global_rank = (int) shrinked_rank;
25      *global_procs = (int) shrinked_size;
26  }
```

Coma no caso anterior, o propio *handler* segue o mecanismo clásico de notificación, propagación e recuperación, pero neste caso a recuperación é bastante máis complexa. O procedemento de creación dun proceso con MPI realízase a través das funcións definidas para iso en MPI: `MPI_Comm_spawn`. Como este procedemento non depende únicamente do aplicativo, debemos ter en conta a aparición de fallos externos (cando se crea un proceso delegamos esa operación parcialmente no sistema subxacente, e polo cal e posible que non se poidan crear novos procesos por motivos de falta de recursos, accounting do sistema operativo ou calquera outra restrición ou problema inesperado), o que nos obriga a cambiar o *handler* mentres que dura o mesmo. De non facer este cambio, o sistema entraría nun bucle onde se lanzaría un erro por cada fallo de inicialización (facendo que máis procesos intenten crear procesos) e por último a un bloqueo do sistema. Esta lóxica está implementada na rutina *respawn*, cuxo pseudocódigo se lista a continuación:

```
1   void respawn(MPI_Comm comm, MPI_Comm * newcomm)
2   {
3       MPI_Comm icomm, scomm, mcomm;
4       MPI_Group cgrp, sgrp, dgrp;
5       int rc, flag, rflag, i, nc, ns, nd, crank, srank, drank;
6       /* ... */
7   redo:
8       if( comm == MPI_COMM_NULL ) {
9           /* trátase dun proceso creado polo handler */
10
11          MPI_Comm_get_parent(&icomm);
12          scomm = MPI_COMM_WORLD;
13          isNewSpawnee = 0;
14      } else {
15          /* trátase dun proceso que detectou o fallo */
16          MPIX_Comm_shrink(comm, &scomm);
17          MPI_Comm_size(scomm, &ns);
```

```
        MPI_Comm_size(comm, &nc);
        nd = nc-ns; /* numero de mortos */
        if( 0 == nd ) {
            MPI_Comm_free(&scomm);
            *newcomm = comm;
            return MPI_SUCCESS;
        }
       /* ... */
        MPI_Comm_set_errhandler( scomm, MPI_ERRORS_RETURN );

        rc = MPI_Comm_spawn(gargv[0], &gargv[1], nd, MPI_INFO_NULL,
                            0, scomm, &icomm, MPI_ERRCODES_IGNORE);
        flag = (MPI_SUCCESS == rc);
        MPIX_Comm_agree(scomm, &flag);
        if( !flag ) {
            if( MPI_SUCCESS == rc ) {
                MPIX_Comm_revoke(icomm);
                MPI_Comm_free(&icomm);
            }
            MPI_Comm_free(&scomm);

            goto redo;
        }

        MPI_Comm_rank(comm, global_rank);
        MPI_Comm_size(comm, global_procs);

    }

    /* ... */
    rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
    rflag = flag = (MPI_SUCCESS==rc);

    MPIX_Comm_agree(scomm, &flag);

    if( MPI_COMM_WORLD != scomm ) {
        MPI_Comm_free(&scomm);
    }

    MPIX_Comm_agree(icomm, &rflag);

    MPI_Comm_free(&icomm);

    if( !(flag && rflag) ) {
        if( MPI_SUCCESS == rc ) {

```

```
64              MPI_Comm_free(&mcomm);
65
66          }
67
68          goto redo;
69      }
70
71
72      /* ... */
73
74      MPI_Errhandler errh;
75
76      MPI_Comm_get_errhandler( comm, &errh );
77
78      MPI_Comm_set_errhandler( mcomm, errh );
79
80      /* ... */
81      *newcomm = mcomm;
82      MPI_Comm_rank(mcomm, global_rank);
83      MPI_Comm_size(mcomm, global_procs);
84      return MPI_SUCCESS;
85  }
```

O primeiro que se fai é comprobar se o proceso é un de nova incorporación ou se pola contra é un proceso xa existente no grupo. Para os procesos ”veteranos” o valor de *comm* nunca será `MPI_COMM_NULL`, a menos que un usuario o especifique intencionadamente. Se o proceso non é de nova creación, aplícase a reducción do comunicador coma no handler anterior, para obter un novo comunicador (despois do *spawn* do novo proceso) xa coherente, sen rastros do proceso fallido. Para forzar a creación do novo proceso e protexernos dos fallos externos derivados da creación dun proceso, utilízase o *handler* `MPI_ERRORS_RETURN` no comunicador que se usará como novo comunicador despois da reducción. Posteriormente inténtase crear o proceso. Nótese neste caso o uso de `MPI_Comm_agree`. Aquí é interesante por dous motivos:

- para asegurarnos da creación do proceso novo e repetir o procedemento sen deixar o *handler*. Débese facer así porque os acuses de recibo (ACK) dos erros xa foron notificados (pódese entender como que se busca que o *respawn* teña un enfoque transaccional e atómico).
- por eficiencia. Ao ser bloqueante podería parecer unha perda da mesma, pero dependendo do aplicativo e do entorno no que se está executando, a inicialización dun proceso podería ser moi custosa. Deste xeito asegurámonos que o novo proceso se engade correctamente ao grupo e a comunicación é funcional, polo menos mentres dure o control

do *handler*.

Se falla, habería que revogar o novo comunicador e liberar o control, e volver comezar a operación. Neste punto teremos tres comunicadores activos:

- `comm`: o comunicador orixinal, no que se produciu o erro.
- `scomm`: o comunicador que comunica o grupo de procesos excluíndo os fallantes. Obtívose por *shrinking* de comm.
- `icomm`: o comunicador asociado ao proceso creado con `MPI_Comm_spawn`.

Para conseguir comunicación entre o grupo existente e o novo, débense mesturar ambos comunicadores. Iso realízase mediante o uso de `MPI_Intercomm_merge`. A continuación compróbase a consistencia dos grupos con `MPI_Comm_agree` e, se todo é correcto, restablécese o *handler* anterior, recalcúlanse os valores de rangos e número de procesos e sobrescríbese o comunicador anterior. Así teriamos a lóxica necesaria para engadir un proceso novo.
Con todo, esta lóxica só controla o sistema dende o punto de vista do grupo de procesos que están a executar a aplicación, non aos novos. Os procesos novos non terán por defecto un mecanismo para decatarse desa comunicación, polo que o novo comunicador só sería correcto para ese grupo. Para facer que o novo proceso activamente se comunique cos outros do grupo, é necesario forzalo dende o punto de entrada do programa:

```
1  int main(int * argc, char ** argv)
2  {
3      /* ... */
4      MPI_Comm parent;
5      MPI_Comm_get_parent(&parent);
6      if (parent == MPI_COMM_NULL) {
7          /* proceso "orixinal", NON creado dende outro proceso MPI */
8          MPI_Comm_dup(MPI_COMM_WORLD, &comm);
9      } else {
10         FT_set_respawn_data(argv, &mpi_id, &NPROC);
11         int stt = attach_to_comm(&comm);
12         MPI_Comm_rank(comm, &mpi_id);
13         MPI_Comm_size(comm, &NPROC);
14         MPI_Comm_set_errhandler(comm,
       FT_RESPAWN_ON_FAILURE_HANDLER);
15         /* ... */
16     }
17     /* ... */
18 }
```

Neste pseudocódigo vese como detectar se o proceso debe "sincronizarse" cos outros ao arrancar. A detección faise a través do valor do proceso pai. En MPI, o valor devolto pola rutina `MPI_Comm_get_parent` será sempre `MPI_COMM_NULL`, excepto para aqueles procesos creados a partir dun proceso MPI (`MPI_Comm_spawn` utliza a referencia do comunicador do proceso pai para crear ao fillo, por iso necesita eses dous parámetros). Con isto en conta, fórzase ao novo proceso a executar o código da rutina `attach_to_comm`. Esta rutina simplemente é un envoltorio que actúa como punto de entrada á rutina (privada na libraría) *respawn*. Así, o proceso novamente creado si que fará a mistura de comunicadores correctamente e a comunicación será completamente funcional.

4.4.4 FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER

Como último *handler* definido na libraría búscase dar tolerancia a aplicativos con características semellantes aos obxectivos dos dous *handlers* previamente estudados, así como aos aplicativos que posuíran un número variable de procesos ou nos que interese por algún motivo ter un número variable de procesos en execución.
A implementación do mesmo non sería máis ca unha mistura dos dous *handlers* anteriores, definindo uns máximos e mínimos no número de procesos. O funcionamento do *handler* é semellante ao mecanismo de histéresis presente en moitos circuítos electrónicos, como poden ser diversos tipos de amplificadores operacionais: definiranse uns marxes superiores e inferiores delimitando o número máximo de procesos e o número mínimo de procesos que poden estar executándose á vez. Segundo vaian ocorrendo fallos, mentres o número de procesos non chegue a ese límite inferior, aplicarase unha solución de *shrinking* ata chegar ao mínimo. Unha vez chegado a ese punto, tentaranse rexenerar procesos por grupos, ou coma un bloque ata chegar ao tope.
Aparte das rutinas auxiliares para a creación de novos procesos, hai que engadir outra que nos permita configurar a histéresis do *handler*. Con iso listo, a implementación do *handler* será unha mestura dos outros dous:

```
void FT_set_spawn_threshold(int top, int bottom) {
    lower_end = bottom;
    upper_end = top;
}

void FT_elastic_respawn_on_failure(MPI_Comm * comm, int * err, ...)
{
    char error_info[MPI_MAX_ERROR_STRING];
    int error_info_length, size, rank, number_of_dead;
    MPI_Group group_f, group_c;
    int ranks_gf, ranks_gc;

```

```
13     MPI_Error_string(*err, error_info, &error_info_length);
14     MPI_Comm_size(*comm, &size);
15     MPI_Comm_rank(*comm, &rank);
16 
17     MPIX_Comm_failure_ack(*comm);
18     MPIX_Comm_failure_get_acked(*comm, &group_f);
19     MPI_Group_size(group_f, &number_of_dead);
20 
21     /* ... */
22 
23     MPI_Comm new_comm;
24 
25     if (lower_end >= (size - number_of_dead)) {
26         repair(comm);
27     } else {
28         /* mesmo código ca o respawn pero crea os procesos
       indicados */
29         respawn_multiple(*comm, comm, upper_end - lower_end);
30     }
31    /* ... */
32     MPI_Comm_group(*comm, &group_c);
33 
34     int shrinked_rank, shrinked_size;
35     MPI_Comm_rank(*comm, &shrinked_rank);
36 
37     MPI_Comm_size(*comm, &shrinked_size);
38 
39     *global_rank = (int) shrinked_rank;
40     *global_procs = (int) shrinked_size;
41 }
```

4.5 Adaptación do código para o uso dos *handlers* customizados

Coa libraría descrita na sección anterior pódese facer un código tolerante a fallos. Con todo, vai haber que facer certos cambios no código para adaptalo aos *handlers*. A cantidade e complexidade destes cambios dependerá da complexidade do *handler* que queremos utilizar. A continuación exporanse os cambios máis básicos:

4.5.1 Inicialización da libraría e dos *handlers*

A libraría desenvolvida define constantes para os *handlers*, dun xeito análogo a como o fai ULFM. Así evítase expoñer a API interna aos usuarios:

```
/* ... */
extern MPI_Errhandler ft_abort_on_failure_error_handler;
extern MPI_Errhandler ft_ignore_on_failure_error_handler;
extern MPI_Errhandler ft_respawn_on_failure_error_handler;
extern MPI_Errhandler ft_respawn_on_failure_error_handler;

/* ... */
#define FT_ERRORS_ARE_FATAL_ON_FAILURE_HANDLER MPI_ERRORS_ARE_FATAL
/* ... */
#define FT_ERRORS_RETURN_ON_FAILURE_HANDLER MPI_ERRORS_RETURN
/* ... */
#define FT_ABORT_ON_FAILURE_HANDLER
    ft_abort_on_failure_error_handler
/* ... */
#define FT_IGNORE_ON_FAILURE_HANDLER
    ft_ignore_on_failure_error_handler
/* ... */
#define FT_RESPAWN_ON_FAILURE_HANDLER
    ft_respawn_on_failure_error_handler
/* ... */
#define FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER
    ft_elastic_respawn_on_failure_error_handler
```

Tamén inclúe rutinas para a inicialización dos compoñentes da libraría que deben ser utilizados canda a inicialización de MPI:

```
MPI_Errhandler ft_abort_on_failure_error_handler = NULL;
MPI_Errhandler ft_ignore_on_failure_error_handler = NULL;
MPI_Errhandler ft_respawn_on_failure_error_handler = NULL;
/* ... */
void FT_init(void)
{
    MPI_Comm_create_errhandler(&FT_abort_on_failure,
    &ft_abort_on_failure_error_handler);
    MPI_Comm_create_errhandler(&FT_ignore_on_failure
    ,&ft_ignore_on_failure_error_handler);
    MPI_Comm_create_errhandler(&FT_respawn_on_failure,
    &ft_respawn_on_failure_error_handler);
}

void FT_finalize(void)
{
    MPI_Errhandler_free(&ft_abort_on_failure_error_handler);
    MPI_Errhandler_free(&ft_ignore_on_failure_error_handler);
    MPI_Errhandler_free(&ft_respawn_on_failure_error_handler);
```

```
17 }
18
19 void FT_set_error_handler(MPI_Comm comm, MPI_Errhandler
       error_handler)
20 {
21     MPI_Comm_set_errhandler(comm, error_handler);
22 }
```

Deberase chamar a `FT_init` despois de `MPI_init` e a `FT_finalize` xusto antes de `MPI_Finalize`. O primeiro é necesario para habilitar os *handlers* e o segundo é necesario para liberar todos os recursos da libraría correctamente. Finalmente, se ben o manexo de handlers pode ser feito coas funcións predefinidas por MPI para tal efecto, recoméndase, por coherencia, utilizar o envoltorio `FT_set_error_handler`.
No tocante ao propio código da aplicación, habería que engadir novas regras de compilación para incluír os fontes da libraría. Para iso bastaría con engadir na compilación os ficheiros `ft_aco.h` e `ft_aco.c`. Unha vez feito iso, débense importar as cabeceiras da libraría aquí descrita no código principal, duplicar o comunicador a utilizar se este é `MPI_COMM_WORLD` (necesario porque este comunicador é un valor interno de MPI, e os resultados serían catastróficos se algún handler o sobreescribira, revogara ou destruíra) e, por último, configurar o *handler* que se queira empregar. Por claridade, preséntase a continuación unha mostra desta configuración en pseudocódigo:

```
1  /* ... */
2  #include "ft_aco.h"
3  /* ... */
4
5  #if defined FT_ACO && defined FT_ABORT_ON_FAILURE
6  void cleanup(void * ptr)
7  {
8      /* ... */
9  }
10 #endif
11
12 /* ... */
13
14 int main(int * argc, char ** argv)
15 {
16     /* ... */
17     MPI_Comm comm;
18     /* ... */
19     MPI_Init(...);
20     FT_init();
21
```

```
22      MPI_Comm parent;
23      MPI_Comm_get_parent(&parent);
24      if (parent == MPI_COMM_NULL) {
25          MPI_Comm_dup(MPI_COMM_WORLD, &comm);
26      }
27
28      /* ... */
29      #ifdef FT_ACO
30          #ifdef FT_ERRORS_ARE_FATAL
31              FT_set_error_handler(comm, MPI_ERRORS_ARE_FATAL);
32          #endif
33          #ifdef FT_ERRORS_RETURN
34              FT_set_error_handler(comm, MPI_ERRORS_RETURN);
35          #endif
36          #ifdef FT_ABORT_ON_FAILURE
37              FT_set_error_handler(comm, FT_ABORT_ON_FAILURE_HANDLER);
38              FT_set_cleanup_function(cleanup, (void *)
        &ft_parameters);
39          #endif
40          #ifdef FT_IGNORE_ON_FAILURE
41              FT_set_error_handler(comm,
        FT_IGNORE_ON_FAILURE_HANDLER);
42              FT_set_respawn_data(argv, &mpi_id, &NPROC);
43          #endif
44          #ifdef FT_RESPAWN_ON_FAILURE
45              FT_set_error_handler(comm,
        FT_RESPAWN_ON_FAILURE_HANDLER);
46              FT_set_respawn_data(argv, &mpi_id, &NPROC);
47          #endif
48          #ifdef FT_ELASTIC_RESPAWN_ON_FAILURE
49              FT_set_error_handler(comm,
        FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER);
50              FT_set_respawn_data(argv, &mpi_id, &NPROC);
51              FT_set_spawn_threshold(top, bottom);
52          #endif
53      #endif
54      /* ... */
55  }
```

4.5.2 Consideracións xerais

Adicionalmente a todo o anteriormente exposto, convén ter en conta as seguintes consideracións:

- Malia que non haxa que facer referencia a ULFM no código, para poder facer uso das

ferramentas aquí descritas débese ter o entorno configurado dun xeito semellante ao descrito na primeira sección deste capítulo.

- Recoméndase utlizar operacións como `MPI_Iprobe` (*probe*) antes das recepcións bloqueantes para evitar posibles interbloqueos. Tamén pode ser útil para reducir o tempo de espera dun proceso ante un erro, posto que lanzará o erro o *probe* antes de que o faga a operación MPI.
- O uso do *probe* tamén pode acelerar o sistema, posto que executalo xusto antes de calquera comunicación MPI permitirá detectar os procesos que fallan e aplicar algunha solución tolerante a fallos, securizando a operación seguinte.
- Traballando con varios fíos convén utilizar como función de inicialización de MPI `MPI_Init_thread` co modo `MPI_THREAD_SINGLE` para evitar múltiples *spawns* ao limitar o número de fíos que poden executar as rutinas MPI.
- Todos os procedementos que usen MPI deberán ter como parámetros punteiros ás variables asociadas ao rango, número de procesos e comunicador. Isto é moi importante para que o sistema poida obter os valores correctos dos datos en todo momento sen recalcular de cada vez o rango e, sobre todo, para evitar erros ao actualizar o valor do comunicador nos *handlers*.

Capítulo 5

Resultados experimentais

Esta sección está adicada ás diversas probas de validación e rendemento usando as ferramentas descritas no capítulo anterior. Exporase o sistema no que se realizaron as probas, a metodoloxía seguida para a avaliación e a análise dos resultados obtidos.

5.1 Sistema utilizado nas probas

As probas foron feitas no clúster Pluton [9]. Pluton é un clúster heteroxéneo de computación de alto rendemento (High Performance Computing, HPC). Forma parte do centro de procesamento de datos da universidade localizado no Centro de Investigación en Tecnoloxías da Información e Comunicación (CITIC). A figura 5.1 amosa un esquema da súa arquitectura. O clúster está composto por un único punto de entrada accesible desde o exterior, denominado nodo *frontend*. Este nodo é o punto de acceso ao clúster que teñen os usuarios para editar/compilar o seu código e enviar traballos ao planificador ou sistemas de colas para a súa posterior execución nos nodos de cómputo. Polo tanto, non se permite executar traballos no nodo *frontend*. Os nodos de computación son os responsables de proporcionar os recursos computacionais do clúster como CPU, memoria e aceleradoras. Estes nodos de cálculo agrúpanse loxicamente en gabinetes de computación, onde cada nodo está asociado a un número de cabina e un número de nodo dentro da cabina. O nodo frontend tamén actúa como servidor de almacenamento conectado á rede NAS no clúster. Todos os ficheiros de usuario almacénanse neste servidor. Accédese a estes ficheiros de forma remota dende os nodos de cálculo a través da rede mediante NFS. Cada unha das cabinas agrupa un número variable de nodos de cómputo que adoitan compartir características de hardware moi semellantes. Para as probas utilizouse a cabina 0, cuxas características técnicas poden observarse na táboa 5.1.

No tocante ao software, o clúster executa *Rocks 7*, unha distribución baseada en *CentOS 7*. É unha distribución libre de Linux, a cal é compatible a nivel binario con *Red Hat Enterprise*

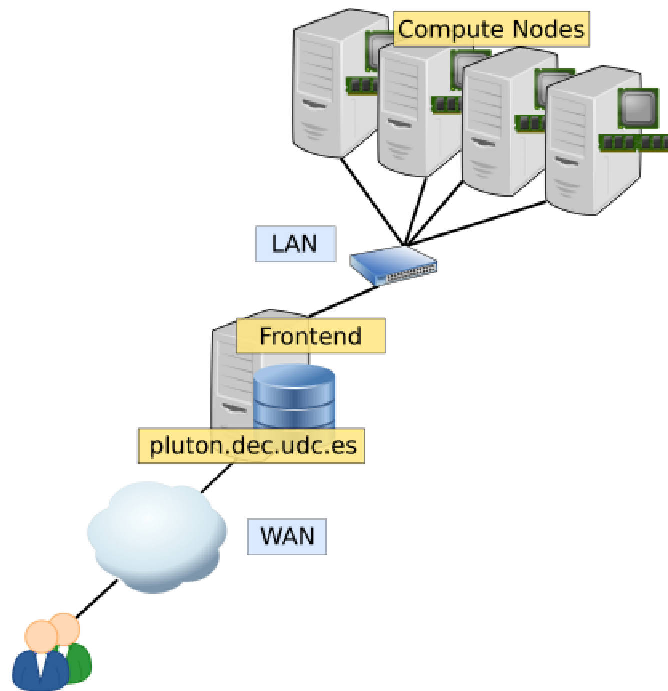

Figura 5.1: Arquitectura de Pluton

Linux. Como planificador de procesos utilízase o *Slurm Workload Manager v19.05.2* e *Lmod v8.1.18* como xestor de módudos do kernel. Tamén ten instaladas varias versións dos compiladores e ferramentas máis importantes no ámbito da computación de altas prestacións, pero para o obxectivo deste traballo tiveron que instalarse e configurarse as librarías adicionais mencionadas no capítulo anterior.

5.2 Metodoloxía utilizada nas probas

Como as solucións de tolerancia a fallos foron implementadas nunha implementación do ACO para o problema do viaxante, utilizáronse algúns conxuntos de datos do TSPLib [20] para a avaliación experimental. Estes son conxuntos de datos para o problema do viaxante de diferentes tamaños e dificultades.

Se ben o problema a resolver ten solucións exactas para os conxuntos de datos previamente mencionados, a utilización das metaheurísticas do ACO para a resolución do mesmo motiva a necesidade de tratar as probas destes sistema cun enfoque estatístico, posto que o non determinismo inherente ao código orixinal implica a non reproducibilidade dos resultados en execucións consecutivas do programa. Tendo isto en conta, a execución das probas consistiría

Modelo de CPU	2 × Intel Xeon E5-2660 Sandy Bridge-EP
Velocidade CPU	3.0 GHz
Núcleos por CPU	8
Fíos por núcleo	2
Tamaño da caché L1	32 KiB
Tamaño da caché L2	256 KiB
Tamaño da caché L3	20 MiB
Memoria RAM	64 GiB DDR3 1600 Mhz (Dual channel)
Discos	1 × HDD 1 TiB SATA3 7.2K rpm
Aceleradoras	1 × NVIDIA Tesla Kepler K20m 5 GiB GDDR5 (0-0-12) 1 × NVIDIA Tesla Kepler K40c 12 GiB GDDR5 (0-13) 3 × NVIDIA Tesla Kepler K20m 5 GiB GDDR5 (0-15) 2 × Intel Xeon Phi 5110P 8 GiB GDDR5 (0-16)
Redes	InfiniBand FDR & Gigabit Ethernet

Tábox 5.1: Especificacións técnicas da cabina 0 utilizada para as probas

nun conxunto de execucións coas mesmas condicións iniciais, cuxos resultados serán tratados estatísticamente. Para poder ter resultados estatísticamente relevantes cada experimento consistiu de cincuenta execucións consecutivas, coa fin de minimizar o impacto de resultados excesivamente malos ou excesivamente bos (ambos casos se poden dar nun algoritmo baseado en heurísticas por simple sorte derivada da exploración pseudoaleatoria dos estados iniciais do problema). Outro aspecto importante á hora de lidiar con este tipo de problemas é escoller correctamente o criterio de parada a utilizar. Debido a que estes algoritmos son estocásticos, cunha dispersión nos resultados obtidos en canto a tempo de execución e iteracións moi elevada, débese optar por un criterio que permita obter resultados significativos sen sacrificar unha excesiva cantidade de horas de computación.
Concretamente, utilizouse un criterio de parada por calidade, o cal consiste en fixar o valor obxectivo a acadar por cada execución no experimento. Neste caso ese valor poderá ser o valor óptimo real, se se está a probar cun problema cuxo tamaño e dificultade permita a resolución de cada execución nun tempo razoable, ou un valor non óptimo prefixado por nós en caso contrario. O paquete software ACO utilizado neste TFG permite introducir o valor obxectivo do problema a resolver por liña de comandos, a través do parámetro `-o`.

Hipótese de avaliación: usando este criterio espérase obter valores máis altos en número de

iteracións necesarias para resolver o problema, así como maior tempo total de computación cando o número de fallos que experimenta o sistema aumenta. Para o caso dos *handlers* que realicen *spawning* de procesos, estes valores deberían ser menores ca nos outros casos. O valor obxectivo promedio obtido tras cada experimento debería estar comprendido entre o valor obxectivo promedio do experimento con N procesos sen inxección de fallos e o valor obxectivo promedio do experimento con 1 proceso e sen inxección de fallos.

5.2.1 Inxección de fallos

A inxección de fallos realizouse manualmente ao final de certas iteracións do ACO nas que se simula un fallo mediante o uso do sinal `SIGKILL` para forzar a terminación abrupta dos procesos. Intentouse facer nun espazo de iteracións regular simulando un sistema real, posto que nestes casos os fallos non serán grupais, senón que xeralmente iran asociados ao MTBF calculable en función da fiabilidade individual dos compoñente que a compoñen. No seguinte pseudocódigo pódese observar a aproximación que se seguiu para cada *handler* posto a proba:

```
1  int main(...)
2  {
3      /* ...*/
4      while (!is_termination_condition_met(...))
5      {
6      /* ... */
7          construct_ant_solutions(...);
8          apply_some_search_method(...); // optional
9          update_pheromones(...);
10         /* ... */
11         #ifdef KILL_AT_50
12         if (iteration == 5 && mpi_id == NPROC - 1) {
13             raise(SIGKILL);
14         }
15         #endif
16
17         #ifdef KILL_ALL_BUT_ONE
18         if (iteration % kill_ratio_iteration == 0) {
19             if (mpi_id != 0 && mpi_id == NPROC - i) {
20                     raise(SIGKILL);
21             }
22             i++;
23         }
24         #endif
25         /* ... */
26     }
27 }
```

O valor de `kill_ratio_iteration` dependerá de cada proba e estará axustado para que sexa nos primeiros compases da execución, sendo maior canto maior sexa o número de iteracións necesario para completar o problema TSP escollido.

5.2.2 Saída de datos e tratamento estatístico

O código orixinal estaba preparado para xerar ficheiros cos resultados de cada execución do código, polo que simplemente se adaptou levemente para que formateara a información necesaria do xeito máis cómodo posible que facilite o análise posterior. A continuación amósase un anaco dun posible ficheiro de resultados:

```
Best: 42029;   Iteration: 39   Found at time 0.482499  Total time:0.50
Best: 42029;   Iteration: 29   Found at time 0.352837  Total time:0.37
Best: 42029;   Iteration: 41   Found at time 0.443432  Total time:0.45
Best: 42029;   Iteration: 34   Found at time 0.398879  Total time:0.41
Best: 42029;   Iteration: 308  Found at time 2.556103  Total time:2.56
Best: 42029;   Iteration: 34   Found at time 0.414223  Total time:0.44
Best: 42029;   Iteration: 31   Found at time 0.360756  Total time:0.37
Best: 42029;   Iteration: 35   Found at time 0.386737  Total time:0.40
Best: 42029;   Iteration: 209  Found at time 1.596187  Total time:1.61
Best: 42029;   Iteration: 38   Found at time 0.419540  Total time:0.43
Best: 42029;   Iteration: 35   Found at time 0.407886  Total time:0.42
Best: 42029;   Iteration: 38   Found at time 0.422656  Total time:0.43
Best: 42029;   Iteration: 41   Found at time 0.434219  Total time:0.45
.
.
.
```

O tratamento estatístico dos datos fíxose utilizando unha folla de cálculo, posto que os datos presentáronse nun formato válido para que esta os procese fácilmente. Para cada experimento obtivéronse os valores máximos, mínimos e promedios en todos os parámetros extraídos. Con

Conxunto de datos	Número de puntos	Óptimo
lin318	318	42029
rat738	738	8806

Táboa 5.2: Características dos *benchmarks* utilizados nas probas

todo, os parámetros relevantes son:

- *Best*: mellor valor obtido. Corresponderase coa mellor ruta (mínima distancia) obtida que conecta as cidades. Nas probas realizadas este correspóndese sempre co valor óptimo
- *Iteration*: número de iteracións que necesitou o algoritmo para obter a mellor ruta.
- *Total time*: tempo total de computación por execución. Interésanos este valor para ver o *overhead* que meten os mecanismos de tolerancia a fallos implementados.

5.3 Análise de resultados

Realizáronse probas preliminares con multitude de solucións de tolerancia a fallos, distintos enfoques de inxección de erros e diferentes tamaños e complexidades do problema do viaxante. Durante o transcurso das mesmas, optouse por descartar as probas dos *handlers* `MPI_ERRORS_ARE_FATAL` e `FT_ABORT_ON_FAILURE_HANDLER` xa que en ningún caso ofrecen tolerancia a fallos propiamente dita. No caso do segundo, si que pode fornecer certo nivel de recuperación tras desastres. Os plans de recuperación tras desastres son mecanismos para mitigar o impacto producido nun sistema cando as medidas de tolerancia a fallos non foron efectivas. Xeralmente estes mecanismos están enfocados a nivel hardware, porque soen estar vencellados a grandes fallos masivos de infraestrutura. Con todo, podemos considerar un sistema software como un sistema reparable, e que no caso no cal todas as medidas de tolerancia a fallos non cumpran o seu papel, se opte por este sistema de recuperación tras desastres coma un xeito de diminuír o seu MTTR. Para o caso concreto que nos compete neste proxecto, poderíase utilizar nun entorno real no que o aplicativo a utilizar teña uns requirimentos que non permitan a aplicación dos *handlers* anteriormente mencionados. Nalgúns casos onde a inicialización do sistema é pouco custosa, podería incluso beneficiar en termos de rendemento o reinicio do sistema a partir de datos dunha execución anterior deixados pola rutina de *cleanup* do propio *handler*. Deixouse fóra tamén os experimentos co *handler* `FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER` porque os resultados pódense inferir dos obtidos por outros experimentos, ao tratarse dunha combinación de solucións.

Asimesmo, as probas preliminares permitiron seleccionar dous *benchmarks* que serían candidatos a obter resultados interesantes para o ámbito deste traballo. Decidiuse utilizar os conxuntos de datos *lin318* e *rat783*, extraídos da TSPLib, por ser problemas simples sen mínimos locais e cun tempo de execución sen inxección de erros pequeno, co obxectivo de obter un conxunto de probas o cal poida ser executado nun tempo razoable. Na táboa 5.2 pódense ver as características destes conxuntos de datos. Para cada conxunto de datos defínense oito experimentos:

- **EXP1**: execución con 16 procesos sen inxección de erros (para obter o "mellor" resultado teórico).
- **EXP2**: execución con 1 proceso (secuencial) sen inxección de erros (para obter o "peor" resultado teórico).
- **EXP3**: execución co *handler* `MPI_ERRORS_RETURN` terminando abruptamente 1 proceso (tolerancia de 1 fallo).
- **EXP4**: execución co *handler* `MPI_ERRORS_RETURN` terminando progresivamente (1 proceso cada x iteracións) 15 procesos (tolerancia de $N-1$ fallos).
- **EXP5**: execución co *handler* `FT_IGNORE_ON_FAILURE_HANDLER` terminando abruptamente 1 proceso (tolerancia de 1 fallo).
- **EXP6**: execución co *handler* `FT_IGNORE_ON_FAILURE_HANDLER` terminando progresivamente (1 proceso cada x iteracións) 15 procesos (tolerancia de $N-1$ fallos).
- **EXP7**: execución co *handler* `FT_RESPAWN_ON_FAILURE_HANDLER` terminando abruptamente 1 proceso (tolerancia de 1 fallo).
- **EXP8**: execución co *handler* `FT_RESPAWN_ON_FAILURE_HANDLER` terminando progresivamente (1 proceso cada x iteracións) 15 procesos (tolerancia de $N-1$ fallos).

Os experimentos paralelos executáronse con 16 procesos nun nodo do clúster. Isto implica o uso de 2 CPU, e 8 núcleos por CPU. Os experimentos secuenciais ocuparon 1 nodo, do cal só se utilizou 1 núcleo dunha das CPU dese nodo.

5.3.1 Análise de resultados para o problema *lin318*

Para este problema abortouse un proceso cada iteración, xa que ao ser un problema pequeno converxe en moi poucas iteracións e incrementar este valor facía que houbera casos nos que o sistema rematara antes de terminar todos os procesos necesarios.

Como se pode observar na táboa 5.3, unha execución sen erros con 16 colonias (procesos) complétase cun número de iteracións promedio moi próximo a 43, e nun tempo promedio de 0.46 segundos de tempo de CPU. Como o algoritmo que soluciona o problema é estocástico, vai presentar moita variabilidade, aínda en experimentos semellantes (se se realizan varios experimentos semellantes obteranse resultados parecidos, pero distintos). Por outra banda, o peor caso sería aquel no que non podemos aproveitar o paralelismo implementado, é dicir, o caso de execución sen erros secuencial (1 colonia/proceso). Neste caso, o número promedio de iteracións por execución é derredor de 313 iteracións, o que supón un aumento de case 7.5 veces do esforzo necesario para acadar a solución óptima con respecto ao caso paralelo. Segundo a hipótese de avaliación proposta na metodoloxía, todas as solucións tolerantes a fallos deberían de ofrecer valores peores ou iguais ca a execución paralela sen erros e mellores ca execución secuencial (en promedio e tendo en conta a desviación estándar asociada á mostra estatística).
Nos experimentos que usan o *handler* `MPI_ERRORS_RETURN` podemos observar que coa inxección de 1 só fallo os resultados correspóndense cos esperados (tendo en conta a variabilidade inherente ao sistema) ao obter un valor case semellante á execución paralela sen fallos tanto en iteracións realizadas ata atopar o valor óptimo coma no tempo de cómputo total. Estes resultados teñen sentido, posto que o sistema comportaríase tras o fallo como se execución fose con 15 procesos o cal, vendo a aceleración obtida entre os casos sen fallo, esa variación de tan só 1 colonia case non afectaría ao rendemento. No experimento coa inxección de 15 fallos, as cousas cambian. Neste caso o rendemento promedio degrádase en case un 700%, achegándose moito aos valores obtidos para o caso secuencial. O tempo de CPU tamén aumenta consecuentemente en liña co observado nos casos sen inxección de fallos, non chegando a superar o tempo de execución correspondente á execución secuencial, o que implica que o *overhead* debido á execución do handler é moi pequeno. A nivel teórico, este valor aumentaría en función do número de comunicacións e do número de fallos inxectados. Lémbrese que este *handler* non realiza realmente ningunha función máis ca devolver os códigos de erro, o sistema a nivel global non se decataría de posibles fallos nos procesos. Isto implica que:

- cada proceso seguirá cos mesmos valores de rango e número de procesos (16) durante toda a execución, independentemente do número de fallos.
- realizará todas as operacións non bloqueantes, indepentemente do estado dos demais procesos.
- bloqueará o sistema ante unha operación bloqueante na que interveña un proceso fallido (non é o caso pois neste código non hai operacións bloqueantes).
- como os erros non son notificados e non se toma ningunha medida de contención, nun

EXP	*N*	*Handler*	Fallos	Iteracións promedio	Tempo promedio (s)
EXP1	1	-	-	313.18 +/- 381.16	2.04 +/- 2.36
EXP2	16	-	-	42.98 +/- 51.59	0.46 +/- 0.39
EXP3	16	MPI_ERRORS_RETURN	1	40.28 +/- 26.59	0.45 +/- 0.18
EXP4	16	MPI_ERRORS_RETURN	15	275.88 +/- 315.62	2.02 +/- 2.13
EXP5	16	FT_IGNORE_ON_FAILURE_HANDLER	1	46.80 +/- 53.37	0.49 +/- 0.38
EXP6	16	FT_IGNORE_ON_FAILURE_HANDLER	15	231.82 +/- 237.66	1.75 +/- 1.66
EXP7	16	FT_RESPAWN_ON_FAILURE_HANDLER	1	38.00 +/- 9.69	0.43 +/- 0.08
EXP8	16	FT_RESPAWN_ON_FAILURE_HANDLER	15	264.98 +/- 244.60	1.96 +/- 1.66

Táboa 5.3: Resultados para os experimentos con parada por calidade (óptimo=49029) para o problema *lin318*

aplicativo onde houbera moitísimas máis comunicacións ca estudada neste TFG, podería producir moito *overhead*, posto que por calquera operación MPI punto a punto cun proceso nun estado inválido (terminado) executaríase o *handler*.

Movéndonos aos experimentos nos que se fai uso do *handler* `FT_IGNORE_ON_FAILURE_HANDLER`, os resultados son semellantes aos obtidos nos experimentos do *handler* anterior. En ambos casos se aprecian uns valores semellantes aos esperados e en ningún caso son peores ca no caso secuencial. Non se aprecia ningún *overhead* provocado polo *shrinking* do comunicador.

No tocante aos experimentos que utilizan `FT_RESPAWN_ON_FAILURE_HANDLER`, se ben coa inxección de 1 fallo se obteñen os resultados esperados, non ocorre así coa inxección dos 15 fallos. Neste caso, non se nota mellora con respecto aos outros experimentos. Isto débese ao tamaño do problema. Debido a que o problema é moi pequeno, non dá tempo de restaurar todas as colonias antes de chegar ao óptimo e por iso se obteñen uns resultados semellantes. Profundizando nesta casuística, o tempo necesario para manexar o fallo, reducir o comunicador e *respawnear* é maior ca o necesario para resolver o problema con menos colonias. Na figura 5.2 pódese ver unha gráfica coa dispersión dos resultados para cada experimento realizado. Cada punto representa o número de iteracións executadas para acadar o valor óptimo en cada unha das 50 execucións do código para cada un dos experimentos. Non se aprecia ningunha desviación entre os experimentos con inxección de fallos e os experimentos sen inxección de fallos.

5.3.2 Análise de resultados para o problema *rat783*

Neste problema abortouse un proceso cada 16 iteracións, tendo un impacto significativo na obtención de valores aceptables pero sen terminar todos os procesos demasiado pronto. A táboa 5.4 amosa os resultados obtidos. Podemos observar que a paralelización xoga un papel máis importante no rendemento ca no conxunto de datos anterior, debido principalmente

EXP	*N*	***Handler***	**Fallos**	**Iteracións promedio**	**Tempo promedio (s)**
EXP1	1	-	-	2952.50 +/- 1957.25	41.88 +/- 27.55
EXP2	16	-	-	339.30 +/- 222.00	6.67 +/- 3.95
EXP3	16	MPI_ERRORS_RETURN	1	315.38 +/- 126.50	6.18 +/- 2.16
EXP4	16	MPI_ERRORS_RETURN	15	2110.05 +/- 1793.32	41.89 +/- 39.66
EXP5	16	FT_IGNORE_ON_FAILURE_HANDLER	1	385.40 +/- 227.32	7.44 +/- 4.07
EXP6	16	FT_IGNORE_ON_FAILURE_HANDLER	15	1909.98 +/- 2103.16	272.60 +/- 125.12
EXP7	16	FT_RESPAWN_ON_FAILURE_HANDLER	1	302.03 +/- 90.74	5.90 +/- 1.39
EXP8	16	FT_RESPAWN_ON_FAILURE_HANDLER	15	352.43 +/- 130.68	6.09 +/- 1.70

Táboa 5.4: Resultados para os experimentos con parada por calidade (óptimo=8806) para o problema *rat783*

ao incremento de tamaño do problema do viaxante a resolver. Os datos obtidos co handler `MPI_ERRORS_RETURN` son semellantes aos obtidos co problema anterior.

No tocante aos experimentos que usan `FT_IGNORE_ON_FAILURE_HANDLER`, o comportamento é o esperado cando se inxecta un só fallo, os resultados son semellantes á execución paralela sen fallos e non se aprecia ningún *overhead* significativo. Isto cambia cando se fai a inxección de 15 fallos. Neste caso aparece un gran *overhead* (derredor dun 660%) en canto a tempo de CPU. A priori non debería haber motivo para este comportamento, pero revisando os datos desglosados pódese observar que este incremento de tempo de computación non vai asociado a un incremento de número de iteracións. Disto podemos concluír que o algoritmo segue sendo igual de eficiente, pero o que aumentou foi o tempo por iteración. O único que podería facer aumentar así o tempo de computación por iteración é un fallo masivo das comunicacións en MPI. Sen embargo, nos experimentos con *respawn* o procesamento no *handler* é moito máis demandante, así como que realízase o mesmo proceso de *shrinking* do comunicador, pero non se aprecia ningún *overhead.* A causa máis probable é a falta do uso da función `MPI_Comm_Agree` neste *handler*, pois non foi engadida por motivos de eficiencia. A falta desta primitiva podería estar provocando que por mala sincronización dos procesos á hora de notificar e propagar o fallo (que se revoge o comunicador cando moi poucos procesos están involucrados en operacións MPI, pois lembremos que o descubrimento de fallos é local e asíncrono para cada grupo de procesos) se produza unha situación de erros en cascada que faga ao sistema entrar múltiples veces no handler para cada fallo. Tamén se pode deber a que o MTBF do fallo inxectado coincida en tempo de execución co tempo que tarda o sistema en reconstruír as comunicacións. Isto tamén causaría un overhead moi grande ao encadearse os erros. Como se veu antes, nun entorno real a tasa de fallos corresponderase cunha métrica da fiablidade individual de cada un dos compoñentes do sistema, polo que ocorrerán fallos en intervalos cercanos a MTBF horas; pola contra, nestas probas os fallos son forzados en

intervalos de decenas de segundos ou segundos, deixando moito menos tempo de reacción as estratexias implementadas o que pode causar comportamentos estraños coma este.

Finalmente, para os experimentos nos que se utiliza FT_RESPAWN_ON_FAILURE_HANDLER os resultados correspóndense cos esperados, posto que pódese observar que o *spawning* de novas colonias permite obviar completamente calquera número de fallos existente sen ningún tipo de *overhead* aparente.

Na figura 5.3 pódese ver unha gráfica coa dispersión dos resultados para cada experimento. Coma nos experimentos utilizando o *benchmark lin318*, non se aprecia ningunha desviación entre os experimentos con inxección de fallos e os experimentos sen inxección de fallos.

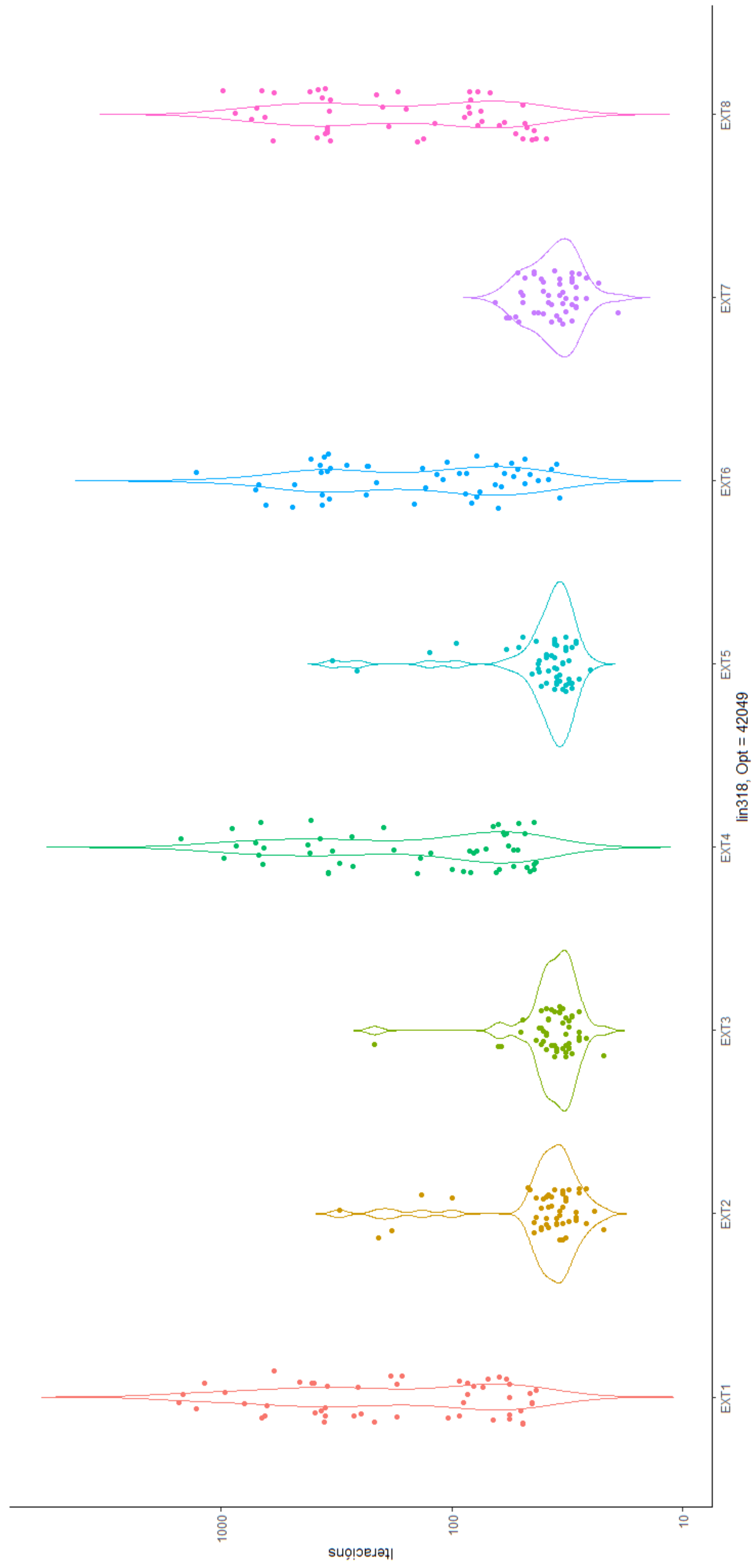

Figura 5.2: Dispersión dos resultados por experimento para o problema *lin318*

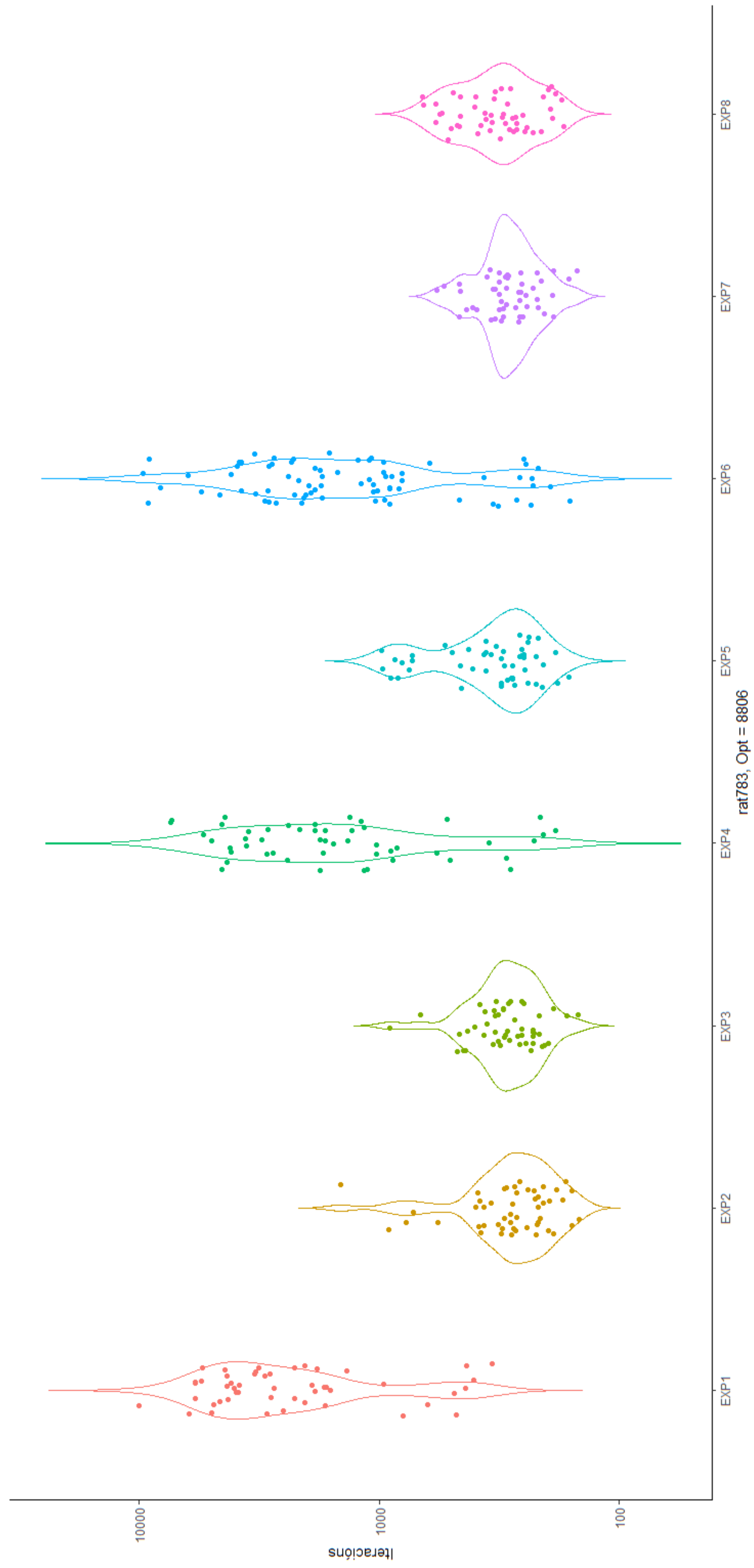

Figura 5.3: Dispersión dos resultados por experimento para o problema *rat783*

Capítulo 6

Conclusións e liñas futuras

NESTE proxecto preséntase un estudo de diferentes estratexias de tolerancia a fallos aplicándoas concretamente a unha versión paralela do algoritmo de optimización por colonia de formigas adaptado para a resolución do problema do viaxante.

Dentro das tarefas principais do proxecto atópanse o estudo pormenorizado do ACO, así como a implementación paralela, a súa aplicación ao problema do viaxante, o estudo da proposta de estándar ULFM de MPI para o tratamento da tolerancia a fallos en programas paralelos e a implementación de solucións de tolerancia a fallos baseadas nesa proposta de estándar.

Para a implementación das solucións de tolerancia a fallos, estudáronse as posibles aplicacións das solucións fornecidas por UFLM e implementáronse solucións propias para a xestión de erros partindo desa tecnoloxía. Concretamente desenvolvéronse os seguintes *handlers*:

- `FT_ABORT_ON_FAILURE_HANDLER`: termina a execución de xeito controlado, permitindo ao usuario o control dos recursos do sistema para evitar perda de información, liberar memoria e recursos compartidos.

- `FT_IGNORE_ON_FAILURE_HANDLER`: detecta o erro, notifícao, propágao e reduce o grupo de procesos a un estado consistente. Despois crea un novo comunicador para que a comunicación entre os procesos se restableza.

- `FT_RESPAWN_ON_ERROR_HANDLER`: detecta o erro e inicia un novo proceso que sustituirá ao proceso fallido. Contén toda a lóxica necesaria para que o inicio, a incorporación do proceso ao grupo e o establecemento da comunicación cos outros procesos do grupo sexa transparente ao usuario.

- `FT_ELASTIC_RESPAWN_ON_FAILURE_HANDLER`: mestura entre os dous *handlers* anteriores. Permite flexibilidade ao usuario para ignorar os erros ou reemplazar procesos fallidos segundo as demandas da aplicación en cada momento.

As implementacións trataron de xeneralizarse ao máximo de xeito que poidan ser utilizadas directamente por outras aplicacións para dar certo nivel de tolerancia a fallos a un sistema paralelo. O resultado é unha pequena libraría que proporciona diferentes niveis de tolerancia a fallos - dependendo do *handler* escollido - a calquera código paralelo que use o estándar MPI, cunha modificación mínima do aplicativo orixinal. Todos os códigos desenvolvidos están publicamente dispoñibles en Github: https://github.com/meluchoMZ/ACOTSP-MPI-OMP/tree/develop. Adicionalmente, neste TFG estudouse o impacto no rendemento da diferentes alternativas, tomando como caso de estudo unha versión paralela do algoritmo ACO aplicado ao problema do viaxante, analizando tamén as vantaxes, problemas e aplicacións recomendadas para cada unha das solucións propostas.

Este traballo pode axudar aos desenvolvedores de software a fornecer de tolerancia fallos a aplicativos paralelos implementados con MPI, sendo capaz de tolerar entre 1 e $N-1$ fallos (N sendo o número de procesos cos que se executa o aplicativo) e cun impacto mínimo no rendemento na maior parte dos casos.

Segundo a computación de altas prestacións está presente na vida diaria de investigadores e en todos os procesos de produción industrial, e co advenimento da computación exaescala, a creación de sistemas paralelos tolerantes a fallos será cada vez máis demandada no futuro. Este traballo pódese entender como o precursor do que podería ser unha implementación dunha librería de tolerancia a fallos estándar. É esperable que se se aproba a extensión ULFM sobre MPI, os diferentes proveedores da tecnoloxía proporcionen aos usuarios pequenas librarías que permitan a implantación desta solucións nun maior número de aplicacións cun menor esforzo de desenvolvemento.

6.1 Conclusións sobre a tecnoloxía subxacente

User Level Failure Mitigation (ULFM) busca dende a súa concepción convertirse nun estándar que brinde as capacidades de tolerancia a fallos ao estándar MPI. Se ben a tecnoloxía é moi interesante e está nun estado avanzado e funcional, carece de ferramentas que axilicen a súa utilización. Ademáis, a falta de documentación extensa e pormenorizada complica a comprensión da tecnoloxía e dificulta moito o desenvolvemento ao engadir unha capa de complexidade enriba duns sistemas que xa son intrínsicamente complexos. O propio estándar de MPI tamén dificulta a implantación de solucións baseadas en ULFM debido a que non obriga á implementación das bases funcionais que permiten que ULFM funcione, pero supoño que isto cambiará cando sexa definitivamente aprobada esta extensión. Con todo, a tecnoloxía é moi prometedora e malia que avanza lentamente, debido ao pequeno equipo de desenvolvemento

que o compón, é unha boa candidata a converterse en parte do estándar no futuro.

6.2 Relación coa titulación

Para a realización deste TFG foi necesario o uso e extensión de diversas competencias adquiridas durante os estudos do grao. Dende o comezo do proxecto, é necesario realizar un análise dos requerimentos do mesmo e establecer unha planificación. Despois débese realizar un análise dos requirimentos propios do sistema a desenvolver, elaborar un deseño o máis desacoplado posible e implementalo.

A comprensión do sistema de partida implicou o uso de competencias de algoritmia, complexidade computacional, programación de linguaxes imperativos, concorrencia e paralelismo e arquitecturas multiprocesador e multicomputador. Para o desenvolvemento do sistema tolerante a fallos foron necesarias esas competencias, pero tamén outras a maiores como son o deseño de software, a compresión e coñecemento de sistemas de tolerancia a fallos, administración e operación de infraestruturas informáticas, o manexo de scripts de linux e coñecementos de compilación e linkado na linguaxe *C*.

En síntese, este proxecto permitiume poñer en práctica os coñecementos e competencias adquiridos na titulación, así como servir como introdución ao ámbito da investigación.

6.3 Liñas futuras

Segundo o visto no capítulo anterior, a libraría desenvolvida introduce un *overhead* mínimo na maior parte dos casos. Con todo, sería interesante extender as probas a máis conxuntos de datos da libraría TSPLib para investigar máis a fondo o *overhead* detectado no *handler* `FT_IGNORE_ON_FAILURE_HANDLER` no problema *rat783*, estudar a reproducibilidade do mesmo e propoñer solucións que o amortigüen.

Bibliografía

[1] M. Dorigo and T. Stützle, *Ant Colony Optimization*, 1st ed. The MIT Press, 2004.

[2] G. Brassard, P. Bratley, and R. G.-B. Giner, *Fundamentos de algoritmia.* Prentice Hall Madrid, 1997, vol. 3, no. 5.1.

[3] C. Blum, "Ant colony optimization: Introduction and recent trends," *Physics of Life reviews*, vol. 2, no. 4, pp. 353–373, 2005.

[4] "User Level Failure Mitigation." [En liña]. Dispoñible en: https://fault-tolerance.org/wp-content/uploads/2012/10/20170221-ft.pdf

[5] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of MPI applications in exascale systems: The ULFM solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, 2020.

[6] "Message Passing Interface." [En liña]. Dispoñible en: https://www.mpi-forum.org/docs/

[7] "ACO-MPI-OMP." [En liña]. Dispoñible en: https://zenodo.org/record/5711353#.YwuOznHP23A

[8] P. González, R. R. Osorio, X. C. Pardo, J. R. Banga, and R. Doallo, "An efficient ant colony optimization framework for HPC environments," *Applied Soft Computing*, vol. 114, p. 108058, 2022.

[9] "Clúster Pluton." [En liña]. Dispoñible en: http://pluton.dec.udc.es/

[10] "Repositorio coas ferramentas implementadas." [En liña]. Dispoñible en: https://github.com/meluchoMZ/ACOTSP-MPI-OMP/tree/develop

[11] "ULFM tutorial SC'20." [En liña]. Dispoñible en: https://fault-tolerance.org/2020/10/24/sc20-tutorial/

[12] G. Brassard, P. Bratley, and R. G.-B. Giner, *Fundamentos de algoritmia.* Prentice Hall Madrid, 1997, vol. 3, no. 5.1.

[13] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," *Local search in combinatorial optimization*, vol. 1, no. 1, pp. 215–310, 1997.

[14] M. Dorigo, "Optimization, learning and natural algorithms," *Ph. D. Thesis, Politecnico di Milano*, 1992.

[15] "MPICH." [En liña]. Dispoñible en: https://www.mpich.org/

[16] "OpenMPI." [En liña]. Dispoñible en: https://www.open-mpi.org/

[17] "OpenMP." [En liña]. Dispoñible en: https://www.openmp.org/

[18] I. Koren and C. M. Krishna, *Fault-tolerant systems.* Morgan Kaufmann, 2020.

[19] "Sueldo del ingeniero informático en españa." [En liña]. Dispoñible en: https://www.jobted.es/salario/ingeniero-inform%C3%A1tico

[20] "TSPLib." [En liña]. Dispoñible en: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/