Universidad Central “Marta Abreu” de Las Villas

Facultad de Matemática, Física y Computación

TRABAJO PARA OPTAR POR EL TÍTULO ACADÉMICO
MÁSTER EN CIENCIA DE LA COMPUTACIÓN

DESARROLLO DE UN PLANIFICADOR PARA LA PLATAFORMA DE CÁLCULO DISTRIBUIDO T-ARENAL.

Autor:
Luis Antonio García González

Tutores:
César Raúl García Jacas
Liesner Acevedo Martinez
Consultante:
Dirk Roose

2018

Centro de Estudios de Matemática Computacional

Dedicatoria

Para Ñico.

Agradecimientos

A mi abuelo Ñico, por la atención y por prepararme para la vida.

A mis padres Sara Vivian y Luis Antonio, por su apoyo y estímulo. Esta tesis es también de ellos.

A Adriana, por confiar en mí y darme tanto amor.

A mi abuela por todo su cariño.

A mi hermana y Carlos por los consejos.

A mis tíos por sus cursos de economía, calidad y para la vida.

A mis tías por su preocupación y ocupación.

A Mario Ernesto por la ayuda.

A Ramoncito y Landy por las incontables horas de conversación.

A Dr.C Angel Cano y Yamilet Pompa por aquella P3. Supo a gloria.

A mis tutores; Liesner por confiar en mí y Cesar por apoyarme y aconsejarme.

A Dirk Roose que sin él no se hubiera podido realizar este trabajo.

A la profesora Maria Matilde por ayudarme siempre.

A Mario por ser tutor sustituto.

A Keidy, Nilda, Rosa y Oristela por revisar.

Al CEMC, el viejo y el nuevo, por educarme y prepararme.

Resumen

En el presente trabajo se discuten diferentes técnicas de planificación de tareas para sistemas distribuidos en ambientes no dedicados así como la adaptación de estas a la plataforma de cálculo distribuido T-arenal. Esta plataforma fue diseñada para ejecutar tareas complejas en cuanto al tiempo, sobre ordenadores desplegados en los laboratorios docentes y conectados por una red local. Inicialmente se describe el paradigma de computación voluntaria así como plataformas que lo implementan. Acerca de estas plataformas se mencionan sus características fundamentales y las aplicaciones que se pueden ejecutar en ellas. También se mencionan los algoritmos de planificación de tareas *self-scheduling*, estrategias desarrolladas inicialmente para la planificación de iteraciones de ciclos paralelos en sistemas homogéneos, donde las iteraciones son independientes entre ellas. Las principales estrategias *self-cheduling* son implementadas sobre T-arenal, utilizando el algoritmo LINPACK para estimar la capacidad computacional de las estaciones de trabajo y poder determinar la cantidad de tareas a enviar en cada pedido. Finalmente se realizaron pruebas de rendimiento ejecutando dos aplicaciones sobre cada una de las estrategias de planificación implementadas sobre la plataforma. Las pruebas realizadas muestran como los nuevos planificadores reducen el tiempo de ejecución y la cantidad de comunicaciones necesarias para completar las tareas en la plataforma.

Abstract

The present research discusses different task scheduling techniques for distributed systems in non-dedicated environments as well as their adaptation to distributed platform T-arenal. This platform was designed to execute complex tasks in terms of time on computers deployed in educational laboratories and connected by a local network. Initially, the paradigm of volunteer computing is described as well as the platforms that implement it. About these platforms are mentioned their fundamental characteristics as well as the applications that can be executed in them. Also, the dynamic strategies self-scheduling to allocate a set of tasks are mentioned, strategies initially developed for the scheduling of parallel cycle iterations in homogeneous systems, where iterations are independent of each other. The main self-cheduling strategies are implemented on T-arenal, using LINPACK benchmark to estimate the computational capacity of the workstations and to determine the amount of tasks to send in each order. Finally, performance tests of two applications were performed on each of the scheduling strategies implemented on the platform. The tests show how the new schedulers reduce the execution time and the amount of communications needed to complete the tasks on the platform.

Índice general

Dedicatoria **I**

Agradecimientos **II**

Resumen **III**

Abstract **IV**

Introducción **1**

1. Computación Voluntaria y estrategias *self-scheduling* **6**

1.1. Introducción 6

1.2. Computación voluntaria 6

1.2.1. Ejemplos de proyectos de computación voluntaria 7

1.2.1.1. GIMPS 7

1.2.1.2. Distributed.net 7

1.2.1.3. Quake-Catcher Network (QCN) 7

1.2.1.4. Berkeley Open Infrastructure for Network Computing (BOINC) 8

1.2.1.5. XtremWeb 8

1.2.1.6. HTCondor 8

1.2.1.7. T-arenal 9

1.2.2. Arquitectura de sistemas de computación voluntaria 10

1.2.3. Aplicaciones 10

1.2.4. Seguridad 11

1.2.5. Políticas de planificación de tareas 11

1.3. Planificación de tareas con algoritmos **Self-scheduling** 12

1.3.1. Algoritmos self-scheduling 13

1.3.2. *Guided Self-scheduling(GSS)* 14

1.3.3. *Factoring Self-scheduling(FSS)* 14

1.3.4. *Trapezoid Self-scheduling(TSS)* 14
1.3.5. *Weighted Factoring self-scheduling* 15
1.3.6. *Quadratic, Exponential and Root Self-scheduling(QSS),(ESS),(RSS)* 15
1.3.7. *Heterogeneous Dynamic Self-scheduling(HDSS)* 16
1.3.8. *Self-scheduling* para sistemas distribuidos 17
1.4. Conclusiones parciales 19

2. Estrategias self-scheduling para la plataforma de cálculo distribuido T-arenal 20
2.1. Introducción 20
2.2. Plataforma de cálculo distribuido T-arenal 20
2.2.1. Planificación a nivel de servidores de peticiones 22
2.2.2. Planificación a nivel de clientes 22
2.2.3. Mecanismo de colaboración 22
2.2.4. API de desarrollo 23
2.3. Adaptación a la plataforma de las estrategias self-scheduling 23
2.3.1. Peso de los clientes 23
2.3.2. Guided Self-Scheduling 24
2.3.3. Factoring Self-Scheduling 24
2.3.4. Trapezoid Self-Scheduling 24
2.4. Modificaciones en el diseño de la plataforma 25
2.4.1. Servidor de Peticiones 25
2.4.2. Cliente 28
2.5. Conclusiones parciales 31

3. Pruebas y resultados 32
3.1. Introducción 32
3.2. Pruebas realizadas sobre el problema de multiplicación matriz-vector. . . 32
3.2.1. Descripción del problema a resolver y del ambiente de pruebas. . 32
3.2.2. Tiempo de ejecución 33
3.2.3. *Speedup* 37
3.2.4. Eficiencia 38
3.2.5. Comunicaciones 39
3.2.6. Balance de Carga 41
3.3. Pruebas realizadas sobre un problema de acoplamiento molecular. 44
3.3.1. Descripción de las pruebas y los escenarios de prueba. 44
3.3.2. Tiempo de ejecución 46
3.3.3. *Speedup* 47

3.3.4. Eficiencia 49
3.3.5. Comunicaciones 50
3.3.6. Unidades generadas. 51
3.4. Conclusiones parciales 52

Conclusiones **54**

Recomendaciones **55**

Referencias **56**

Índice de figuras

2.1. Topología de la plataforma. 21
2.2. Diagramas UML de las clases necesarias para la implementación de una aplicación en la plataforma. 23
2.3. Paquetes del servidor de peticiones. Vista de relación 26
2.4. Diagrama de clases del paquete ***tarenal.server***. 27
2.5. Paquetes del Cliente. Vista de relación 28
2.6. Paquetes del Cliente. Vista de relación. 29
2.7. Paquetes del Cliente. Vista de relación. 30

3.1. Mediana de las comunicaciones generadas para cada una de las pruebas realizadas. 39
3.2. Balance de carga para cada una de las pruebas realizadas 42
3.3. Número de comunicaciones, cuando se utiliza las estrategias GSS, PSS, FSS y TSS para los 5 escenarios. 51
3.4. Unidades generadas necesarias para terminar la ejecución de la aplicación. . . . 52

Índice de tablas

3.1. Descripción de los nodos utilizados por ambientes. 33

3.2. Mediana del tiempo de ejecución(segundos) del problema para un vector de 8192 elementos. 33

3.3. Mediana del tiempo de ejecución(segundos) del problema para un vector de 16384 elementos. 34

3.4. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 1 nodo(9,679473$e-06$). 34

3.5. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 2 nodos(0,003936367). 35

3.6. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 4 nodos(0,001833963). 35

3.7. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 8 nodos(1,152619$e-05$). 35

3.8. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 1 nodo(0,004266478). 36

3.9. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 2 nodos(7,339108$e-07$). 36

3.10. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 4 nodos(2,796184$e-05$). 37

3.11. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 8 nodos($4{,}454626e-08$). 37

3.12. *Speedup* del problema para un vector de 8192 elementos. 37

3.13. *Speedup* del problema para un vector de 16384 elementos. 38

3.14. Eficiencia del problema para un vector de 8192 elementos. 38

3.15. Eficiencia del problema para un vector de 16384 elementos. 38

3.16. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 1 nodo($5{,}043477e-07$). . 40

3.17. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 2 nodos(0.0002352295). . 40

3.18. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 4 nodos($3{,}198628e-07$). 40

3.19. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 8 nodos($1{,}779841e-06$). 40

3.20. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 1 nodo(0,0009111189). . 41

3.21. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 2 nodos($2{,}640758e-06$). 41

3.22. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 4 nodos($1{,}499105e-07$). 41

3.23. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 8 nodos($6{,}727263e-08$). 41

3.24. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 2 nodo(0,0007530543). 43

3.25. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 4 nodos(0,0003140667). 43
3.26. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 8 nodos(0,005464635). 43
3.27. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 2 nodo(6,642106$e-07$). 44
3.28. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 4 nodos(4,938106$e-08$). 44
3.29. Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 8 nodos(9,126325$e-09$). 44
3.30. Descripción de los nodos utilizados por ambientes. 45
3.31. Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 1. 46
3.32. Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 2. 46
3.34. Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 4. 47
3.33. Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 3. 47
3.35. Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 5. 47
3.36. Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 1 48
3.37. Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 2. 48
3.38. Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 3. 48
3.39. Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 4. 48
3.40. Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 5. 48

3.41. Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 1. 49
3.42. Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 2. 49
3.43. Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 3. 49
3.44. Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 4. 50
3.45. Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 5. 50

Introducción

Introducción

La programación paralela es una técnica de las ciencias de la computación que hace uso de múltiples recursos computacionales de manera simultánea para ejecutar tareas costosas en cuanto al tiempo o al espacio. Su principal objetivo es reducir el tiempo de ejecución de estas tareas o mantener el mismo tiempo de ejecución al aumentar la cantidad de datos a procesar. Trabaja bajo la premisa de que, la mayoría de los problemas se pueden dividir en problemas más pequeños (secuencias de instrucciones), los cuales pueden ser resueltos de manera concurrente. Según la Taxonomía de Flynn, las aplicaciones paralelas pueden ser clasificadas atendiendo a los datos y las instrucciones [16]. Las computadoras paralelas pueden ser clasificadas según el *hardware* que las componen o la manera de organizar la memoria [16, 28, 44, 55]

Según la organización de la memoria, las computadoras paralelas se pueden clasificar como ordenadores de memoria compartida, de memoria distribuida u ordenadores híbridos. Los ordenadores de memoria compartida son aquellos donde todas las unidades de procesamiento acceden al mismo espacio de memoria y se pueden ver los cambios realizados por otras unidades de procesamiento. El acceso a la memoria puede ser tanto uniforme (*UMA-Uniform Memory Access*) como no uniforme (*NUMA-Non Uniform Memory access*). Los ordenadores de memoria distribuida son aquellos donde cada unidad de procesamiento tiene su respectivo espacio de memoria local, el cual no es visible para otra unidad de procesamiento. Los ordenadores híbridos son aquellos que tienen grupos de unidades de procesamiento compartiendo un espacio de memoria local y al mismo tiempo estos grupos se comunican entre sí mediante una red interna [16, 28, 44, 55].

Según el *hardware*, las computadoras paralelas pueden ser clasificadas como sistemas distribuidos o sistemas multinúcleos y/o multiprocesadores. Los sistemas multinúcleos son aquellos que tienen varias hilos de procesamiento en una misma unidad, mientras que los sistemas distribuidos se caracterizan por administrar varias unidades de procesamiento en distintos dispositivos conectados por una red local. Los ejemplos más significativos de las arquitecturas distribuidas son los clusters y los *grids* de computadoras. Un cluster no es más que un grupo de ordenadores conectados por una red local que en la mayoría de los casos tienen los mismos recursos computacionales (homogéneo), existiendo ejemplos

de clusters heterogéneos [16, 28, 44, 55]. La mayor cantidad de supercomputadoras en el **TOP500** son cluster de alto desempeño [57]. Un *grid* de computadoras proporciona acceso confiable, consistente y generalizado a recursos heterogéneos (almacenamiento, poder de cómputo) entre diferentes dominios administrativos y su objetivo es permitir el intercambio de estos recursos de una manera unificada, maximizando su uso. Un *grid* de computadoras es capaz de conectar cluster, *mainframes* y otros *grids* entre ellos [4, 39].

La programación paralela brinda la posibilidad de resolver problemas complejos en cuanto al tiempo en periodos más cortos, por esta razón se emplea de forma frecuente en casi todas las disciplinas científicas. Por ejemplo, en ciencias de la vida se utiliza en simulaciones de sistemas biológicos complejos [25], análisis filogenéticos [17], desarrollo de flujos de trabajo para el análisis de virus [41] o en el análisis de secuencias [48]. En el pronóstico del clima el análisis concurrente es de suma importancia [9, 31, 37, 46] y sus aplicaciones van desde el análisis de grandes lotes de información [37], hasta la evaluación y visualización de pronósticos [9, 46].

El rendimiento de la ejecución de una aplicación paralela es dependiente del *hardware* en el que se ejecute. Se espera que mientras más recursos computacionales puedan ser utilizados de manera concurrente, para la solución de una tarea, menor sea el tiempo que se demora en dar una respuesta. En la actualidad existe la competencia para llegar primero a la ejecución de 1 $EFlops/s$ [57], (alrededor de 10^{18} operaciones de punto flotante por segundos), pero esta competencia se realiza entre en un número reducido de países. China, Estados Unidos, Japón, Alemania, Francia y El Reino Unido poseen más de la mitad de las supercomputadoras listadas en el **TOP500** [57]. Esto implica que el acceso a estas grandes infraestructuras sea un proceso complejo y una motivación para crear proyectos alternativos capaces de ejecutar aplicaciones costosas en cuanto al tiempo.

Una solución alternativa es la creación de proyectos basados en computación voluntaria, paradigma computacional donde personas (también llamados voluntarios) donan recursos computacionales a proyectos que los utilizan para distribuir y ejecutar tareas, así como para el almacenamiento de información [1, 5, 13, 35, 47]. Estos proyectos tienen el inconveniente de ser volátiles, propensos a errores y heterogéneos dadas las características de los recursos que los conforman. También tienen la restricción de que las tareas que se ejecuten en ellos deben ser divididas en subtareas independientes entre sí, debido a que no existe comunicación entre los recursos donados [47].

No obstante, los proyectos de computación voluntaria son alternativas utilizadas en una amplia variedad de problemas. Existen proyectos destinados a la búsqueda de vida extraterrestre [1], al seguimiento y alerta temprana de sismos [51] o para probar algoritmos de encriptación [15]. Entre los ejemplos más significativos está *BOINC* (Berkeley Open

Infrastructure for Network Computing) [15], el cual brinda una plataforma de computación voluntaria de propósito general y además ofrece un *framework* sobre el cual se pueden desarrollar otras plataformas. En la actualidad *BOINC* permite la conexión de dispositivos celulares [5, 15, 47] para la ejecución de tareas y tiene asociado a él alrededor de 200000 voluntarios, conectados con casi un millón de dispositivos, alcanzando una capacidad de cómputo superior a 16 $PFlops/S$ [3].

Basado en estas ideas, y con el propósito de aprovechar los recursos computacionales disponibles en la Universidad de las Ciencias Informáticas, se desarrolló, por el grupo de Bioinformática, la plataforma de cálculo distribuido T-arenal. Esta plataforma sigue el paradigma de computación voluntaria y está pensada para desplegarse en un ambiente no dedicado y heterogéneo, utilizando computadoras conectadas por una red local y con una alta variedad de *hardware* [27]. T-arenal permite ejecutar las tareas de forma distribuida, dividiéndolas en pequeñas subtareas denominadas **unidades de trabajo** y distribuyéndolas entre las computadoras asociadas, aprovechando el poder de cálculo que ofrecen las estaciones de trabajo en conjunto. En la plataforma los usuarios pueden definir la granularidad con la que quieren distribuir las tareas, pero no es capaz de determinar de manera automática la cantidad de tareas a asignar a cada estación de trabajo en dependencia del ambiente en el que se ejecuta y las tareas que faltan por procesar. Esto implica que si el usuario no determina la granularidad en un problema determinado, la plataforma envía en cada instante una sola **unidad de trabajo**. Esta manera de asignación de tareas mantiene la plataforma balanceada pero maximiza las comunicaciones y realiza un mal manejo de los recursos computacionales asociados, influyendo de forma negativa en los tiempos de ejecución de los trabajos.

Por lo tanto, tomando en consideración lo planteado en párrafos anteriores se identifica como problema científico: ¿Cómo reducir el tiempo de ejecución y la cantidad de comunicaciones generadas por las tareas ejecutadas en la Plataforma de cálculo distribuido T-arenal teniendo en cuenta los recursos computacionales de los clientes asociados al sistema?. De esta manera la presente investigación tiene como **objeto de estudio** sistemas de computación voluntaria.

Para darle solución al problema científico se planteó el siguiente objetivo general de investigación, que consiste en: Desarrollar un planificador de tareas para T-arenal que reduzca el tiempo de ejecución y la cantidad de comunicaciones teniendo en cuenta el número de tareas sin asignar y los recursos computacionales de los ordenadores integrados a la plataforma. De esta forma se tiene como **campo de acción** los planificadores de tarea para sistemas distribuidos heterogéneos y no dedicados.

El objetivo general antes especificado se desglosa en los siguientes objetivos

específicos:

1. Especificar los algoritmos de planificación de tareas que pueden ser adaptados a T-arenal.

2. Especificar un método que estime la importancia de los clientes en T-arenal.

3. Implementar para T-arenal diversos algoritmos de planificación de tareas que tengan en cuenta la cantidad de tareas por planificar y la importancia de los clientes para T-arenal.

4. Validar que los planificadores de tarea propuestos para T-arenal reducen el tiempo de ejecución y el número de comunicaciones de las aplicaciones que se ejecutan.

Preguntas de Investigación:

1. ¿ Qué algoritmos de planificación de tareas pueden ser adaptados a T-arenal?

2. ¿ Cómo puede ser estimada la importancia de cada cliente en T-arenal?

3. ¿ Qué mecanismos de asignación de tareas serían viables en la plataforma, teniendo en cuenta su característica heterogénea y no dedicada ?

4. ¿ Qué influencia tienen los planificadores de tareas propuestos en el rendimiento de las aplicaciones ejecutadas en T-arenal?

Después de haber evaluado el marco teórico se formuló la siguiente hipótesis de investigación:

H: El uso de una estrategia *self-scheduling* para planificar tareas en los servidores de peticiones optimiza el tiempo de ejecución de las tareas que se ejecuten en la plataforma y las comunicaciones entre los servidores de peticiones y sus clientes.

Este trabajo posee como valor práctico el desarrollo de un nuevo planificador para la plataforma de cálculo distribuido T-arenal. El planificador hará un mejor uso de las estaciones de trabajo asociadas a la plataforma, optimizará las comunicaciones y reducirá el tiempo de ejecución de las aplicaciones en la plataforma.

Como valor metodológico este trabajo aporta la aplicación de métricas para medir la escalabilidad de las aplicaciones sobre los nuevos planificadores propuestos, tales como: tiempo de ejecución, *speedup* y eficiencia. Además, se utilizaron las métricas cantidad de comunicaciones y cantidad de unidades generadas para evaluar el proceso de generación de unidades en cada uno de los planificadores. Para medir como influye el planificador

en los clientes se tuvo en cuenta el balance de carga en cada ejecución de una tarea y se utilizaron pruebas de significación no paramétricas para determinar la existencia de diferencias significativas en las métricas utilizadas.

La tesis está estructurada en tres capítulos. En el Capítulo 1 se tratan de manera general, los fundamentos de la programación paralela y distribuida, así como los aspectos generales de la computación voluntaria y sus ejemplos más significativos. Además, se describen las estrategias *self-scheduling* para la planificación de tareas. En el Capítulo 2 se aborda la adaptación de las estrategias *self-scheduling* en la plataforma, así como el mecanismo para el cálculo de la importancia de los clientes con respecto al servidores de peticiones que los administra. En el Capítulo 3, se muestran los resultados de las diferentes pruebas realizadas a las estrategias *self-scheduling* estudiadas en el trabajo. Este documento culmina con las Conclusiones, Recomendaciones y Referencias Bibliográficas.

Capítulo 1. Computación Voluntaria y estrategias *self-scheduling*

Capítulo 1. Computación Voluntaria y estrategias *self-scheduling*

1.1 Introducción

En este capítulo se describe el paradigma de computación voluntaria, mencionando algunos de sus rasgos característicos y dónde se está aplicando actualmente. Se mencionan las características fundamentales de las plataformas que hacen uso de este paradigma, incluyendo la plataforma de cálculo distribuido que sirve de base este trabajo, de la cual se realiza una breve descripción. Para finalizar se describen las estrategias *self-scheduling*, algoritmos desarrollados inicialmente para planificar ciclos concurrentes en sistemas homogéneos, pero adaptados a sistemas distribuidos.

1.2 Computación voluntaria

Computación voluntaria(CV) es un paradigma computacional donde personas (también llamados voluntarios) donan recursos computacionales a proyectos, que los utilizan para distribuir y ejecutar tareas y/o para almacenamiento [1, 5, 13, 35]. La mayoría de los sistemas de CV tienen la misma estructura básica: un programa cliente que se ejecuta en las estaciones de los voluntarios y que periódicamente contactan con un servidor para solicitar tareas o para devolver alguna terminada [5]. En los sistemas voluntarios, las tareas son divididas en subtareas, denominadas unidades de trabajo, cada unidad de trabajo es enviada a una estación voluntaria, esta estación la ejecuta y devuelve el resultado si antes no ocurre algún error. Los problemas de tipo *Bag of Tasks* son los que se ejecutan en este tipo de sistemas, siendo característico de estos problemas su descomposición en tareas independientes (subtareas o unidades de trabajo) que se pueden ejecutar fuera del orden de presentación [42]. Un proyecto de computación voluntaria tiene la característica de ejecutar varias tareas simultáneamente [59].

Existen también otras formas de metacomputación, tales como *grid* de computadoras, que buscan facilitar el uso de servidores a través de Internet. Los *grids* de computadoras tienen la desventaja de ser complejos de configurar y de acceso privado. La mayoría de

estos proyectos están desarrollados siguiendo requerimientos complejos de configuración, lo que implica que posibles colaboradores no se sientan interesados en participar [35].

1.2.1 Ejemplos de proyectos de computación voluntaria

Ejemplos de aplicaciones que utilizan proyectos voluntarios son disímiles. Van desde búsqueda de vida extraterrestre [1] o nuevos números primos [63], orientados a la bioinformática [3], romper códigos cifrados [19, 47] o control y seguimiento de sismos [20, 38, 51]. El paradigma de Computación voluntaria provee mayor rendimiento que sistemas tradicionales como *clusters* con un menor costo en términos de instalación, mantenimiento, energía e infraestructura. No obstante, estos ambientes son particularmente cambiantes dado la naturaleza de los recursos: propensos a errores, heterogéneos y no dedicados [29]. Se tiene como punto de partida la segunda mitad de los noventas, con el nacimiento del proyecto "The Great Internet Mersenne Prime Search" [15, 18, 47, 59].

1.2.1.1 GIMPS

Great Internet Mersenne Prime Search (GIMPS) nació en 1996 y su objetivo es buscar números primos de la forma $2^p - 1$, denominados primos *Mersennes* . El proyecto ha encontrado los últimos 15 primos *Mersenne* conocidos hasta enero del 2017, 13 de los cuales fueron en su momento el número primo más grande conocido. El último primo *Mersenne* conocido es $2^{74207281} - 1$ o $M_{74207281}$ para reducir. Este proyecto cuanta con la colaboración de más de 178000 colaboradores, con alrededor de 1500000 CPUs y una capacidad de computo superior a los 142 *TFLOPS/S* [63].

1.2.1.2 Distributed.net

Distributed.net fue el segundo proyecto, comenzando en el año 1997 y fue el primero de propósito general. Esta plataforma ha utilizado las ventajas de la computación voluntaria para ganar varios concursos de criptografía tales como *RSA secret-key*, organizado por los laboratorios RSA [15, 18, 47, 59]. Distributed.net ha sido empleado para resolver problemas como: *Optimal 26-Mark Golomb Rulers, DES-III, DES II-2, DES II-1, RC5-56* y *RC5-64* [19, 47].

1.2.1.3 Quake-Catcher Network (QCN)

El proyecto QCN conecta acelerómetros de sistemas microelectromecánicos de bajo costo a una red de voluntarios conectados a través de Internet [51]. Sensores MEMS

son conectados a las estaciones voluntarias por puerto USB para recolectar y enviar información de movimientos de tierra a un servidor central [38]. Esta arquitectura es paradigma de bajo costo comparado con las redes de sensores tradicionales y su objetivo es la detección y caracterización rápida de temblores de tierra. BOINC, ver subsección 1.2.1.4, es utilizado para iniciar y administrar las comunicaciones entre los voluntarios y el servidor [20, 38].

1.2.1.4 Berkeley Open Infrastructure for Network Computing (BOINC)

BOINC (Berkeley Open Infrastructure for Network Computing) es una plataforma *open-source* y *framework* para computación voluntaria y *grid* de computadoras y es ampliamente la plataforma que más se utiliza [5, 15, 59]. La plataforma provee una interfaz donde los voluntarios pueden escoger sobre que proyecto trabajar y la cantidad de recursos que donarán [15]. Actualmente, al menos 200000 voluntarios están registrados en la plataforma conectando más de un millón de estaciones, y una capacidad de cómputo superior a 16 PFlops/S. Esta capacidad de cálculo pone a BOINC en el lugar 6 del **top500** [3]. Actualmente BOINC tiene versiones de clientes para dispositivos móviles y dispositivos Kindle. Para estos dispositivos la plataforma tiene la restricción de que solo ejecutará alguna tarea cuando estén conectados a la corriente y su batería esté por encima del 90 % de carga. Otra restricción con respecto a los dispositivos móviles es con respecto a la transferencia de datos, ya que se necesita que estén conectados a una red Wi-Fi. [5].

1.2.1.5 XtremWeb

Xtremweb fue desarrollado por la Universidad de París y provee herramientas para facilitar la creación de proyectos de computación voluntaria, enviando a cada cliente una unidad de trabajo por cada petición. Cuando un cliente completa la tarea asignada, envía el resultado al servidor y realiza la petición de una nueva tarea [59]. XtremWeb tiene 3 módulos, el cliente, el coordinador y los *workers*. Mediante el cliente un usuario puede subir una tarea vía Internet. El coordinador recibe la tarea y la distribuye entre los *workers*. Los *workers* instalados en los dispositivos voluntarios son los encargados de ejecutar la tarea [18].

1.2.1.6 HTCondor

HTCondor fue desarrollado en la Universidad de Wisconsin-Madison. Las estaciones de trabajo son ubicadas en un lista cuando están libres y son eliminadas cuando están ocupadas [18]. Para esto HTCondor hace uso de un paradigma de gestión de recursos

basado en *matchmarking*, donde cada estación de trabajo disponible publica los recursos que se pueden utilizar, las restricciones para su uso y el tiempo que estará disponible. Cada vez que llega una nueva tarea, trae los requerimientos de recursos necesarios, el proceso *matchmaker* toma estos requerimientos y busca la estación de trabajo disponible que cumpla con los requerimientos de la tarea [67]. Este mecanismo disminuye el riesgo a fallos presente en los sistemas heterogéneos y no dedicados por la volatilidad y heterogeneidad de los recursos [34]. HTCondor provee gestores de colas, políticas de planificación. También ofrece mecanismos para administrar y monitorear trabajos y recursos [12, 24].

En HTCondor los propietarios de los recursos tienen prioridad a la hora de ejecutar sus trabajos. Adicionalmente, usuarios que normalmente utilizan un porciento pequeño de los recursos en sus ejecuciones tienen prioridad con respecto los que normalmente necesitan mayor cantidad de recursos [6]. Los trabajos de usuarios con mayor prioridad pueden adelantarse a los trabajos en ejecución y HTCondor garantiza que los trabajos que se eliminan de un recurso se inicien de nuevo en otro recurso hasta que se completen. HTCondor está diseñado especialmente para ejecutar tareas sobre un ambiente de máxima disponibilidad, donde alta disponibilidad hace referencia al uso de un número considerable de recursos computacionales sobre un periodo largo de tiempo [6, 12].

1.2.1.7 T-arenal

T-arenal es una plataforma de cálculo distribuido desarrollada en la Universidad de las Ciencias Informáticas, La Habana, Cuba. Se crea para ejecutar tareas costosas computacionalmente sobre los recursos computacionales desplegados en los laboratorios docentes conectados a través de la red local [27]. Cada estación de trabajo asociada a la plataforma ejecutará una aplicación cliente que periódicamente solicita ejecutar subtareas a un servidor de peticiones. El servidor de peticiones es el encargado de dividir las tareas, enviar subtareas a los clientes y recibir los resultados enviadas por los clientes. Los usuarios pueden enviar tareas conectándose a un servidor central, el cual tiene una lista de servidores de peticiones ordenados descendentemente, teniendo en cuenta la carga de trabajo y el rendimiento de los clientes asociados a él. El servidor central asignará la tarea al primer servidor de peticiones en la lista [27]. Como este sistema se abordará en la presente tesis, en el Capítulo 2 se dará una mayor descripción del mismo.

1.2.2 Arquitectura de sistemas de computación voluntaria

La mayoría de los proyectos de computación voluntaria utilizan una arquitectura cliente-servidor para la distribución de las aplicaciones que serán ejecutadas y los datos que serán procesados [19, 45, 63]. Recientemente se han realizado propuestas de esquemas multiservidores enfocados a resolver problemas de cuello de botellas que pueden aparecer si existen muchas conexiones concurrentes al servidor [10, 26, 30]. Para la mayoría de los proyectos (e.g.,GIMPS [63], distributed.net [19] y BOINC [45]), los voluntarios pueden descargar la aplicación cliente de los sitios oficiales. Estas aplicaciones se conectan a los servidores, ejecutan las tareas que se les envían, devuelven los resultados de la ejecución y están diseñadas para ejecutarse con una mínima prioridad o estar inactivas mientras el dispositivo esté en uso. Esto se puede resolver chequeando constantemente los eventos del teclado, del mouse o esperando que se active el protector de pantalla [5, 15, 47, 56].

La primera responsabilidad del servidor es gestionar las bases de datos, donde se almacena toda la información de usuarios, voluntarios y el estado de las tareas. Otra responsabilidad es dividir las tareas en subtareas(unidades de trabajo) y enviárselas a los clientes o almacenarlas de forma tal que los clientes tengan acceso a ellas. Una vez que una unidad de trabajo es terminada por un cliente es enviada al servidor. El servidor se encarga entonces de recibir y combinar los resultados de los clientes para construir la solución final. Por otro lado el servidor debe manejar la seguridad de los datos y la comunicación, así como proveer el mecanismo donde los usuarios puedan subir sus trabajos [59].

La mayoría de los proyectos tienen una aplicación cliente que se encarga de manejar tanto las comunicaciones como la parte de cálculo. Esto implica una implementación genérica para manejar las comunicaciones y el riesgo de tener cuellos de botellas y *buffers* sobrecargados. Otra desventaja es la necesidad de tener clientes para cada una de las arquitecturas y sistemas operativos existentes [56].

1.2.3 Aplicaciones

Los proyectos de computación voluntaria tienen la restricción de que los clientes no se pueden comunicar entre sí. Por lo tanto, las aplicaciones que se ejecutan en estos ambientes tienen como una característica que las subtareas que se generen son independientes entre ellas. Este tipo de aplicaciones son conocidas normalmente como aplicaciones *Bag of Tasks*. Ejemplos de este tipo de aplicaciones son: el análisis de base de datos independientes, búsquedas dentro de un espacio de parámetros que se pueden dividir en varias regiones independientes o simulaciones múltiples utilizando diferentes puntos de partida o restricciones. Esta topología no excluye necesariamente cálculos de

etapas múltiples con cálculos que dependen de uno o más cálculos anteriores. Todas las dependencias para una unidad de trabajo pueden ser ejecutadas y devueltas al servidor antes de comenzar a ejecutar la tarea en particular. Las unidades de trabajo deben tener la propiedad de que el tiempo de comunicación (enviar y recibir) junto al tiempo que se demora en generarlas (*overhead*) sea significativamente menor que el tiempo que se demora en ejecutarlas.

1.2.4 Seguridad

Los voluntarios en las plataformas voluntarias son anónimos y no se tiene conexión a las identidades reales de ellos. Para registrarse solo basta un correo electrónico o algún otro mecanismo de autenticación. Debido a su anonimato, los voluntarios no son responsables de los proyectos. En este sentido, los sistemas de CV utilizan la confianza basada en políticas para todos los proyectos [47]. Los voluntarios deberían estar atentos de que el ejecutable descargado pueda incluir un virus o que el programa en el servidor haya sido reemplazado por un *hacker* o que un proyecto pueda ser un engaño.

Para resolver que el cliente pueda tener incluido un virus o haya sido remplazado, la mayoría de los proyectos utilizan validaciones criptográficas de ejecutables, ya sea mediante un hash MD5, SHA o mediante cifrado de clave pública. De igual manera, los voluntarios deben elegir solo proyectos ejecutados por organizaciones confiables. Otras amenazas de seguridad para los proyectos que se ejecutan pudieran ser: falsificación de resultados, falsificación de crédito o desbordamiento del servidor de datos [56, 59].

1.2.5 Políticas de planificación de tareas

El crecimiento acelerado de proyectos sobre computación voluntaria junto al lento aumento de los voluntarios obliga a un mejor manejo de los recursos compartidos. La etapa de planificación implica dedicar un tiempo extra(*overhead*) para estimar la cantidad de tareas a enviar en cada momento, tiempo extra que afecta al tiempo total de ejecución. La estrategia que se adopte depende en gran medida de la información que se tenga acerca de la tarea y de los recursos disponibles: tiempo de ejecución estimado, costo de las comunicaciones de la aplicación, velocidad CPU, cantidad de memoria, entre otros.

La mayoría de las políticas de planificación en proyectos de computación voluntaria están basadas en heurísticas y están divididos en planificadores *naive* y planificadores basados en el conocimiento [47]. Las estrategias *naive* realizan la planificación sin tener en cuenta información del sistema o de la tareas, mientras que las políticas basadas en el conocimiento tienen en cuenta información del sistema (calidad de los clientes,

comunicación, entre otros) e información acerca de las tareas [47]. Ejemplos de políticas *naive* pueden ser primero en llegar primero en ser servido(FCFS) o asignación aleatoria, mientras como políticas basadas en el conocimiento se tiene *round robin* pesado o políticas basados en umbrales [47, 59].

Existen otros enfoques donde las aplicaciones se pueden considerar como secuencias múltiples de tareas y se modelan utilizando Grafos dirigidos cíclicos. En estos casos, los nodos representan subtareas y las aristas denotan dependencias, donde los resultados del nodo del cual parte la arista, son necesarios para la ejecución de la tarea que está donde termina la arista. [2, 65]. El problema aquí es encontrar una asignación de tareas sobre un conjunto de dispositivos disponibles que minimice el tiempo de ejecución y las comunicaciones necesarias para terminar la tarea. Las soluciones para estos problemas en su mayoría están basadas en técnicas de inteligencia artificial [21, 65]. En [10, 11, 26, 30] se propone la familia de algoritmos llamados *self-scheduling*, esquemas que definen una función decreciente para la cantidad de subtareas a enviar en cada paso con el objetivo de encontrar un equilibrio entre comunicaciones y balance de carga.

BOINC y Distributed.net utilizan la política denominada buffer múltiple. Con esta política los clientes pueden descargar y ejecutar concurrentemente múltiples subtareas. La cantidad de tareas a descargar se define estáticamente teniendo la ventaja de no incrementar el tiempo de ejecución con una etapa de planificación, pero tiene la desventaja de no ser adaptativa. XtremeWeb utiliza como política de planificación Buffer vacío, donde un cliente puede descargar solo una sub-tarea, ejecutarla, enviar el resultado y descargar otra sub-tarea si fuera necesario. Esta estrategia tiene la ventaja de tener siempre balanceado el sistema en detrimento de las comunicaciones [59].

1.3 Planificación de tareas con algoritmos Self-scheduling

El balance de la carga en sistemas distribuidos es clave para un consumo eficiente de los recursos computacionales con los que cuenta el sistema. Esta importancia se incrementa cuando el sistema es heterogéneo. Para hacer un uso eficiente de los recursos computacionales en un sistema distribuido, es necesario asignar correctamente la carga de trabajo entre los nodos existentes en el sistema.

Existen diferentes estrategias de planificación, las cuales se pueden clasificar en dos grupos: algoritmos de planificación estáticos y los algoritmos de planificación dinámicos. Los primeros tienen la ventaja de definir la carga de trabajo para cada nodo en tiempo de compilación, lo que evita un tiempo de ejecución extra (*overhead*). La desventaja es que al definir la carga de trabajo en tiempo de compilación estos algoritmos no se adaptan

en tiempo de ejecución a las variaciones que puede tener el sistema. Los algoritmos dinámicos planifican la carga de trabajo en tiempo de ejecución, lo que les permite adaptarse fácilmente a los cambios que se pueden presentar en el sistema.

En el grupo de algoritmos dinámicos se encuentra la familia de algoritmos *self-scheduling* o auto-planificación. Estos algoritmos fueron creados para resolver los problemas de planificación de ciclos paralelos en sistemas homogéneos [26] y son aplicados en bucles donde las iteraciones sean independientes entre ellas, de manera tal que se puedan planificar de manera independiente. Estos algoritmos fueron creados para sistemas homogéneos y de memoria compartida, pero han sido adaptados de manera eficiente a ambientes heterogéneos y a sistemas distribuidos [11, 22, 30, 32, 66].

Cuando se debe planificar un grupo considerable de tareas se nos presenta dos problemas: el balance de la carga y el costo de las comunicaciones. Estos factores son inversamente proporcionales. Si se atiende solamente el balance de la carga esto influye a un mayor costo en las comunicaciones. Por el contrario, si se atiende solamente a las comunicaciones esto implica un desequilibro de la carga, sobre todo si las tareas son de distintos tamaños. En este sentido, Kruskal y Weiss presentan dos modelos que atienden por separados a estos dos factores [36]. Por un lado presentan un modelo estático que asigna una tarea a cada procesador libre, obteniéndose un buen balance de la carga, pero el costo en las comunicaciones es el máximo posible. Por otro lado, proponen un modelo que divide el número de tareas entre la cantidad de procesadores, enviando a cada procesador la misma cantidad de tareas. Este algoritmo consigue un bajo costo en las comunicaciones, pero provoca un desequilibro en el balance de la carga, sobre todo si las tareas que se desean ejecutar tienen distintos tiempos de ejecución. Kruskal y Weiss demostraron en [36] que una buena estrategia es asignar un número de tareas decreciente a cada procesador.

1.3.1 Algoritmos self-scheduling

Además del trabajo de Kruskal y Weiss [36], se han desarrollado un conjunto de algoritmos derivados de las ideas originales de éstos. Los algoritmos *self-scheduling* son un conjunto de esquemas dinámicos y centralizados para planificar bucles. Todos se han desarrollado teniendo en cuenta siempre el objetivo de tener un equilibro entre el costo de las comunicaciones y el balance de la carga. Existen algoritmos para ambientes homogéneos [33, 50, 62] y heterogéneos [11, 22, 32, 64, 66], esquemas que tienen en cuenta la importancia de cada uno de los nodos [11, 32, 64, 66], o el comportamiento de estos en el sistema [11, 66].

1.3.2 *Guided Self-scheduling(GSS)*

El algoritmo *Guided Self-scheduling(GSS)* fue propuesto por Polychronopoulos y Kuck [50]. Ellos proponen en cada instante dividir la cantidad de tareas sin procesar (R) entre la cantidad de procesadores existentes en el sistema (P), considerando que en cada instante se planificarán $c_i = \frac{R}{P}$. Como la cantidad de tareas asignadas en cada instante es siempre decreciente se consigue un buen equilibro de la carga y se reduce los costos de comunicación. En los casos que la proporción entre la cantidad de tareas y la cantidad de procesadores es muy grande suele ocurrir que en las primeras asignaciones se planifiquen la mayoría de las tareas, influyendo en que la cantidad de tareas planificadas luego no sean suficientes para tener un balance de la carga.

1.3.3 *Factoring Self-scheduling(FSS)*

Factoring self-scheduling fue propuesto por Hummel en 1992 [33]. Igual que *Guided*, *Fatoring* asigna un número decreciente de tareas a cada procesador. La diferencia es que *Factoring* realiza la planificación por etapas. En cada etapa se divide a partes iguales una porción α de las tareas sin planificar R entre los procesadores existentes P. Esto implica que *Factoring* tenga mejor balance de la carga que *Guided*, sobre todo, en los casos donde los tamaños de las tareas son variables y no se pueden estimar. Definir la porción α, puede ser una tarea difícil, en [33] proponen que en cada etapa se planifique solamente la mitad de las tareas sin planificar. En sus experimentos muestran que alcanzan un buen tiempo de ejecución para una variedad de aplicaciones en sistemas homogéneos. En cada momento se planifica $c_i = \frac{R}{\alpha * P}$.

1.3.4 *Trapezoid Self-scheduling(TSS)*

Trapezoid Self-scheduling fue propuesto en 1993 [62]. Este algoritmo propone un función lineal decreciente para asignar el número de tareas a cada procesador. Con esta función se intenta minimizar los costos de sincronización y comunicación. En *Trapezoid* definen cual será la cantidad de tareas a planificar en el inicio F y el final L de la ejecución. Luego definen un factor decreciente D, y definen entonces que en cada momento i se planificaran $c_i = c_{i-1} - D$. Donde $\lfloor D = \frac{(F-L)}{N-1} \rfloor$ y $N = \lceil \frac{2*I}{(F+L)} \rceil$. Como valores inicial F y final L proponen tomar $F = \frac{I}{2*P}$ y $L = 1$. Sobre dichos valores comentan que son propuestas conservativas, pero que con ellas alcanzan buenos tiempos de ejecución, dejando abierta la discusión sobre cuáles serían los valores con los se obtendrían mejores tiempos [62]. Sobre esto comentan que es muy dependiente de la tarea a ejecutar y qué tan uniforme sean los datos de entrada [62], alegando también que escogerlos añade más

flexibilidad al algoritmo. La función lineal decreciente induce a un buen balance de carga incluso en un ambiente donde los datos de entrada no sean uniformes [62]. Otra fortaleza de *Trapezoid* es que reduce el tiempo que necesita el planificador para decidir el tamaño del próximo *chunck* (paquete de unidades de trabajo). D y N se calculan una sola vez, al inicio de la ejecución [62].

1.3.5 *Weighted Factoring self-scheduling*

Susan Hummel presenta en [32] una comparación entre una estrategia *Work Stealing* introducida en [54] y una estrategia *self-scheduling*. El estudio se realiza extendiendo el *Factoring self-scheduling* [33] a un ambiente heterogéneo. Con este fin, se estima el rendimiento de cada nodo en el sistema utilizando un algoritmo *bechmarking*. Los autores llegan a la conclusión de que las estrategias que decrementan el tamaño de los *chunks* son aconsejables para ambientes homogéneos. Plantean la pregunta de que sí se extiende a un ambiente heterogéneo cuán escalable sería. De igual manera platean la pregunta de cuál sería el número máximo de estaciones de trabajo que se podrían conectar sin provocar un cuello de botella a la hora de planificar el próximo *chunck*. Sobre este aspecto realizaron pruebas hasta un conjunto de 64 procesadores, teniendo como resultado que el cuello de botella en el planificador no es un problema. Proponen la idea de adaptar este esquema en un ambiente donde los procesadores puedan estar o no activos.

1.3.6 *Quadratic, Exponential and Root Self-scheduling(QSS),(ESS),(RSS)*

En [22] presentan tres nuevas propuestas de algoritmos *self-scheduling* para ambientes heterogéneos. La primera es el resultado de desarrollar un serie de Taylor hasta el término cuadrático, teniendo como resultado una función cuadrática que determina la cantidad de tareas a enviar a un nodo $C(t) = a + bt + ct^2$. Donde $a = C_0, b = (4C_{\frac{n}{2}} - C_N - 3C_0)/N, c = (2C_0 + 2C_N - 4C_{\frac{N}{2}})/N^2$, $N = \frac{6I}{4C_{\frac{N}{2}} + C_N + C_0}$, C_0 y C_N son determinados al inicio de la ejecución $C_{N/2} = \frac{C_0 + C_N}{\gamma}$. Si γ se escoge igual a dos se tendría una función linea semejante a TSS [62] y en dependencia del valor que tome γ, Ct puede estar por debajo o por encima de la curva de la función lineal. Sobre cuáles son los valores apropiados para γ y C_N se realizaron diferentes pruebas, ejecutando versiones de aplicaciones para multiplicar matrices y ordenamiento (*HeapSort*) con distintos números de tareas y cantidad de procesadores. Los resultado de estas pruebas mostraron que los mejores valores para γ están en el rango de $[3,7]$ y para C_N están en el rango de $[1,8]$, teniendo como valor predominante para $\gamma = 6$. Los autores dejan abierta la discusión sobre cuáles sería los valores apropiados e indican que dependen del número de tareas, la cantidad de

procesadores y la complejidad de las tareas que se desean ejecutar.

Para *Exponential* y *Root* parten de la hipótesis de que $C(t)$ es una función decreciente en base del tiempo. Partiendo de la ecuación diferencial $\frac{dC(t)}{dt} = f(t)$. Para *Exponential* plantean que la pendiente(negativa) es proporcional al tamaño del *chunk*. Por esto definen como *Exponential self-scheduling* la función $\frac{dC(t)}{dt} = -kf(t)$. Como tamaño inicial escogen $C_0 = \frac{I}{2*P}$ como se propone en *Trapezoid* [62]. Definen la cantidad de *chunks* que serán enviados en toda la ejecución como $N = -ln[1-2Pk]/k$, planteando que será independiente de la cantidad de tareas a ejecutar. Al igual que para QSS realizaron pruebas sobre diferentes cantidades de tareas y procesadores, tomando valores para $k = [0{,}010 - 0{,}024 : 0{,}001]$ y obteniendo mejores tiempos cuando $k = 0{,}019$.

Para *Root* plantean que la pendiente(negativa) es inversamente proporcional al tamaño del *chunk*. Por esto definen como *Root self-scheduling* la función $\frac{dC(t)}{dt} = -k/f(t)$. Ejecutando los mismos experimentos obtuvieron que para multiplicar matrices el valor mínimo siempre se encontró para $k = 35$. A diferencia de *HeapSort*, en este no hubo coincidencia en los mínimos alcanzados. Como conclusión de este trabajo, comparando las propuestas con las versiones clásicas de *self-scheduling(CSS,GSS,TSS,FSS)*, en un ambiente heterogéneo, el de mejor comportamiento alcanzado fue QSS, siendo segundo ESS y los de peor resultados CSS y RSS en el grupo de pruebas sobre multiplicación de matrices. En el caso de *HeapSort*, los de peor rendimiento fueron CSS, GSS. Concluyen afirmando que un número pequeño de *chunks* es asociado a un mal rendimiento, afirmando que lo contrario no se ha podido demostrar. Una cantidad pequeña de *chuncks* implica un desbalance de la carga y que un número grande de *chunks* implica mayor comunicación. Plantean que en sus pruebas pudieron confirmar que el desbalance de la carga es el factor determinante en el rendimiento de la ejecución sobre un ambiente heterogéneo. Afirman de igual forma que los planificadores que tienen un comportamiento cóncavo hacia abajo(*concave down*) son ineficientes, porque producen un número pequeño de *chunks* en cada iteración ,implicando un desbalance.

1.3.7 *Heterogeneous Dynamic Self-scheduling(HDSS)*

En [11] definen un esquema de dos fases. Una primera fase de aprendizaje donde se planifica una porción de las tareas con el objetivo de tener el comportamiento de cada nodo para la ejecución en curso. Esta fase es responsable de calcular el peso para cada procesador. En el inicio el planificador envía a cada nodo(que se conecta por primera vez) la cantidad mínima de tareas que se pueden enviar(esta cantidad mínima es definida por el usuario). Luego de esto, la cantidad de tareas que se envía a cada nodo es incremental en un factor determinado por el usuario. Esta fase termina cuando el comportamiento de todos

los nodos es estable. El comportamiento de un nodo es estable cuando la diferencia entre los pesos de la planificación actual con respecto a la anterior es menor que un umbral determinado. El peso para cada ejecución se termina dividiendo la cantidad de tareas enviadas entre el tiempo total de procesamiento de estas tareas. Este tiempo total es la suma de los tiempos de ejecución y de comunicación. De esta manera se tiene para cada nodo la cantidad de tareas que puede ejecutar en cada instante de tiempo.

Una vez completada la primera fase, HDSS usa una adaptación de GSS [50]. GSS divide la cantidad de tareas sin planificar entre el total de procesadores $C_I = R/P$. HDSS incluye una modificación a la función y plantea que $P = W = \sum_{j=1}^{p} w_j$. Donde $w_j = \frac{\#\_de\_tareas\_ejecutadas}{Tiempo\_Total}$. De esta forma en la segunda fase la función que determina la cantidad de tareas a enviar a cada nodo es $C_t = Rw_i/W$.

Las pruebas realizadas en este trabajo, sobre un ambiente altamente heterogéneo y comparadas con los algoritmos que según sus criterios tienen los mejores resultados en el balance de la carga para ambientes heterogéneos [7, 40, 61], concluyeron que HDSS alcanza un 219 % de rendimiento sobre estos.

1.3.8 *Self-scheduling* para sistemas distribuidos

Han introduce en [30] adaptaciones de los algoritmos clásicos GSS, FSS y TSS para un ambiente distribuido montado en la nube. Proponen tener en cuenta la velocidad de procesamiento de cada estación de trabajo para un mejor rendimiento del sistema. Acotan que es efectivo en la práctica medir y tener en cuenta la velocidad de procesamiento para la etapa de planificación, aunque es una clasificación no precisa y dependiente de factores disímiles tales como memoria principal o cache.

Para estimar la importancia de cada nodo del sistema en [30] proponen evaluar cada nodo con respecto al nodo con menor velocidad de cálculo. De esta forma cada nodo tendrá una capacidad virtual definida como $v_j = Speed(P_j)/min_{1\leq i\leq p}\{Speed(P_i)\}$, donde $Speed(P_j)$ es la velocidad de cálculo del nodo j.

Teniendo en cuenta los siguientes términos:

- I es el número total de subtareas a planificar.
- p es el número de nodos en el sistema que van a colaborar en la ejecución de la tarea.
- P_i representa el i-ésimo nodo.
- $R_0 = I$.

- $C_i = f(R_{i-1}, p)$ f es la función que determina la cantidad de subtareas a enviar en el instante i, C_i es el número de subtareas a enviar.
- $R_i = R_{i-1} - C$.
- $V = \sum_{j=1}^{p} v_j$.

Distributed Guided Self-Sheduling(DGSS)

Para DGSS, Han propone determinar el tamaño de cada *chunk* en dependencia de la capacidad virtual del nodo que realiza la petición y su importancia para el sistema. Cada vez que un nodo realiza la petición se le enviará una cantidad de subtareas definidas siguiendo el algoritmo 1.

Algoritmo 1: Algoritmo para determinar la cantidad de tareas a enviar en DGSS.

Entrada: v_i capacidad virtual del nodo i, nodo que realiza la petición
Salida: C, número de tareas a enviar

```
1 C = ⌈R/V⌉ * v_i
2 R = R − C
3 return C
```

Distributed Factoring Self-Sheduling(DGSS)

Para la versión distribuida de FSS en [30] proponen definir una cantidad máxima de subtareas para una etapa de la planificación y al igual que DGSS tiene en cuenta la capacidad virtual del nodo que realiza la petición y su importancia para el sistema. Como factor α para determinar la cantidad de subtareas a tener en cuenta en cada etapa siguen lo propuesto por [33], quienes definen como un buen valor $\alpha = 2$. Cada vez que llega una petición el servidor entonces manda una cantidad de subtareas calculada por el algoritmo 2.

Distributed Trapezoid Self-Sheduling(DTSS)

Para DTSS se toman los valores iniciales propuestos por [62] adaptándolos al ambiente distribuido:

$$F = \lfloor \frac{I}{2V} \rfloor, L = 1, F = \lceil \frac{2*I}{F+L} \rceil$$
$$D = \lfloor \frac{F-L}{N-1} \rceil, current = F$$

En el algoritmo 3 se muestra como se calcula la cantidad de subtareas que se enviarán a un nodo determinado. Cuando un nodo realiza una petición, se le enviará una cantidad de tareas igual a la unión de v *chunks*, donde v estará definido por cuán mejor es el nodo que está realizando la petición con respecto al peor nodo.

Algoritmo 2: Algoritmo para determinar la cantidad de tareas a enviar en DFSS.

Entrada: v_i capacidad virtual del nodo i, nodo que realiza la petición
Entrada: α factor que determina la cantidad de subtareas a tener en cuenta en cada momento
Salida: C, número de tareas a enviar

1 $C = \lceil R/(\alpha * V) \rceil * v_i$
2 $DC_{sum} = DC_{sum} - C$
3 **if** (se han asignado todas las subtareas en la etapa actual) **then**
4 $R = R - DC_{sum}$
5 $DC_{sum} = 0$
6 **endif**
7 *return* C

Algoritmo 3: Algoritmo para determinar la cantidad de tareas a enviar en DTSS

Entrada: v_i capacidad virtual del nodo i, nodo que realiza la petición
Salida: C, número de tareas a enviar

1 $c = v_i * current - \frac{v_i*(v_i-1)}{2} * D$
2 $current = current - v_i * D$
3 *return* c

1.4 Conclusiones parciales

En el presente capítulo fueron expuestas las principales características de los proyectos de computación voluntaria así como los ejemplos más significativos. De igual forma se describieron los algoritmos de planificación pertenecientes a la familia de algoritmos *self-scheduling*, sus principales características y las condiciones donde alcanzan el mejor rendimiento. Como conclusiones de este capítulo podemos decir 1) que los proyectos de computación voluntaria son excelentes alternativas para la computación de alto desempeño o rendimiento; pese a las desventajas que pueden presentar dada la naturaleza volátil de los recursos, el ambiente heterogéneo y la restricción en cuanto a los trabajos que se pueden ejecutar, 2) que los algoritmos *self-scheduling* son estrategias ampliamente utilizadas para la planificación de tareas, creados para la planificación de iteraciones de ciclos paralelos en sistemas homogéneos, pero son perfectamente ajustables a ambientes heterogéneos cuando las tareas que se ejecuten puedan ser divididas en subtareas independientes entre sí.

Capítulo 2. Estrategias self-scheduling para la plataforma de cálculo distribuido T-arenal

Capítulo 2. Estrategias self-scheduling para la plataforma de cálculo distribuido T-arenal

2.1 Introducción

En el presente capítulo se describe la plataforma de cálculo distribuido T-arenal, así como las modificaciones realizadas sobre ésta que permiten la adaptación de las estrategias *self-scheduling*. Específicamente, en la sección 2.2 se mencionan las características fundamentales de la plataforma. En la sección 2.3 se describe cómo se evalúa la calidad de los clientes en el sistema, teniendo en cuenta la cantidad de operaciones punto flotante por segundo($flops/s$) que pueden realizar. En esta sección se introducen las modificaciones realizadas sobre las estrategias *Guided Self-Scheduling*, *Factoring Self-Scheduling* y *Trapezoid Self-Scheduling* para su adaptación a la plataforma. Finalmente en la sección 2.4 se muestran los principales cambios en el diseño de la plataforma que permiten implementar las estrategias de planificación *self-scheduling*.

2.2 Plataforma de cálculo distribuido T-arenal

La plataforma de cálculo distribuido T-arenal está diseñada para ejecutar tareas sobre un ambiente no dedicado y heterogéneo. Fue desarrollada en la UCI y su objetivo es sacar provecho de los recursos computacionales existentes en los laboratorios docentes de la institución, conectados por una red local. Para esto se utilizó un enfoque multi-servidor que combina las bondades de los esquemas peer-to-peer y cliente-servidor [26, 27].

La plataforma está dividida en dos partes: front-end y back-end. El front-end es la interfaz mediante la cual los usuarios podrán acceder a las funcionalidades del sistema, mientras que el back-end es el responsable de ejecutar todas las peticiones realizadas vía front-end. El back-end administra la cola de trabajos, los usuarios, los servidores de peticiones existentes en la red y los rangos IP que estos tienen asociados. Para la comunicación entre los módulos de la plataforma se utiliza Java RMI [49] así como Java Sockets y Apache FTP [8] para compartir ficheros.

El back-end del sistema está basado en un modelo multi-servidor organizado como un

árbol de tres niveles, ver **figura 2.1**. Está separado en 3 módulos: el servidor central(nodo raíz), los servidores de peticiones(nodos internos) y los clientes(nodos hojas). El servidor central es la interfaz del sistema entre el back-end y el front-end y tiene dos tareas fundamentales: la administración de la plataforma y la planificación de las tareas. En la parte de administración de la plataforma se controlan los recursos del sistema, los usuarios registrados y es donde se asignan los rangos **IP** a su correspondiente servidor de peticiones. En el proceso de planificación de tareas el servidor central debe recibir las tareas enviadas y asignarlas al servidor de peticiones con mayor cantidad de recursos disponibles, así como decirle a los servidores de peticiones libres a que servidor de petición ocupado debe ayudar en su ejecución.

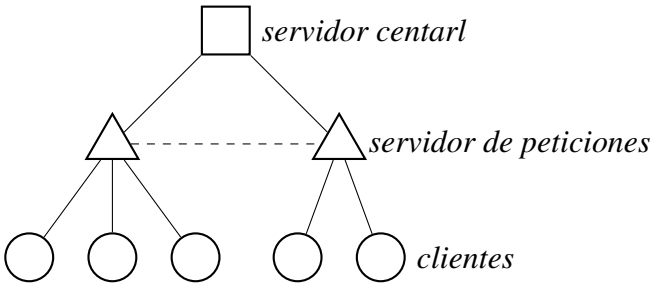


Figura 2.1: Topología de la plataforma.

Los servidores de peticiones serán los encargados de dividir las tareas en unidades de trabajo y enviárselas a los clientes. Con los resultados de las unidades de trabajo ejecutadas en los clientes el servidor de peticiones debe formar un resultado final. Un servidor de peticiones puede manejar al mismo tiempo n tareas. Por defecto, el sistema viene para que el servidor de peticiones atienda una tarea en cada instante. Además, el servidor de peticiones debe tener control de las unidades de trabajo creadas y de cuales todavía no se tiene respuesta. Un servidor de peticiones libre tienen la capacidad de colaborar con otro que esté ocupado en la ejecución de una tarea.

El módulo cliente frecuentemente solicita unidades de trabajo, ejecuta la unidad asignada y envía el resultado al servidor de peticiones. Todas las comunicaciones son iniciadas por los clientes. Cuando una unidad de trabajo es asignada, el cliente no volverá a contactar al servidor de peticiones hasta culminar la tarea o si una excepción es lanzada. Si a la hora de hacer una solicitud no hay trabajos, el cliente esperará un tiempo determinado para volver hacer una solicitud. Si en algún momento el cliente no puede contactar al servidor de peticiones, entonces se conecta al servidor central en búsqueda de un nuevo servidor de peticiones al que asociarse.

2.2.1 Planificación a nivel de servidores de peticiones

El servidor central tiene una lista con todos los servidores de peticiones. Esta lista estará ordenada de forma ascendente según la función $S_i = N_i + 1/TC_i$. Donde N_i es el balance de carga del servidor i: $N_i = 0$ si el servidor de peticiones está libre, $N_i = 1$ si el servidor de peticiones está colaborando con otro servidor y $N_i = 2$ en otro caso. TC_i es el número de clientes asociados al servidor de peticiones i cuyo tiempo de petición promedio es menor a AVE_i^{γ}. AVE_i^{γ} es el tiempo promedio de petición de los clientes asociados al servidor de peticiones i y se actualiza siguiendo la **ecuación (2.1)**.

$$AVE_i^{\gamma} = \alpha \Delta RT_i + (1-\alpha) * AVE_i^{\gamma-1} \tag{2.1}$$

2.2.2 Planificación a nivel de clientes

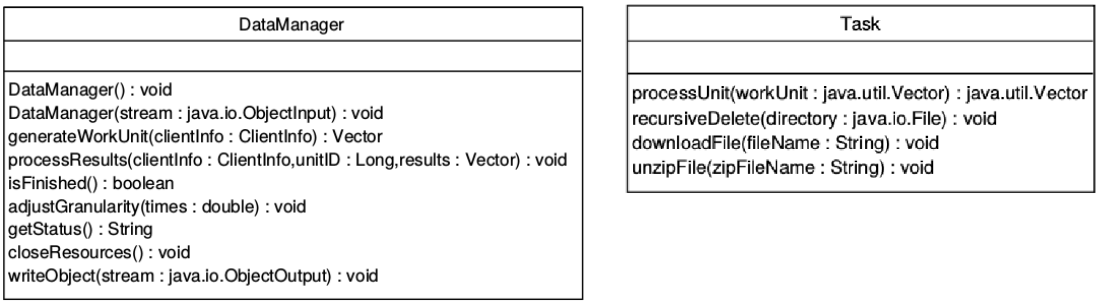
Cada servidor de peticiones tiene asociado clientes que periódicamente solicitan unidades de trabajo. Cada vez que un cliente solicita trabajo el servidor de peticiones le envía una unidad de trabajo, siguiendo la estrategia *Pure Self-Scheduling*, lo cual garantiza un balance de carga óptimo, pero genera la mayor cantidad de comunicaciones posible. Esto influye negativamente en el tiempo de ejecución de la aplicación, ya que para cada mensaje hay que dedicarle tiempo de la ejecución a su preparación(*overhead*) y envío(comunicaciones).

2.2.3 Mecanismo de colaboración

El mecanismo de colaboración funciona entre los servidores de peticiones y ocurre cuando hay servidores de peticiones libres mientras otros están ejecutando algún trabajo. El servidor central le dice al servidor de peticiones libre a que servidor ocupado ayudará. Cuando esto ocurre el servidor de peticiones libre funcionará como un proxy entre los clientes asociados a él y el servidor de peticiones al que va a ayudar. Si el sistema tuviera m servidores de peticiones con n clientes asociados cada uno, y solo uno servidor de peticiones tiene tarea, este servidor recibiría a lo máximo $(m-1)+n$ peticiones concurrente, mucho menor a las $m*n$ peticiones que podría tener si solo hubiera un servidor de peticiones.

2.2.4 API de desarrollo

La implementación de una aplicación para la plataforma requiere extender de dos clases java: *DataManager* y *Task*. La clase *DataManager* se ejecuta en los servidores de peticiones mientras que la clase *Task* es ejecutada por los clientes. El propósito de la clase *DataManager*, ver **2.2a**, consiste en generar las unidades de trabajo que serán enviadas a los clientes, procesar los resultados devueltos por los clientes, adjustar la granularidad si así lo desea el desarrollador, generar el estado de la ejecución y determinar cuando debe terminar de ejecutarse la tarea. El parámetro *clientInfo* en la función *generateWorkUnit* garantiza que la aplicación se ejecute sobre ciertas restricciones; por ejemplo, que se pueda ejecutar sobre una plataforma determinada. El objetivo de la clase *Task*,ver **2.2b**, es procesar las unidades de trabajo generadas por el servidor de peticiones.



(a) **Data Manager** (b) **Task**

Figura 2.2: Diagramas UML de las clases necesarias para la implementación de una aplicación en la plataforma.

2.3 Adaptación a la plataforma de las estrategias self-scheduling

2.3.1 Peso de los clientes

La plataforma se ejecuta sobre un ambiente no dedicado y heterogéneo. Por ser un sistema heterogéneo es necesario tener en cuenta la capacidad computacional de las estaciones de trabajo del sistema. Para estimar la capacidad computacional de las estaciones de trabajo se utilizó el algoritmo LINPACK [43, 58]. Con las operaciones punto flotante por segundo($flops/s$) estimadas utilizando el algoritmo LINPACK se puede determinar el peso de cada cliente como $w_i = \frac{f_i}{F}$, donde f_i son los $flops/s$ que se pueden ejecutar en el cliente i, y $F = \sum_i f_i$.

2.3.2 Guided Self-Scheduling

Guided self-scheduling(GSS) realiza la planificación basado en una función decreciente, teniendo en cuenta la cantidad de unidades que faltan por procesar y el número de procesadores existentes. Esto lo hace porque se ejecuta sobre un ambiente homogéneo. Para poder implementar GSS sobre la plataforma es necesario que GSS tenga en cuenta el peso de cada uno de los clientes en el sistema y envíe a cada uno de ellos un número de unidades proporcional a sus pesos. Para esto, cada vez que un cliente realice una petición a un servidor de peticiones, GSS enviará $c = \frac{w_i * R}{W}$, donde w_i es el peso del cliente i, R es la cantidad de unidades que quedan por asignar y W es la suma de los pesos de todos los clientes que pertenecen al servidor de peticiones.

2.3.3 Factoring Self-Scheduling

Factoring self-scheduling(FSS) al igual que GSS realiza la planificación basado en una función decreciente teniendo en cuenta la cantidad de unidades que faltan por procesar y el número de procesadores existentes. La diferencia es que FSS divide la planificación por etapas, donde en cada etapa se planifica un porción de las unidades por procesar y se reparten a partes iguales entre los procesadores. FSS asume que todos los procesadores están activos, pero en T-arenal el estado de los clientes puede cambiar durante la ejecución y el sistema no es capaz de saberlo en tiempo real. De igual manera, determinar la cantidad de clientes a tener en cuenta para una etapa de FSS es difícil y costosa computacionalmente. Para FSS en T-arenal, solo se puede tener en cuenta el cliente que está haciendo la petición actual para cada etapa. Cada vez que un cliente i realice una petición, el servidor de peticiones enviará $c = \frac{w_i * R}{\alpha * W}$, tomando $\alpha = 2$ como proponen en [33].

2.3.4 Trapezoid Self-Scheduling

Trapezoid Self-scheduling(TSS) propone un función lineal decreciente para asignar el número de tareas a cada procesador. Con esta función se intenta minimizar los costos de sincronización y comunicación. Para esto se inicializan el tamaño del primer y del último envío y a partir de éstos una constante decreciente, que es la que define la linealidad de la función. Para su adaptación a la plataforma se toma la idea del trabajo [30] donde a cada cliente se le asigna una cantidad de unidades en dependencia de cuán mejor es con respecto al cliente con peor rendimiento. Cuando se comienza la ejecución de una tarea el servidor de peticiones inicializa las variables F, L, S , D, declaradas en la sección 1.3.4 utilizando el **Algoritmo 4**.

Algoritmo 4: Inicialización de las variables globales.

Entrada: $N \rightarrow$ Número total de unidades
Entrada: $W = \sum_{i=1}^{N_c} w_i \rightarrow w_i$ es el peso del cliente i, N_c es el número total de clientes
Entrada: $p_{worst} \rightarrow$ número de núcleos del cliente cuyo peso es $min(w_k), k \in [1, N_c]$
Salida: D, $current$

1 $F = \lfloor \frac{N}{2*W} \rfloor \rightarrow$ Número de unidades que serán enviadas en el primer envío
2 $L = 2 * p_{worst} \rightarrow$ Número de unidades que serán enviadas en el último envío
3 $S = \lceil \frac{2*N}{F+L} \rceil \rightarrow$ número de pasos en la planificación
4 $current = F \rightarrow$ Inicialización de $current$ (para ser utilizado **Algoritmo 5**)
5 $D = \lfloor \frac{F-L}{S-1} \rfloor \rightarrow$ En cada paso del Algoritmo 5, $current$ es reducida con D unidades

Cada vez que un cliente realiza una solicitud, el servidor de peticiones envía c unidades, donde c se calcula utilizando el **Algoritmo 5**.

Algoritmo 5: Cálculo de la cantidad de unidades a enviar en un paso con TSS para T-arenal.

Entrada: $w_i \rightarrow$ Peso del cliente i que realiza la petición
Entrada: D, $current \rightarrow$ Salida del Algoritmo 4
Salida: $c \rightarrow$ Cantidad de unidades que serán enviadas en el paso actual

1 $c = 0$
2 $w_f = min(w_k), k \in [1, N_c] \rightarrow$ Peso del peor cliente
3 $v = \lfloor w_i / w_f \rfloor$
4 $c = v_i * current - \frac{v_i*(v_i-1)}{2} * D$
5 $current = current - v_i * D$
6 *return* c

2.4 Modificaciones en el diseño de la plataforma

En esta sección se muestran las principales modificaciones que se realizaron en el diseño de la plataforma para poder adaptar los algoritmos antes descritos. En específico, se presentan los cambios realizados en los módulos cliente y servidor de peticiones.

2.4.1 Servidor de Peticiones

El servidor de peticiones se encarga de la ejecución y planificación de las tareas. Para esto, los servidores de peticiones solicitan periódicamente tareas al servidor central. Una vez asignadas, los servidores de peticiones se encargan de dividir las tareas en pequeñas

subtareas y de repartir las diferentes unidades de trabajos entre los clientes, cada vez que estos soliciten tareas y estén autorizados a interactuar con el mismo. Los servidores de peticiones también se encargar del control de unidades pendientes de respuestas y otras que han expirado por algún error ocurrido. El servidor de peticiones procesa el resultado de cada una de las subtareas generadas, para finalmente construir el resultado de la tarea original. En la figura 2.3 se muestran los principales paquetes que interactúan en los servidores de peticiones.

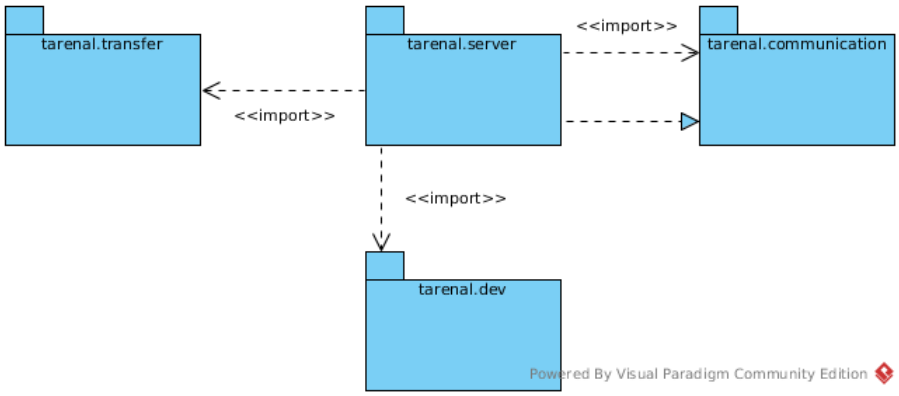


Figura 2.3: Paquetes del servidor de peticiones. Vista de relación

Los objetivos de los paquetes mostrados en la figura 2.3 son:

- En ***En tarenal.communication*** se definen las interfaces de comunicación tanto para los clientes como para el servidor central.
- ***tarenal.dev*** define las clases para el desarrollo de una aplicación distribuida.
- ***tarenal.transfer*** define e implementa las interfaces para la transferencia de archivos.
- ***tarenal.server*** implementa las interfaces de comunicación para que puedan realizarse todas las funcionalidades con el servidor central y los clientes. Implementa además, las funcionalidades relacionadas con el procesamiento de una ejecución.

El paquete ***tarenal.server*** se encarga de la ejecución de la tarea. En este paquete se implementa toda la lógica de dividir las tareas en unidades de trabajo. También se define toda la lógica para tener la información de las unidades de trabajo que se han asignado a algún cliente y están pendientes de respuesta. Las principales clases se muestran en la figura 2.4 y sus objetivos son:

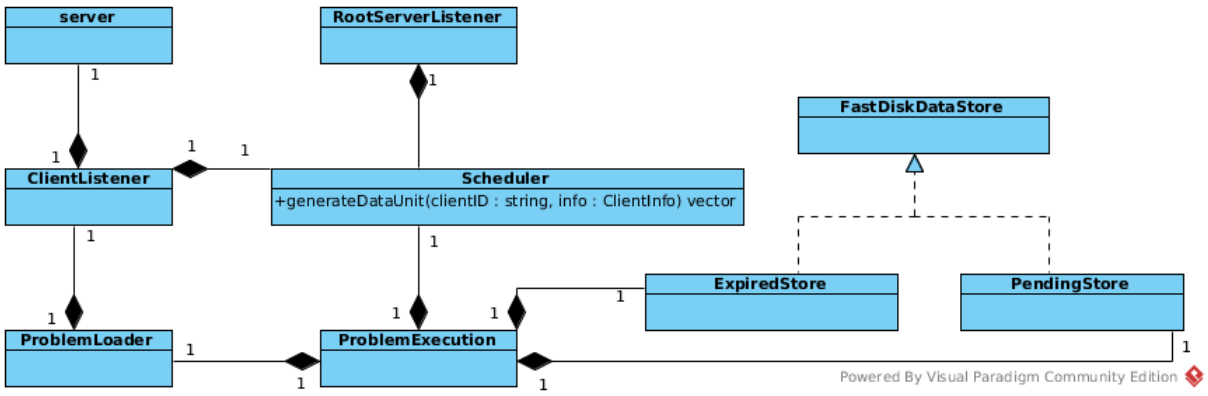


Figura 2.4: Diagrama de clases del paquete ***tarenal.server***.

- ***Server*** configura los parámetros con los que inicia el servidor de peticiones. Estos valores iniciales son definidos en los ficheros de configuración. Además de crear los objetos que serán publicados para el intercambio de información.

- ***RootServerListener*** implementa las funcionalidades delegadas desde el servidor central.

- ***ClientListener*** es la clase que implementa las comunicaciones con el cliente.

- ***Scheduler*** hereda de *Thread*. Es la clase que se encarga de realizar las peticiones de tareas al servidor central, así como la encargada de distribuir las unidades de trabajo entre los clientes y tener control de las unidades que están creadas y no tienen respuestas.

- ***ProblemExecution*** representa a una ejecución que está siendo atendida. Contiene información como el identificador, la prioridad, cantidad de unidades generadas, cantidad de resultados recibidos, los requerimientos computacionales de la ejecución que se atiende, entre otros.

- ***ProblemLoader*** es la responsable de cargar dinámicamente las librerías y las clases que serán ejecutadas.

- ***PendingStore*** brinda las funcionalidades para gestionar las unidades de trabajo que se encuentran pendientes de respuestas y todavía no han expirado.

- ***ExpiredStore*** brinda las funcionalidades para gestionar las unidades de trabajo que expiran por alguna causa.

La clase ***Scheduler*** es la encargada de generar las unidades de trabajo que serán enviadas cada vez que un cliente solicite tareas. La función ***generateDataUnit*** genera la unidad de trabajo y se la entrega a la clase ***ClientListener*** que es la clase que se encarga de las comunicaciones entre los clientes y los servidores de peticiones. A la clase ***Scheduler*** se le añade una función que en dependencia de la cantidad de unidades de trabajo sin asignar y las características del cliente que realiza la petición define la cantidad de unidades que serán enviadas al cliente.

2.4.2 Cliente

El cliente es el subsistema que realiza una solicitud de una unidad de trabajo al servidor de peticiones correspondiente, hace el procesamiento de la unidad asignada y envía el resultado al servidor de peticiones. En la figura 2.5 se muestra los principales paquetes del módulo cliente así como las relaciones entre ellos.

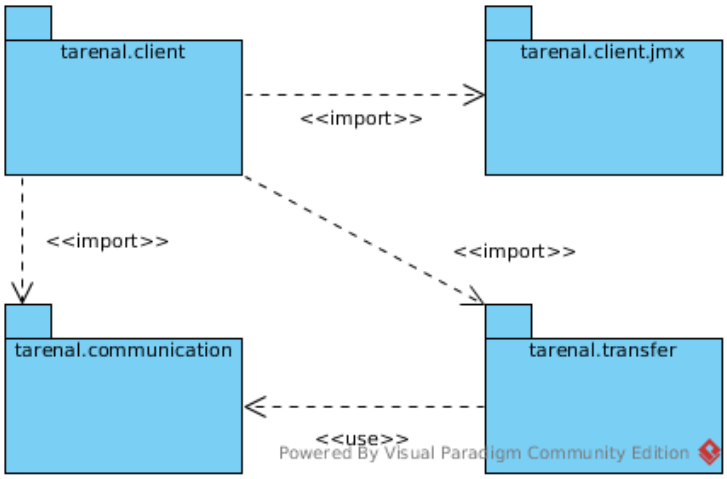


Figura 2.5: Paquetes del Cliente. Vista de relación

Los objetivos de los paquete mostrados en la figura 2.5 son:

- En ***tarenal.communication*** se definen e implementan las entidades que gestionan la comunicación con el servidor de peticiones.
- En ***tarenal.transfer*** se definen e implementan las clases e interfaces para la transferencia de archivos con el servidor de peticiones correspondiente.
- En ***tarenal.client*** se implementan todas las funcionalidades para recibir, procesar y enviar los resultados de las unidades de trabajo asignadas.

- En ***tarenal.client.jmx*** se definen e implementan las interfaces de comunicación con el front-end del cliente.

El paquete ***tarenal.client***, ver el diagrama de clases en la figura 2.6, define las acciones de petición, recibimiento y ejecución de la unidad de trabajo, así como el envío del resultado.

En el paquete ***tarenal.client*** se definen las siguientes clases:

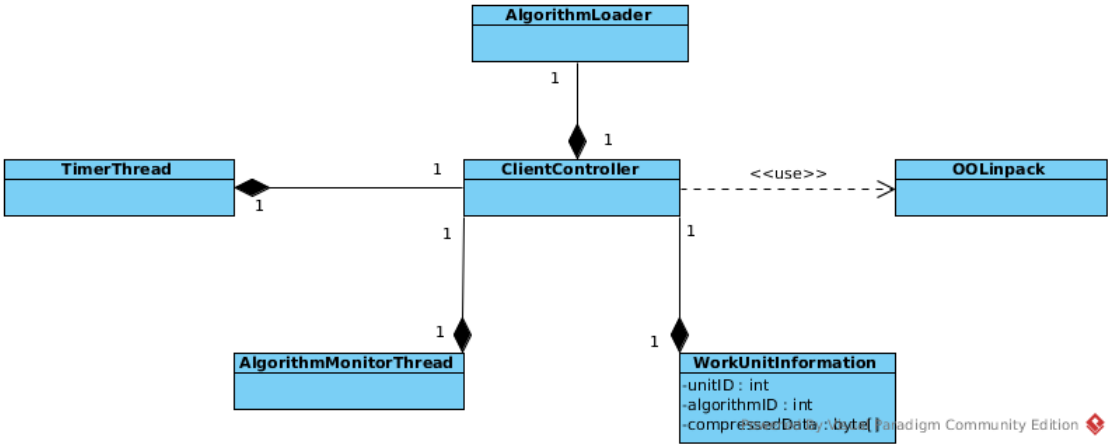


Figura 2.6: Paquetes del Cliente. Vista de relación.

- ***OOLinkpack*** calcula la cantidad de operaciones por segundo($flops/s$) que se pueden realizar en el cliente utilizando el algoritmo *Linkpack*.
- ***ClientController*** hereda de la clase ***Thread*** y su tarea es gestionar las funcionalidades del cliente, guardar información sobre la unidad de trabajo que procesa y el estado de la ejecución, así como determinar la información de *hardware* del cliente.
- ***AlgorithmLoader*** carga mediante introspección, la clase que procesará la unidad de trabajo asignada.
- ***AlgorithmMonitorThread*** hereda de *Thread* y se encarga de monitorear el estado del procesamiento de una unidad de trabajo.
- ***TimerThread*** se utiliza para limitar el tiempo de ejecución de las unidades de trabajo en los clientes. Este tiempo es enviado por el servidor de peticiones.
- ***WorkUnitInformation*** contiene toda la información relacionada a la unidad de trabajo a procesar.

Para adaptar las estrategias de planificación es necesario que el cliente pueda ejecutar varias unidades de trabajo al mismo tiempo. Para esto la clase ***ClientController*** debe manejar varias instancias de las clases ***AlgorithmMonitorThread*** y ***TimerThread***. Lo mismo pasa con la clase ***WorkUnitInformation***, inicialmente tiene control del algoritmo que se desea ejecutar y de la unidad de trabajo que se desea analizar. La diferencia radica en la cantidad de datos que se va a analizar. En la versión original a cada cliente se le envía una unidad de trabajo. En la nueva versión de la plataforma, con un planificador incluido que envía una cantidad de unidades de trabajo proporcional a la importancia del cliente, cada cliente debe ser capaz de atender y ejecutar más de una unidad de trabajo al mismo tiempo.

En la figura 2.7 se muestra el diagrama de clases del paquete cliente luego de realizar las transformaciones, ahora con dos clases incluidas, que permitirán la ejecución por parte del cliente de varias unidades de trabajo al mismo tiempo. Para esto, la nueva versión de la clase ***WorkUnitInformation*** tendrá asociada una lista con la información de cada una de las unidades de trabajo que se desean analizar en el cliente. Por otro lado se introduce la clase ***ExecutorThread***, que será la clase encargada de monitorear la ejecución de una unidad de trabajo y el tiempo en el que se está ejecutando. De esta manera la clase ***ClientController*** podrá ejecutar varias unidades de trabajo al unísono.

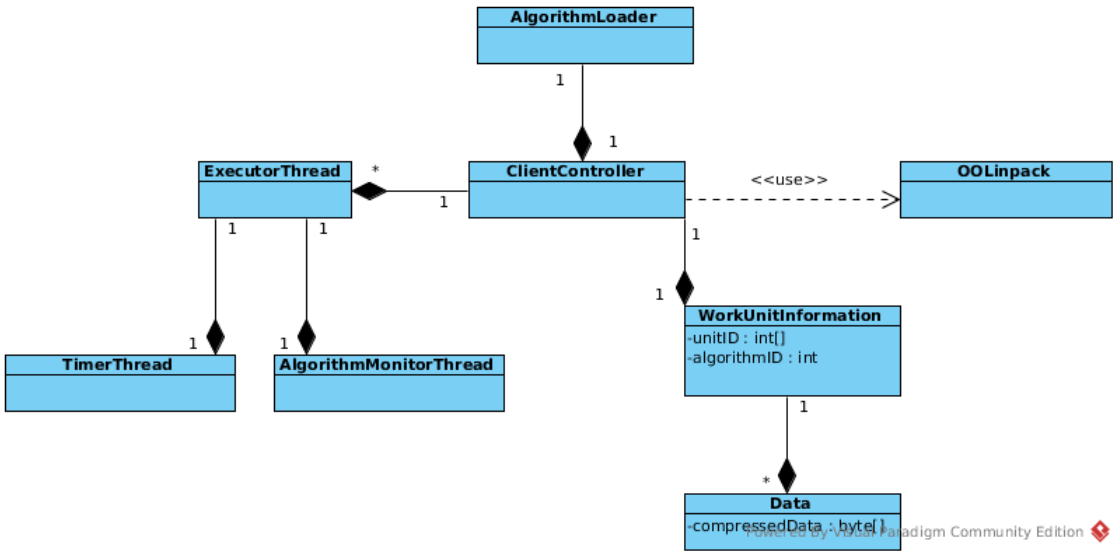


Figura 2.7: Paquetes del Cliente. Vista de relación.

2.5 Conclusiones parciales

En el presente capítulo se enumeraron las principales características de la plataforma de cálculo distribuido T-arenal. De igual manera se muestra como fueron evaluados los clientes utilizando el algoritmo LINPACK para determinar su importancia en la plataforma. Finalmente se introducen las adaptaciones de las estrategias *self-scheduling* a la plataforma. Como conclusiones de este capítulo se puede tener que 1) la plataforma de cálculo distribuido T-arenal brinda una excelente alternativa al cálculo de alto rendimiento o desempeño; 2) el algoritmo LINPACK proporciona un rasgo característico de las estaciones de trabajo, utilizado para caracterizar a los clientes teniendo en cuenta las operaciones punto flotante por segundo(*flops/s*) de estas; 3) se implementaron las estrategias *self-scheduling* sobre T-arenal teniendo en cuenta la capacidad computacional de cada uno de las estaciones de trabajo; y 4) se aprovechan los recursos de las estaciones de trabajo asociadas a la plataforma para un mejor desempeño de esta.

Capítulo 3. Pruebas y resultados

Capítulo 3. Pruebas y resultados

3.1 Introducción

En este capítulo se describen las pruebas realizadas para comparar el rendimiento de cada estrategia en la plataforma. Se mencionan escenarios y ambientes que se utilizaron en las pruebas ejecutadas, así como las estaciones de trabajo que se utilizaron en las pruebas. Se realizaron dos pruebas independientes, la primera empleando un problema de multiplicar una matriz por un vector. En esta prueba se analiza el tiempo de ejecución, las comunicaciones y el balance de carga de las estrategias propuestas con respecto al planificador implementado en la plataforma y se determina la existencia de diferencia significativa en los resultados obtenidos. En la segunda prueba se utiliza un problema de acoplamiento molecular, en diferentes escenarios simulando situaciones donde el sistema es no dedicado. Para esto se analiza el tiempo de ejecución, el número de comunicaciones y el número de unidades generadas cuando el escenario es no dedicado.

3.2 Pruebas realizadas sobre el problema de multiplicación matriz-vector.

3.2.1 Descripción del problema a resolver y del ambiente de pruebas.

Para comparar el rendimiento de las estrategias de planificación descritas en este trabajo con respecto a la estrategia implementada en la plataforma se realizaron pruebas utilizando un problema matriz-vector con tamaños $8192x8192$ y $16384x16384$. Se tomó como unidad de trabajo la multiplicación de una fila de la matriz por el vector. Para realizar las pruebas se utilizaron 1,2,4 y 8 nodos como clientes y un servidor de peticiones mostrándose en la tabla 3.1 la distribución de nodos por ambiente. Se realizaron 10 ejecuciones de cada algoritmo sobre un mismo juego de datos existiendo variables que muestran ausencia de normalidad luego de aplicar la prueba Kolmogorov Smirnov [14, 53]; tomándose la mediana como medida de tendencia central. Para determinar si existía diferencia significativa entre todas las estrategias tomadas en cuenta se utilizó la prueba no paramétrica Kruskal-Wallis [14, 53] , así como la prueba no paramétrica Mann-Whitney [14, 53] corregidas por el método **BY** para determinar si existía diferencia

por pares.

Tabla 3.1: Descripción de los nodos utilizados por ambientes.

Procesador	1	2	4	8
Intel(R) Core(TM) i3-2130 CPU @ 3.40GHz	1	1	1	2
Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz	0	1	1	4
Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz	0	0	2	2

3.2.2 Tiempo de ejecución

En la tabla 3.2 se muestra la mediana del tiempo de ejecución obtenido por cada estrategia, en cada uno de los ambientes para un tamaño de $8192x8192$. En esta se puede observar como las estrategias *Guided self-scheduling(GSS)*, *Factoring self-scheduling(FSS)* y *Trapezoid self-scheduling(TSS)* mejoran el tiempo de ejecución alcanzado por la estrategia actual del sistema(*Pure self-scheduling(PSS)*). Se puede ver también como entre las estrategias *self-scheduling* la que mayor tiempo de ejecución tiene es *Guided self-scheduling*. *Trapezoid self-scheduling* alcanza los mejores tiempos de ejecución para todos los ambientes, excepto cuando se utiliza un solo nodo, donde la estrategia que mejor tiempo de ejecución alcanza es *Factoring self-scheduling*.

En la tabla 3.3 se muestra la mediana del tiempo de ejecución obtenido por cada estrategia, en cada uno de los ambientes para un tamaño $16384x16384$. En esta se puede observar un comportamiento similar al ocurrido en la tabla 3.2, donde las estrategias GSS, FSS y TSS mejoran el tiempo de ejecución alcanzado por PSS. *Guided self-scheduling* vuelve a ser la estrategia con peor tiempo de ejecución y*Trapezoid self-scheduling* alcanza los mejores tiempos de ejecución con excepción de cuando se utilizan 4 nodos, donde la estrategia que mejor tiempo de ejecución alcanza es *Factoring self-scheduling*, aunque la diferencia con *Trapezoid self-scheduling* es mínima. Para un nodo *Guided self-scheduling* devuelve un error en el nodo de ejecución. Esto se debe a que para *Guided self-scheduling* se necesita más memoria que la reservada para la ejecución del cliente.

Tabla 3.2: Mediana del tiempo de ejecución(segundos) del problema para un vector de 8192 elementos.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	4836	1272	**1260**	1271
2	2465	685	682	**670**
4	1251	398	378	**367**
8	660	293	289	**212**

Tabla 3.3: Mediana del tiempo de ejecución(segundos) del problema para un vector de 16384 elementos.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	10000	–	3005	**2969**
2	5083	1684	1563	**1513**
4	2662	988	**968**	970
8	1411	648	587	**546**

Para comprobar que la diferencia entre los tiempos de ejecución de las distintas estrategias son significativas se utilizó la prueba no paramétrica Kruskal-Wallis para determinar si existían diferencias significativas globales entre todas las observaciones, así como la prueba no paramétrica Mann-Whitney para determinar los pares entre los que existen diferencias significativas, de encontrarse diferencias globales por Kruskal-Wallis. Las tablas 3.4-3.7 muestran la diferencia significativa del tiempo de ejecución entre las estrategias de planificación ejecutadas para un tamaño 8192x8192, mientras que las tablas 3.8-3.11 muestran la diferencia significativa del tiempo de ejecución entre las estrategias de planificación ejecutadas para un vector de 16384 elementos. En estas se puede observar como para todos los ambientes la diferencia si es significativa entre las estrategias GSS, FSS y TSS con respecto a la estrategia PSS.

Las tablas 3.4 y 3.5 muestran las diferencias entre las estrategias cuando se utiliza un nodo y dos nodos para un problema con 8192 elementos. Para un nodo y dos nodos la prueba no paramétrica Kruskal Wallis devolvió un valor de $9{,}67 * 10^{-6}$ y 0,004 respectivamente, valores inferiores a 0,5, indicando que existe diferencia significativa global entre las variables. Se observa como entre las estrategias GSS, FSS y TSS no existe diferencia significativa, pero si hay diferencia significativa cuando las comparamos con la estrategia PSS.

Tabla 3.4: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 1 nodo($9{,}679473e-06$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	1.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	1.00	0.96	0.00

En la tabla 3.6 se muestra la diferencia significativa entre las estrategias de planificación cuando se utilizan 4 nodos para un problema con 8192 elementos. En este ambiente la prueba no paramétrica Kruskal Wallis devolvió un valor de 0,002. En la tabla

Tabla 3.5: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 2 nodos(0,003936367).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	1.00		
PSS (pure)	0.04	0.04	
TSS (trapezoid)	1.00	0.13	0.04

se puede observar el mismo comportamiento de las tablas 3.4 y 3.5 en cuanto a que hay diferencia significativa entre las estrategias GSS, FSS y TSS con respecto a la estrategia PSS. Se puede observar que FSS no tiene diferencia significativa con GSS y TSS, pero si existe diferencia significativa entre los tiempos de ejecución de GSS y TSS, estrategia con los mejores tiempos de ejecución.

Tabla 3.6: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 4 nodos(0,001833963).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.44		
PSS (pure)	0.03	0.03	
TSS (trapezoid)	0.61	0.03	0.03

La tabla 3.7 muestra las diferencias significativas cuando se ejecutan las estrategias para 8 nodos y el vector tiene 8192 elementos. En este caso la prueba no paramétrica Kruskal Wallis devolvió un valor de $1{,}15 * 10^{-5}$. Se puede observar como en este escenario *Trapezoid self-scheduling* tiene diferencia significativa con el resto de las estrategias.

Tabla 3.7: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 8192 elementos en 8 nodos($1{,}152619e-05$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.67		
PSS (pure)	0.01	0.02	
TSS (trapezoid)	0.00	0.00	0.01

Las tablas 3.8-3.11 muestran la diferencia significativa del tiempo de ejecución entre las estrategias de planificación ejecutadas para un vector de 16384 elementos. La tabla 3.8 muestra la diferencia significativa de los algoritmos medidos para 16384 elementos ejecutados en un solo nodo, donde la prueba no paramétrica Kruskal Wallis devolvió el valor de 0,004. En ella se muestra la existencia de diferencia significativa entre las

estrategias GSS, FSS y TSS con respecto a la estrategia PSS, así como la inexistencia de diferencia significativa entre *Factoring self-scheduling* y *Trapezoid self-scheduling*. Estas dos características se van a repetir para los otros ambientes en los que se realizaron pruebas.

Tabla 3.8: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 1 nodo(0,004266478).

	FSS (factoring)	PSS (pure)
PSS (pure)	0.02	
TSS (trapezoid)	0.43	0.01

La tabla 3.9 muestra la diferencia significativa del tiempo de ejecución entre los algoritmos probados en 2 nodos para 16384 unidades. En este ambiente la prueba no paramétrica Kruskal Wallis devolvió un valor de $7{,}33 * 10^{-7}$. La tabla muestra como los únicos algoritmos que no tienen diferencia significativa entre ellos son *Factoring self-scheduling* y *Trapezoid self-scheduling*. De igual manera muestra como existe diferencia significativa entre estas estrategias y el resto de las estrategias utilizadas.

Tabla 3.9: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 2 nodos($7{,}339108e-07$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.05	0.00	0.00

La diferencia significativa del tiempo de ejecución entre los algoritmos cuando utilizan 4 nodos y el tamaño del problema es de 16384 unidades es mostrada en la tabla 3.10, donde el valor devuelto por la prueba no paramétrica Kruskal Wallis fue $2{,}79 * 10^{-5}$. Para esta cantidad de nodos el algoritmo con mejor tiempo de ejecución fue *Factoring self-scheduling* y la estrategia con mejor *speedup* y eficiencia fue *Guided self-scheduling*. Se puede ver como no existe diferencia significativa entre las estrategias GSS, FSS y TSS, aunque si existe entre estas y la estrategia PSS.

En la tabla 3.11 se muestra la diferencia significativa de las estrategias probadas cuando el número de unidades es de 16384 y el número de nodos es 8 teniendo un valor de diferencia global según Kruskal Wallis igual a $4{,}45 * 10^{-8}$. Aquí se muestra como existe diferencia significativa entre todas las estrategias comprobadas.

Tabla 3.10: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 4 nodos($2{,}796184e-05$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.09		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.73	0.73	0.00

Tabla 3.11: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del tiempo de ejecución para cada una de las estrategias para un vector de 16384 elementos en 8 nodos($4{,}454626e-08$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.02	0.00	0.00

3.2.3 *Speedup*

Las tablas 3.12 y 3.13 muestran el *speedup* alcanzado cuando se ejecuta el problema para un vector de 8192 y 16384 elementos, para 2,4 y 8 nodos, tomando como referencia el tiempo de ejecución de cada estrategia en un nodo. Los resultados alcanzados muestran como la estrategia PSS alcanza el mejor *speedup* y eficiencia, pero también obtiene los peores tiempos de ejecución, ver tablas 3.2 y 3.3.

La tabla 3.12 muestra el *speedup* alcanzado cuando el tamaño del vector es de 8192 elementos. Se pude observar como crece el *speedup* a la vez que crece el número de clientes y como *Trapezoid self-scheduling* alcanza el mejor *speedup* para todos los ambientes de ejecución cuando se tienen en cuenta las nuevas estrategias .

Tabla 3.12: *Speedup* del problema para un vector de 8192 elementos.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	1.00	1.00	1.00	1.00
2	**1.96**	1.85	1.84	**1.89**
4	**3.86**	3.19	3.33	**3.46**
8	**7.32**	4.34	4.35	**5.98**

En la tabla 3.13 se muestra el *speedup* cuando el tamaño del vector es de 16384 elementos. En el caso de *Guided self-scheduling* se muestra el *speedup* alcanzado con 4 y 8 nodos, tomando como referencia el tiempo alcanzado para 2 nodos. Esto se debe a que no se pudo calcular el tiempo de ejecución de *Guided self-scheduling* con 1 nodo por requerimientos de memoria, ver tabla 3.3. Se puede ver como el *speedup* para 16384

elementos tiene el mismo comportamiento que para 8192 elementos, donde el *speedup* crece a la vez que crece el número de clientes y como *Trapezoid self-scheduling* alcanza el mejor *speedup* para todos los ambientes de ejecución con excepción de 4 nodos, donde el mejor *speedup* lo alcanza *Guided self-scheduling*. Se puede decir que de forma general el *speedup* alcanzado para 8192 elementos es mayor que el alcanzado para 16382 elementos.

Tabla 3.13: *Speedup* del problema para un vector de 16384 elementos.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	1.00	-	1.00	1.00
2	**1.97**	1.00	1.92	**1.96**
4	**3.76**	**3.41**	3.10	3.06
8	**7.09**	5.20	5.12	**5.44**

3.2.4 Eficiencia

Las tablas 3.14 y 3.15 muestran la eficiencia alcanzada cuando se ejecuta el problema para un vector de 8192 y 16384 elementos, para 2,4 y 8 nodos, tomando como referencia el tiempo de ejecución de cada estrategia en un nodo. De manera similar a lo ocurrido con el *speedup*, para los dos casos, la mejor eficiencia es alcanzada por la estrategia PSS, mientras que entre las restantes estrategias *self-scheduling* el que mejor eficiencia alcanza es *Trapezoid self-scheduling*. Se puede ver como la eficiencia para todos los casos es superior al 50 % y como la eficiencia de las estrategias cuando el tamaño del problema es de 8192 elementos es superior a cuando el tamaño del problema es de 16384 elementos.

Tabla 3.14: Eficiencia del problema para un vector de 8192 elementos.

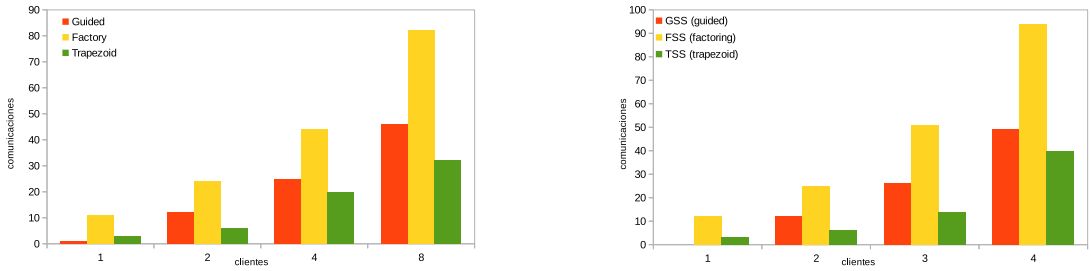
nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	1.00	1.00	1.00	1.00
2	**0,98**	0,92	0,92	**0,94**
4	**0,96**	0,79	0,83	**0,86**
8	**0,91**	0,54	0,54	**0,74**

Tabla 3.15: Eficiencia del problema para un vector de 16384 elementos.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
1	1.00	-	1.00	1.00
2	**0,98**	1.00	0,96	**0,98**
4	**0,94**	**0,85**	0,78	0,77
8	**0,88**	0,65	0,64	**0,68**

3.2.5 Comunicaciones

Las figuras 3.1a y 3.1b muestran la mediana de las comunicaciones generadas por cada uno de las estrategias GSS, FSS y TSS, en cada una de los ambientes de prueba, para un tamaño de 8192x8192 y 16384x16384 elementos. Nos referimos a comunicaciones como la cantidad de veces que el servidor de peticiones envía unidades de trabajo a los clientes. No se muestra las comunicaciones generadas por la estrategia PSS porque esta genera tantas comunicaciones como unidades de trabajo tiene la tarea, i.e. para el problema de tamaño 8192x8192 genera 8192 unidades de trabajo. En las figuras 3.1a y 3.1b se puede observar como *Trapezoid self-scheduling* es la estrategia que genera la menor cantidad de comunicaciones en todos los ambientes, con excepción de cuando se utiliza solo un nodo. En este caso la estrategia que tiene menor cantidad de comunicaciones es *Guided self-scheduling*. *Factoring self-scheduling* es la estrategia con mayor cantidad de comunicaciones, esto se debe a que *Factoring self-scheduling* sacrifica las comunicaciones en pos de alcanzar un mejor balance de carga.



(a) 8192 (b) 16384

Figura 3.1: Mediana de las comunicaciones generadas para cada una de las pruebas realizadas.

Las tablas 3.16-3.23 muestran la diferencia significativa por pares entre la cantidad de comunicaciones generadas, para cada una de las estrategias *self-scheduling*, en cada uno de los ambientes para el problema de multiplicar una matriz por un vector con tamaño 8192x8192 y 16382x16382. En el caso de 8192 unidades de trabajo, la prueba no paramétrica Kruskal-Wallis devolvió para cada ambiente(1,2,4 y 8 nodos) los valores $5{,}04 * 10^{-7}$, 0,0002, $3{,}19 * 10^{-7}$ y $1{,}77 * 10^{-6}$ respectivamente, indicando diferencia significativa global entre cada una de las estrategias en todos los ambientes. Para el caso de 16384 unidades la prueba no paramétrica Kruskal-Wallis devolvió para cada ambiente(1,2,4 y 8 nodos) los valores 0,0009, $2{,}64 * 10^{-6}$, $1{,}49 * 10^{-7}$y $6{,}72 * 10^{-8}$ respectivamente, indicando también diferencia significativa global entre cada una de las estrategias en todos los ambientes. Se puede observar en las tablas como siempre existe

diferencia significativa por pares entre la cantidad de comunicaciones generadas por cada estrategia utilizada.

Tabla 3.16: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 1 nodo($5{,}043477e-07$).

	FSS (factoring)	GSS (guided)
GSS (guided)	$2{,}9e-05$	
TSS (trapezoid)	$2{,}9e-05$	$2{,}9e-05$

Tabla 3.17: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 2 nodos(0.0002352295).

	FSS (factoring)	GSS (guided)
GSS (guided)	0.0033	
TSS (trapezoid)	0.0033	0.0033

Tabla 3.18: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 4 nodos($3{,}198628e-07$).

	FSS (factoring)	GSS (guided)
GSS (guided)	0.00011	
TSS (trapezoid)	0.00011	0.00011

Tabla 3.19: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 8192 elementos y 8 nodos($1{,}779841e-06$).

	FSS (factoring)	GSS (guided)
GSS (guided)	0.00024	
TSS (trapezoid)	0.00024	0.00024

Tabla 3.20: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 1 nodo(0,0009111189).

	FSS (factoring)
TSS (trapezoid)	0.0016

Tabla 3.21: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 2 nodos(2,640758$e-06$).

	FSS (factoring)	GSS (guided)
GSS (guided)	0.00023	
TSS (trapezoid)	0.00023	8.8e-05

Tabla 3.22: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 4 nodos(1,499105$e-07$).

	FSS (factoring)	GSS (guided)
GSS (guided)	6.1e-05	
TSS (trapezoid)	4.4e-05	4.4e-05

Tabla 3.23: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY de las comunicaciones generadas para cada una de las estrategias para un vector de 16384 elementos y 8 nodos(6,727263$e-08$).

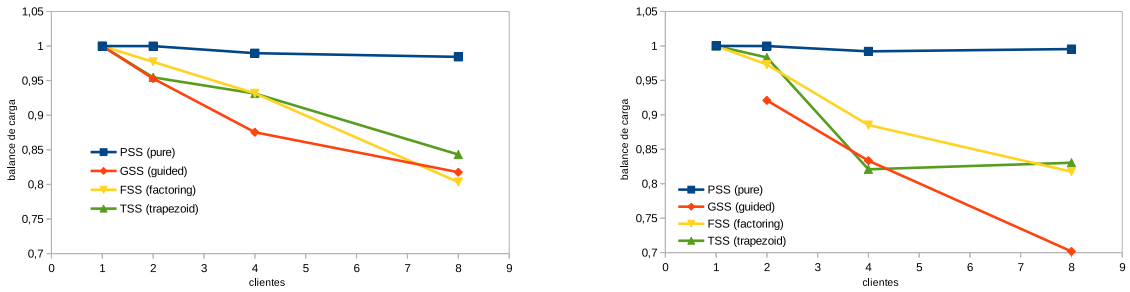
	FSS (factoring)	GSS (guided)
GSS (guided)	9.6e-06	
TSS (trapezoid)	0.00014	0.00014

3.2.6 Balance de Carga

Las figuras 3.2a y 3.2b muestran la mediana del balance de carga para cada uno de las estrategias, en cada una de los ambientes de prueba, para un tamaño de 8192x8192 y 16384x16384 elementos. En la fig 3.2a se muestra el balance para 8192 elementos. Se puede ver como para 1 nodo el balance siempre es 1, esto se debe al hecho de que sea solo un nodo el que esté ejecutando tareas.También se puede ver como el mejor balance lo tiene la estrategia PSS. Esto era esperado teniendo en cuenta la cantidad unidades que

atiende cada cliente. Entre las estrategias GSS, FSS y TSS se puede ver como *Factoring self-scheduling* tiene mejor balance de carga para 2 y 4 nodos. También se puede observar como *Trapezoid self-scheduling* mejora el balance de carga con respecto a las otras estrategias a medida que aumenta el número de clientes, obteniendo mejor balance de carga para 8 nodos.

Similares características ocurre cuando el problema sube a 16384 elementos, ver figura 3.2b, donde *Pure self-scheduling* mantiene mejor balance de carga que el resto de los esquemas comprobados. La diferencia radica a que entre las estrategias GSS, FSS y TSS el que mejor balance de carga obtiene es *Trapezoid self-scheduling*, alcanzando mejor balance de carga para 2 y 8 nodos. Solo siendo superado por *Factoring self-scheduling* con 4 nodos, donde el tiempo de ejecución en este ambiente fue mejor también para *Factoring self-scheduling*, ver tabla 3.3, demostrando la importancia que tiene el balance de carga para el tiempo de ejecución total.



(a) 8192 (b) 16384

Figura 3.2: Balance de carga para cada una de las pruebas realizadas

Las tablas 3.24-3.26 muestran la diferencia significativa por pares del balance carga, para cada uno de los algoritmos utilizados, en 2,4 y 8 nodos, para el problema de multiplicar una matriz por un vector con tamaño 8192x8192. Para cada uno de estos ambientes(2,4 y 8 nodos) la prueba no paramétrica Kruskal-Wallis, para determinar si existía diferencia significativa global devolvió los valores 0,0007, 0,0003 y 0,005, señalando diferencia significativa global entre el balance de carga de las estrategias utilizadas.

Se puede ver que existe diferencia significativa entre la estrategia PSS con respecto a las estrategias GSS, FSS y TSS, evidenciando que *Pure self-scheduling* es la estrategia que maximiza el balance de carga. Podemos notar como entre las estrategias *Factoring self-scheduling* y *Trapezoid self-scheduling* no existe en ningún ambiente diferencia significativa. Se puede observar también como entre *Factoring self-scheduling* y *Guided self-scheduling* existe diferencia significativa para 2 y 4 nodos, que es cuando *Factoring*

self-scheduling tiene mejor balance que *Guided self-scheduling*. Para 8 nodos, tabla 3.26 se puede ver como no existe diferencia significativa por pares entre las estrategias GSS, FSS y TSS.

Tabla 3.24: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 2 nodo(0,0007530543).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.02		
PSS (pure)	0.02	0.02	
TSS (trapezoid)	0.19	0.76	0.02

Tabla 3.25: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 4 nodos(0,0003140667).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.04		
PSS (pure)	0.01	0.01	
TSS (trapezoid)	1.00	0.01	0.01

Tabla 3.26: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 8192 elementos y 8 nodos(0,005464635).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	1.00		
PSS (pure)	0.01	0.01	
TSS (trapezoid)	0.82	0.28	0.01

Las tablas 3.27-3.29 muestran la diferencia significativa por pares del balance carga, para cada uno de los algoritmos utilizados, en 2,4 y 8 nodos, para el problema de multiplicar una matriz por un vector con tamaño $16384x16384$. Similar a lo que ocurre cuando el problema es de tamaño 8192 unidades, se puede ver que existe diferencia significativa entre la estrategia PSS con respecto al resto de las estrategias *self-scheduling*, evidenciando que *Pure self-scheduling* es la estrategia que maximiza el balance de carga.

Podemos notar como entre las estrategias *Factoring self-scheduling* y *Trapezoid self-scheduling* no existe diferencia significativa para 2 y 8 nodos, pero si hay diferencia

Tabla 3.27: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 2 nodo(6,642106$e-07$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.17	0.00	0.00

cuando se ejecutan en 4 nodos, donde *Factoring self-scheduling* tiene mejor balance de carga, y mejor tiempo de ejecución, mostrando que la diferencia en el balance de carga si es determinante. Se puede ver también como en todos los ambientes las diferencias que tienen *Factoring self-scheduling* y *Trapezoid self-scheduling* con *Guided self-scheduling* si son significativas.

Tabla 3.28: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 4 nodos(4,938106$e-08$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.00	0.06	0.00

Tabla 3.29: Significaciones de la prueba por pares de Mann-Whitney corregidas por el método BY del balance de carga para un vector de 16384 elementos y 8 nodos(9,126325$e-09$).

	FSS (factoring)	GSS (guided)	PSS (pure)
GSS (guided)	0.00		
PSS (pure)	0.00	0.00	
TSS (trapezoid)	0.15	0.00	0.00

3.3 Pruebas realizadas sobre un problema de acoplamiento molecular.

3.3.1 Descripción de las pruebas y los escenarios de prueba.

Para evaluar y comparar las estrategias descritas en el capítulo 2 se realizaron pruebas utilizando una herramienta de acoplamiento molecular [60] sobre un conjunto de 1000 moléculas para la proteína 4UDC [23] descargada de *Protein Data Bank(PDB)* [52]. Se

utilizó un servidor de peticiones y se realizaron las pruebas en 3 ambientes(4,8 y 16 nodos). La tarea se dividió en 1000 unidades de trabajo, una molécula por unidad de trabajo.

La Tabla 3.30 muestra para cada ambiente, las especificaciones de hardware(Tipo de CPU y Cantidad de RAM) de las estaciones de trabajo utilizadas.

Tabla 3.30: Descripción de los nodos utilizados por ambientes.

Procesador y RAM	4	8	16
Core(TM) 2 Duo CPU E4500 @ 2.20GHz ; 1GB	1	2	5
Core(TM) i3-2120 CPU @ 3.30GHz ; 4GB	1	3	6
Core(TM) i3-2130 CPU @ 3.40GHz ; 4GB	1	2	3
Core(TM) i5-4460 CPU @ 3.20GHz ; 4GB	1	1	2

Se comparó el rendimiento de las estrategias de planificación PSS, GSS, FSS y TSS como se describe en el capítulo 2 para los siguientes 5 escenarios, representando un ambiente dedicado y 4 no dedicados (simulados):

- Escenario 1: los clientes están siempre activos (ambiente dedicado).
- Escenario 2: la mitad de los clientes están inactivos durante el 5 % del tiempo de ejecución del escenario 1.
- Escenario 3: la mitad de los clientes están inactivos durante el 10 % del tiempo de ejecución del escenario 1.
- Escenario 4: la mitad de los clientes están inactivos durante el 20 % del tiempo de ejecución del escenario 1.
- Escenario 5: la mitad de los clientes están inactivos durante un tiempo fijo de 2700 segundos.

Se escogieron estos escenarios para simular(lo más cercano posible) ambientes no dedicados en situaciones reales, dado que un servidor de peticiones puede manejar varios clientes distribuidos entre dos o más laboratorios y los clientes en un laboratorio pueden estar siendo utilizados o apagados. No obstante, en el estudio de escalabilidad, con un número creciente de clientes, estos intervalos de inactividad implican un incremento en el tiempo de ejecución, el cual acentúa el problema de las comunicaciones y la regeneración de unidades. Por lo tanto, también realizamos pruebas con clientes que están inactivos durante un por ciento fijo del tiempo. Se tomó un intervalo fijo de 2700 segundos= 45 minutos porque corresponde con el periodo que se demora una clase en un laboratorio.

Tabla 3.31: Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 1.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	41464	20749	**19771**	20283
8	21135	10821	**10767**	10857
16	10759	5760	**5598**	6588

Tabla 3.32: Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 2.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	44074	25802	**20791**	20833
8	21079	13758	11308	**10995**
16	11357	7561	**6114**	6593

Note que en los escenarios 2,3 y 4, el tiempo de ejecución de referencia fue tomado del escenario 1, dependiendo de la estrategia utilizada. Por ejemplo, si se va a ejecutar la aplicación en el escenario 2, utilizando PSS, el 5 % es calculado, tomando como referencia, el tiempo de ejecución para PSS en el escenario 1.

3.3.2 Tiempo de ejecución

Se muestra en las tablas 3.31-3.35 la mediana del tiempo de ejecución de la aplicación utilizando los 4 planificadores para los 5 escenarios diseñados, en los 3 distintos ambientes(4, 8 y 16 nodos). Para todos los escenarios PSS tiene los tiempos de ejecución más altos. Este fenómeno era esperado teniendo en cuenta que no se aprovechan los recursos computacionales de cada cliente, la cantidad de comunicaciones que se realizan y el tiempo dedicado a la creación(*overhead*) de las unidades.

En el escenario 1(ambiente dedicado) la estrategia FSS tiene rendimiento ligeramente mejor que los algoritmos GSS y TSS, ver Tabla 3.31, se puede ver también como para 2 de los 3 ambientes(8 y 16 nodos) GSS tiene mejor tiempo de ejecución que TSS. En el escenario 2, GSS tiene un tiempo de ejecución mayor a FSS y TSS, mientras que FSS alcanza mejores tiempos de ejecución que TSS en 2 de los 3 ambientes, ver Tabla 3.32.

En el escenario 3, la diferencia entre los planificadores GSS, FSS y TSS es mínima utilizando 4 nodos, siendo TSS quien tiene el tiempo de ejecución más alto en este ambiente. FSS siempre alcanza los mejores tiempos de ejecución, ver Table 3.33.

Tabla 3.34: Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 4.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4 nodos	45699	24623	**24202**	24505
8 nodos	23502	15106	12842	**12697**
16 nodos	12048	7685	**7048**	7335

Tabla 3.33: Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 3.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	42917	22220	**21799**	22451
8	21846	14560	**11066**	11545
16	11630	7711	**6295**	7195

Para el escenario 4, se tiene un comportamiento similar al escenario 2, donde TSS alcanza mejor tiempo de ejecución para 8 nodos y FSS alcanza mejores tiempos de ejecución para 4 y 16 nodos, siendo GSS el que peor tiempo de ejecución obtiene entre las estrategias GSS, FSS y TSS 3.34.

Para el escenario 5 se tiene que TSS alcanza los mejores tiempo de ejecución para los tres ambientes. Esto podría suceder porque,teniendo en cuenta el tiempo de ejecución del escenario 1, 2700 segundos implica un mayor porciento de tiempo inactivo de los clientes cuando se ejecuta FSS que cuando se ejecuta TSS, ver Tabla 3.35.

Tabla 3.35: Mediana del tiempo de ejecución(segundos) de la aplicación cuando se utiliza los 4 planificadores en el escenario 5.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	42143	23330	22304	**21857**
8	20707	13039	12461	**11879**
16	12409	8480	8859	**8118**

3.3.3 *Speedup*

Las Tablas 3.36-3.40 muestran el *speedup* obtenido por la aplicación en 8 y16 nodos, comparado con el tiempo alcanzado en 4 nodos, cuando se utilizan cada uno de las estrategias. Para cada uno de de los escenarios, PSS tiene el mejor *speedup*, pero también alcanza los peores tiempos de ejecución, Tablas 3.31-3.35.

Tabla 3.36: Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 1

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**1.96**	**1.92**	1.84	1.87
16	**3.85**	**3.60**	3.53	3.08

Tabla 3.37: Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 2.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**2.09**	1.88	1.84	**1.89**
16	**3.88**	**3.41**	3.40	3.16

Tabla 3.38: Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 3.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**1.96**	1.53	**1.97**	1.94
16	**3.69**	2.88	**3.46**	3.12

Tabla 3.39: Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 4.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**1.94**	1.63	1.88	**1.93**
16	**3.79**	3.2	**3.43**	3.34

Tabla 3.40: Speedup de la aplicación cuando se utiliza los 4 planificadores en el escenario 5.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**2.03**	1.79	1.79	**1.84**
16	**3.40**	**2.75**	2.52	2.69

Para GSS, FSS y TSS, el *speedup* obtenido, cuando se va desde 4 a 16 nodos, es entre 2,5 y 3,5. Se puede observar como el *speedup* de GSS en el primer escenario es el mejor entre las estrategias GSS, FSS y TSS, a diferencia de cuando se va desde 0 % a 20 %, donde el *speedup* de GSS disminuye cuando se compara con el *speedup* obtenido por FSS y TSS.

3.3.4 Eficiencia

Las tablas 3.41-3.45 muestran la eficiencia de la aplicación en los diferentes escenarios para cada una de las estrategias. Se puede observar como para cada uno de los escenarios PSS alcanza la mejor eficiencia, obteniendo en dos escenarios una eficiencia superior a 1. Esto se podía prever teniendo en cuenta que el *speedup* de PSS es el mejor en todos los casos, ver tablas 3.36-3.40, pero también es la estrategia con mayores tiempo de ejecución, ver tablas 3.31-3.35.

Tabla 3.41: Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 1.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**0,98**	**0,96**	0,92	0,93
16	**0,96**	**0,90**	0,88	0,77

Tabla 3.42: Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 2.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**1,05**	0,94	0,92	**0,95**
16	**0,97**	**0,85**	**0,85**	0,79

Tabla 3.43: Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 3.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**0,98**	0,76	**0,98**	0,97
16	**0,92**	0,72	**0,87**	0,78

Se puede observar como la eficiencia de GSS es la mejor entre las estrategias GSS, FSS y TSS en el escenario dedicado y decrece con respecto a estas cuando se va desde

0 % a 20 %. De igual forma se puede observar como la eficiencia de la aplicación en el sistema, en cada uno de los escenarios y ambientes nunca es menor a 0.5.

Tabla 3.44: Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 4.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**0,97**	0,82	0,94	**0,96**
16	**0,95**	0,80	**0,86**	0,84

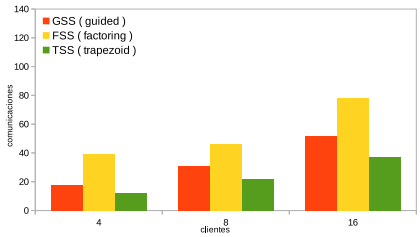
Tabla 3.45: Eficiencia de la aplicación cuando se utiliza los 4 planificadores en el escenario 5.

nodos	PSS (pure)	GSS (guided)	FSS (factoring)	TSS (trapezoid)
4	**1**	1	1	1
8	**1,02**	0,89	0,89	**0,92**
16	**0,85**	0,69	0,63	**0,67**

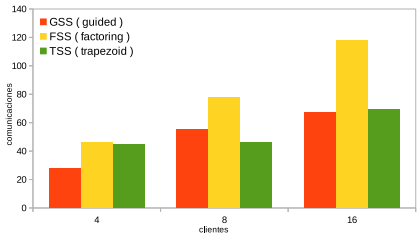
3.3.5 Comunicaciones

Cada vez que un servidor envía trabajos a un cliente se envía un paquete compuesto por dos partes: la primera parte(*header*) esta formada por información acerca de la tarea, identificador de la tarea, momento de creación del paquete, información acerca del servidor de peticiones y el algoritmo que se desea ejecutar, la segunda parte está conformada por las unidades de trabajo que se envían. Cuando PSS es utilizado, cada comunicación contiene solo una unidad de trabajo, mientras que cuando se utiliza una de las otras estrategias varias unidades de trabajo son enviadas en un mismo paquete. Mientras que con PSS se necesita generar el (*header*) para cada unidad de trabajo, con las otras estrategias esto no es necesario, reduciendo el tiempo de generación(*overhead*) de la planificación.

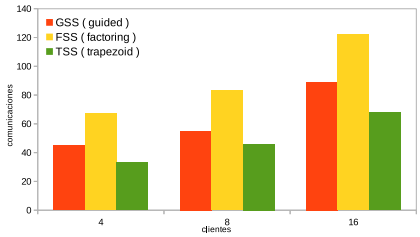
Las Figuras 3.3a, 3.3b, 3.3c, 3.3d y 3.3e presentan respectivamente el número de comunicaciones generadas por GSS, FSS y TSS para los escenarios 1,2,3,4 y 5 respectivamente. No se muestran las comunicaciones generadas por PSS porque este genera tantas comunicaciones como unidades de trabajo tengan las tareas, para la aplicación de pruebas serían 1000 o más comunicaciones. Para los 5 escenarios FSS genera la mayor cantidad de comunicaciones. Esto es esperado porque FSS sacrifica las comunicaciones para obtener mejor balance de carga. En la mayoría de los casos TSS es la estrategia con menor cantidad de comunicaciones.



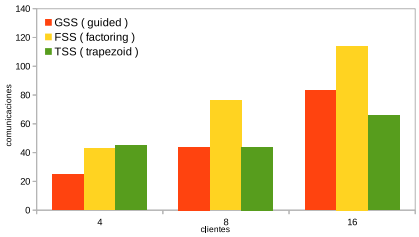
(a) Escenario 1



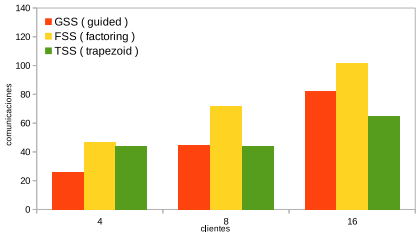
(b) Escenario 2



(c) Escenario 3



(d) Escenario 4



(e) Escenario 5

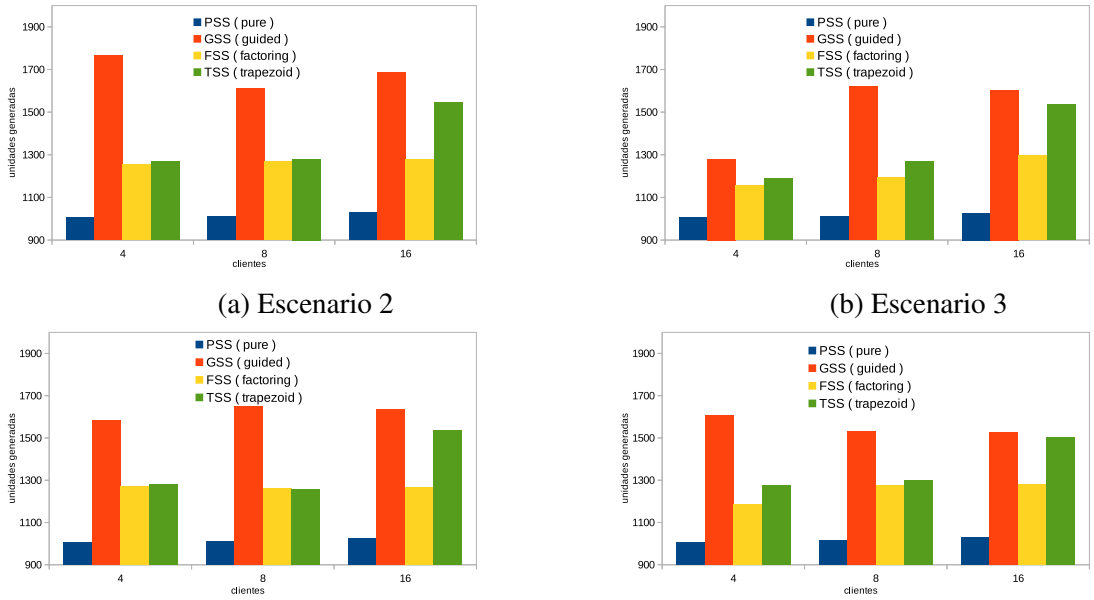
Figura 3.3: Número de comunicaciones, cuando se utiliza las estrategias GSS, PSS, FSS y TSS para los 5 escenarios.

3.3.6 Unidades generadas.

El número de unidades generadas en el escenario 1(ambiente dedicado) siempre es 1000. En el resto de los escenarios, los clientes se pueden desactivar o tardarse en la terminación de sus ejecuciones, lo que implica que el servidor de peticiones tiene que generar y planificar otra vez las unidades de trabajo asignadas a estos clientes. Las Figuras 3.4a, 3.4b, 3.4c y 3.4d muestran las unidades generadas para cada planificador para los escenarioss 2,3,4 y 5 respectivamente.

PSS genera el menor número de unidades(ligeramente por encima de 1000) porque con PSS un cliente nunca está ejecutando más de una unidad. GSS es el planificador que genera la mayor cantidad de unidades de trabajo, esto se debe a que GSS envía mayor cantidad de unidades por paquete que FSS y TSS. El número de unidades generadas siempre es menor para FSS. Para 4 y 8 nodos la diferencia entra la cantidad de unidades generadas por FSS y TSS es mínima, fenómeno que no ocurre para 16 nodos, donde dicha diferencia aumenta

significativamente. También se puede ver como la diferencia entre las unidades generadas por FSS y TSS aumenta a la par que aumenta el número de nodos.



(a) Escenario 2

(b) Escenario 3

(c) Escenario 4

(d) Escenario 5

Figura 3.4: Unidades generadas necesarias para terminar la ejecución de la aplicación.

3.4 Conclusiones parciales

En este capítulo se presentaron los resultados del análisis del rendimiento de un problema de multiplicación matriz-vector con diferentes tamaños y una aplicación de acoplamiento molecular sobre la plataforma de cálculo distribuido T-arenal utilizando 4 estrategias de planificación. En un primer conjunto de pruebas se compararon las estrategias de planificación utilizando un problema matriz-vector. En estas pruebas se determinó que TSS obtiene mejor rendimiento que las demás estrategias comparadas en cuanto tiempo de ejecución y comunicaciones. En un segundo conjunto de pruebas se compararon las estrategias de planificación en un ambiente dedicado y varios no dedicados utilizando diferentes métricas para evaluar diferentes objetivos de nuestro trabajo. Como conclusiones se puede mencionar 1) que los algoritmos FSS y TSS tienen mejor rendimiento que GSS y estos tres tienen un rendimiento mucho mejor que PSS, reduciendo significativamente el tiempo de ejecución de las aplicaciones utilizadas en las pruebas; 2) que TSS normalmente requiere la menor cantidad de comunicaciones para completar las tareas; 3) que en los ambientes no dedicados(simulados) FSS es el que necesita generar menor cantidad de tareas para completar la ejecución de la aplicación utilizada; 4) que PSS es el que tiene un mejor *speedup*, aunque también es el que tiene

mayor tiempo de ejecución; 5) que TSS es el que mejor tiempo de ejecución alcanza en la plataforma para la mayoría de las pruebas realizadas.

Conclusiones

Atendiendo a las pruebas realizadas se llegó a las siguientes conclusiones:

1. Los algoritmos de planificación *self-scheduling* fueron creados para la planificación de iteraciones independientes en ciclos paralelos, pero pueden ser adaptados a sistemas distribuidos heterogéneos y no dedicados ya que pueden tener en cuenta, para la etapa de planificación, las características computacionales de los nodos asociados al sistema.

2. Utilizar el algoritmo LINPACK para medir la importancia de los nodos en el proceso de planificación de los esquemas *self-scheduling* en un ambiente no dedicado y heterogéneo permite realizar una asignación de tareas más eficiente, mejorando significativamente el rendimiento de las aplicaciones que se ejecutan en él.

3. *Trapezoid self-scheduling* genera en la mayoría de los casos la menor cantidad de comunicaciones, influyendo esto, en un mejor rendimiento del tiempo de ejecución total de las aplicaciones.

4. *Factoring self-scheduling* alcanza los mejores tiempos de ejecución cuando su balance de carga es significativamente superior.

5. Las diferentes pruebas realizadas demuestran que las estrategias *self-scheduling* propuestas, utilizando el algoritmo LINPACK para estimar la importancia de los clientes, mejoran el rendimiento de T-arenal, reduciendo el tiempo de ejecución y la cantidad de comunicaciones.

Recomendaciones

1. Tener en cuenta el historial de los clientes para evaluar la importancia de estos en la plataforma.
2. Implementar un módulo *easybuild* para adaptar la plataforma a un ambiente cluster de alto desempeño.
3. Agregar T-arenal a la Plataforma de estudios bioinformáticos desarrollada en el CEMC.

Referencias

[1] Adam-Bourdarios, C., Cameron, D., Filipcic, A., Lancon, E., Wu, W. and Collaboration, A. [2015], 'Atlas@home: Harnessing volunteer computing for hep', *Journal of Physics: Conference Series* **664**(2).

[2] AlEbrahim, S. and Ahmad, I. [2017], 'Task scheduling for heterogeneous computing systems', *The Journal of Supercomputing* **73**(6), 2313–2338.

[3] Alexander, N. S. and Palczewski, K. [2017], 'Crowd sourcing difficult problems in protein science', *Protein Science* **26**(11).

[4] Alkhanak, E. N., Lee, S. P., Rezaei, R. and Parizi, R. M. [2016], 'Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues', *Journal of Systems and Software* **113**, 1 – 26.

[5] Alonso-Monsalve, S., García-Carballeira, F. and Calderón, A. [2017], 'ComBos: A complete simulator of Volunteer Computing and Desktop Grids', *Simulation Modelling Practice and Theory* **77**, 197–211.

[6] Anderson, E. J. and Linderoth, J. [2017], 'High throughput computing for massive scenario analysis and optimization to minimize cascading blackout risk', *IEEE Transactions on Smart Grid* **8**(3), 1427–1435.

[7] Anson, H., Thomas, D. B., Tsoi, K. H. and Luk, W. [2010], Dynamic scheduling monte-carlo framework for multi-accelerator heterogeneous clusters, *in* '2010 International Conference on Field-Programmable Technology', pp. 233–240.

[8] Apache Software Foundation [2018], 'Apache commons', https://commons.apache.org. Online; accedido 27 septiembre del 2018.

[9] Balaji, V., Maisonnave, E., Zadeh, N., Lawrence, B. N., Biercamp, J., Fladrich, U., Aloisio, G., Benson, R., Caubel, A., Durachta, J., Foujols, M.-A., Lister, G.,

Mocavero, S., Underwood, S. and Wright, G. [2017], 'CPMIP: measurements of real computational performance of Earth system models in CMIP6', *Geoscientific Model Development* **10**(1), 19–34.

[10] Bassini, S., Danelutto, M., Dazzi, P., Joubert, G. R. and Peters, F. J., eds [2018], *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, Vol. 32 of *Advances in Parallel Computing*, IOS Press.

[11] Belviranli, M. E., Bhuyan, L. N. and Gupta, R. [2013], 'A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures', *ACM Trans. Archit. Code Optim.* **9**(4), 57:1–57:20.

[12] Bockelman, B., Bejar, J. C. and Hover, J. [2017], 'Interfacing htcondor-ce with openstack', *Journal of Physics: Conference Series* **898**(9).

[13] Castro, J., Monasterio, V. and Carro, J. [2016], Volunteer computing approach for the collaborative simulation of electrophysiological models, *in* '2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Paris, France', IEEE, pp. 118–123.

[14] Chatfield, C. [2018], *Statistics for technology: a course in applied statistics*, Routledge.

[15] Chorazyk, P., Godzik, M., Pietak, K., Turek, W., Kisiel-Dorohinicki, M. and Byrski, A. [2017], 'Lightweight volunteer computing platform using web workers', *Procedia Computer Science* **108**, 948–957. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[16] Czarnul, P. [2018], *Parallel Programming for Modern High Performance Computing Systems*, Chapman and Hall/CRC.

[17] da Silva, J. O., Orellana, E. T. V. and Delgado, M. X. T. [2017], 'Development of a parallel version of phyml 3.0 using shared memory', *IEEE Latin America Transactions* **15**(5), 959–967.

[18] Dharmapala, P., Koneshvaran, L., Sivasooriyathevan, D., Ismail, I. and Kasthurirathna, D. [2017], Peer-to-peer distributed computing framework, *in* '2017 6th National Conference on Technology and Management (NCTM)', IEEE, pp. 126–131.

[19] *Distributed.net* [2018], http://www.distributed.net/Main_Page. Online; accedido 27 septiembre del 2018.

[20] Dominguez, L. A., Yildirim, B., Husker, A. L., Cochran, E., Christensen, C., Cruz-Atienza, V. M. and Lawrence, J. F. [2015], 'The red atrapa sismos (quake-catcher network in mexico): Assessing performance during large and damaging earthquakes', *Seismological Research Letters* **86**(3), 848–855.

[21] Dorronsoro, B. and Pinel, F. [n.d.], Combining machine learning and genetic algorithms to solve the independent tasks scheduling problem, *in* '2017 3rd IEEE International Conference on Cybernetics (CYBCONF)'.

[22] Díaz, J., Reyes, S., Niño, A. and Muñoz-Caro, C. [2009], 'Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: Application to internet-based grids of computers', *Future Generation Computer Systems* **25**(6), 617 – 626.

[23] Edman, K., Hosseini, A., Bjursell, M. K., Aagaard, A., Wissler, L., Gunnarsson, A., Kaminski, T., Köhler, C., Bäckström, S., Jensen, T. J. et al. [2015], 'Ligand binding mechanism in steroid receptors: From conserved plasticity to differential evolutionary constraints', *Structure* **23**(12), 2280–2290.

[24] Fajardo, E. M., Dost, J. M., Holzman, B., Tannenbaum, T., Letts, J., Tiradani, A., Bockelman, B., Frey, J. and Mason, D. [2015], 'How much higher can htcondor fly?', *Journal of Physics: Conference Series* **664**(6), 062014.

[25] Ganesan, N., Li, J., Sharma, V., Jiang, H. and Compagnoni, A. [2016], 'Process simulation of complex biological pathways in physical reactive space and reformulated for massively parallel computing platforms', *IEEE/ACM transactions on computational biology and bioinformatics* **13**(2), 365–379.

[26] García-González, L. A., García-Jacas, C. R., Acevedo-Martínez, L., Trujillo-Rasúa, R. A. and Roose, D. [2018], Self-scheduling for a heterogeneous distributed platform, *in* 'Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy', pp. 232–241.

[27] García-Jacas, C. R., Aguilera-Mendoza, L., González-Pérez, R., Marrero-Ponce, Y., Acevedo-Martínez, L., Barigye, S. J. and Avdeenko, T. [2015], 'Multi-server

approach for high-throughput molecular descriptors calculation based on multi-linear algebraic maps', *Molecular Informatics* **34**(1), 60–69.

[28] Golub, G. H. and Ortega, J. M. [2014], *Scientific computing: an introduction with parallel computing*, Elsevier.

[29] Guler, H., Cambazoglu, B. B. and Ozkasap, O. [2015], 'Task allocation in volunteer computing networks under monetary budget constraints', *Peer-to-Peer Networking and Applications* **8**(6), 938–951.

[30] Han, Y. and Chronopoulos, A. T. [2017], 'Scalable loop self-scheduling schemes for large-scale clusters and cloud systems', *International Journal of Parallel Programming* **45**(3), 595–611.

[31] Heinze, R., Dipankar, A., Henken, C. C., Moseley, C., Sourdeval, O., Trömel, S., Xie, X., Adamidis, P., Ament, F., Baars, H. et al. [2017], 'Large-eddy simulations over germany using icon: a comprehensive evaluation', *Quarterly Journal of the Royal Meteorological Society* **143**(702), 69–100.

[32] Hummel, S. F., Schmidt, J., Uma, R. N. and Wein, J. [1996], Load-sharing in heterogeneous systems via weighted factoring, *in* 'Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures', SPAA '96, ACM, New York, NY, USA, pp. 318–328.

[33] Hummel, S. F., Schonberg, E. and Flynn, L. E. [1992], 'Factoring: A method for scheduling parallel loops', *Commun. ACM* **35**(8), 90–101.

[34] Kaur, M. and Kadam, S. S. [2017], 'Discovery of resources using madm approaches for parallel and distributed computing', *Engineering Science and Technology, an International Journal* **20**(3), 1013–1024.

[35] Kopal, N. [2018], *Secure Volunteer Computing for Distributed Cryptanalysis*, Kassel university press GmbH.

[36] Kruskal, C. P. and Weiss, A. [1985], 'Allocating independent subtasks on parallel processors', *IEEE Transactions on Software Engineering* **SE-11**(10), 1001–1016.

[37] Lautenschlagera, M., Adamidisa, P. and Kuhnb, M. [2015], 'Big data research at dkrz–climate model data production workflow', *Big Data and High Performance Computing* **26**, 133.

[38] Lawrence, J. F., Cochran, E. S., Chung, A., Kaiser, A., Christensen, C. M., Allen, R., Baker, J. W., Fry, B., Heaton, T., Kilb, D. et al. [2014], 'Rapid earthquake characterization using mems accelerometers and volunteer hosts following the m 7.2 darfield, new zealand, earthquake', *Bulletin of the Seismological Society of America* **104**(1), 184–192.

[39] Liu, F., Guo, W. and Zhao, X. [2018], Network resource management and scheduling in grid computing, *in* '2018 International Conference on Robots & Intelligent System (ICRIS)', IEEE, pp. 207–210.

[40] Luk, C.-K., Hong, S. and Kim, H. [2009], Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, *in* '2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)', IEEE, pp. 45–55.

[41] Maarala, A. I., Bzhalava, Z., Dillner, J., Heljanko, K. and Bzhalava, D. [2018], 'Virapipe: scalable parallel pipeline for viral metagenome analysis from next generation sequencing reads', *Bioinformatics* **34**(6), 928–935.

[42] Mao, M. and Humphrey, M. [2014], Resource provisioning in the cloud: An exploration of challenges and research trends, *in* 'Handbook of Research on Architectural Trends in Service-Driven Computing', IGI Global, pp. 589–612.

[43] Mohammadi, M. and Bazhirov, T. [2018], Comparative benchmarking of cloud computing vendors with high performance linpack, *in* 'Proceedings of the 2Nd International Conference on High Performance Compilation, Computing and Communications', HP3C, ACM, New York, NY, USA, pp. 1–5.

[44] Moldovan, D. I. [2014], *Parallel processing from applications to systems*, Elsevier.

[45] Montes, D., Añel, J. A., Pena, T. F., Uhe, P. and Wallom, D. C. [2017], 'Enabling boinc in infrastructure as a service cloud system', *Geoscientific Model Development* **10**(2), 811.

[46] Nardini, P., Böttinger, M., Scheuermann, G. and Schmidt, M. [2017], Visual Study of the Benguela Upwelling System using Pathline Predicates, *in* K. Rink, A. Middel, D. Zeckzer and R. Bujack, eds, 'Workshop on Visualisation in Environmental Sciences (EnvirVis)', The Eurographics Association.

[47] Nouman Durrani, M. and Shamsi, J. A. [2014], 'Volunteer computing: Requirements, challenges, and solutions', *Journal of Network and Computer Applications* **39**(1), 369–380.

[48] Pan, T., Flick, P., Jain, C., Liu, Y. and Aluru, S. [2018], 'Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems', *IEEE/ACM Transactions on Computational Biology and Bioinformatics* pp. 1–1.

[49] Pitt, E. and McNiff, K. [2001], *Java. rmi: The Remote Method Invocation Guide*, Addison-Wesley Longman Publishing Co., Inc.

[50] Polychronopoulos, C. D. and Kuck, D. J. [1987], 'Guided self-scheduling: A practical scheduling scheme for parallel supercomputers', *IEEE Transactions on computers* **100**(12), 1425–1439.

[51] *Quake-Catcher Network (QCN)* [2018], http://qcn.stanford.edu. Online; accedido 27 septiembre del 2018.

[52] Rose, P. W., Prlic, A., Altunkaya, A., Bi, C., Bradley, A. R., Christie, C. H., Costanzo, L. D., Duarte, J. M., Dutta, S., Feng, Z., Green, R. K., Goodsell, D. S., Hudson, B., Kalro, T., Lowe, R., Peisach, E., Randle, C., Rose, A. S., Shao, C., Tao, Y.-P., Valasatava, Y., Voigt, M., Westbrook, J. D., Woo, J., Yang, H., Young, J. Y., Zardecki, C., Berman, H. M. and Burley, S. K. [2017], 'The rcsb protein data bank: integrative view of protein, gene and 3d structural information', *Nucleic Acids Research* **45**(D1), D271–D281.

[53] Ross, S. M. [2014], *Introduction to probability and statistics for engineers and scientists*, Academic Press.

[54] Rudolph, L., Slivkin-Allalouf, M. and Upfal, E. [1991], A simple load balancing scheme for task allocation in parallel machines, *in* 'Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures', SPAA '91, ACM, New York, NY, USA, pp. 237–245.

[55] Schmidt, B., Gonzalez-Dominguez, J., Hundt, C. and Schlarb, M. [2017], *Parallel Programming: Concepts and Practice*, Morgan Kaufmann.

[56] *SETI@home, BOINC, and Volunteer Distributed Computing* [2012], *Annual Review of Earth and Planetary Sciences* **40**(1), 69–87.

[57] Strohmaier, E., Meuer, H. W., Dongarra, J. and Simon, H. D. [2018], 'Top500 list'. Online; accedido 27 septiembre del 2018.
URL: *URLhttps://www.top500.org/*

[58] Sukharev, P. V., Vasilyev, N. P., Rovnyagin, M. M. and Durnov, M. A. [2017], Benchmarking of high performance computing clusters with heterogeneous cpu/gpu architecture, *in* '2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)', pp. 574–577.

[59] Toth, D. M. [2008], Improving the productivity of volunteer computing, PhD thesis, Worcester Polytechnic Institute.

[60] Trott, O. and Olson, A. J. [2010], 'Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading', *Journal of computational chemistry* **31**(2), 455–461.

[61] Tsoi, K. H. and Luk, W. [2010], Axel: a heterogeneous cluster with fpgas and gpus, *in* 'Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays', ACM, pp. 115–124.

[62] Tzen, T. H. and Ni, L. M. [1993], 'Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers', *Ieee transactions on parallel and distributed systems* **4**(1), 87–98.

[63] Woltman, G. [2018], 'The great internet mersenne prime search (gimps)', https://www.mersenne.org. Online; accedido 27 septiembre del 2018.

[64] Xu, J. and Chronopoulos, A. [1999], Distributed self-scheduling for heterogeneous workstation clusters, *in* 'Proc. of the 12th Intl. Conf. on Parallel and Distributed Computing Systems', pp. 211–217.

[65] Xu, Y., Li, K., Hu, J. and Li, K. [2014], 'A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues', *Information Sciences* **270**, 255–287.

[66] Yang, C.-T. and Chang, S.-C. [2003], *A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1079–1088.

[67] Zhang, Z., Bockelman, B., Carder, D. W. and Tannenbaum, T. [2015], Lark: Bringing network awareness to high throughput computing, *in* '2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing', pp. 382–391.