DESARROLLO E IMPLEMENTACIÓN DE CÓDIGO EN PARALELO DE DINÁMICA MOLECULAR PARA EL ESTUDIO DE ZEOLITAS

AUTOR:
HERNANDO RECAMANN CHAUX

DIRECTOR:
CRISTIAN BLANCO TIRADO
CODIRECTOR:
MANUEL GUILLERMO FLOREZ

MAESTRÍA EN INGENIERÍA
ÁREA DE INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
UNIVERSIDAD INDUSTRIAL DE SANTANDER
BUCARAMANGA, JUNIO DE 2007

TABLA DE CONTENIDO

Pág.

INTRODUCCIÓN 1
1. PLANTEAMIENTO DEL PROBLEMA 2
2. MARCO TEORICO 3
2.1 DINÁMICA MOLECULAR 3
2.2 PARALELIZACIÓN 6
2.3 TIPO DE TRABAJO 9
3.CREACIÓN DE UN PROGRAMA EN PARALELO 11
3.1 PARTICIONAMIENTO O DESCOMPOSICIÓN 12
3.2 COMUNICACIÓN 13
3.3ORQUESTACIÓN,SINCRONIZACIÓN, AGRUPACIÓN O AGLOMERACIÓN 15
3.4 LOCALIZACIÓN, MAPEO O ASIGNACIÓN 15
4. TOPOLOGIA DE LA RED PARALELA 16
4.1 TIPOS DE TOPOLOGIA 16
5. MPI 17
5.1 FUNCIONES 17
5.2 EJEMPLOS 18
5.3 APLICACIONES 23
6. RESULTADOS 31
6.1 DESCRIPCIÓN DE PROGRAMAS 31
6.2 DESCRIPCIÓN DE ARCHIVOS SECUENCIALES 36
6.3 DIAGRAMA DE FLUJO 39
6.4 GRAFICACIÓN DE RESULTADOS 46
6.5 CONFIGURACIÓN DEL SISTEMA 55
4.5.1 MODELO Y MÉTODO 55
4.5.2 COMUNICACIÓN 55
4.5.3 ACELERACIÓN Y EFICIENCIA 56

4.5.4 COSTO DE PARALELIZACIÓN 58
4.5.5 RAPIDEZ ENTRE EWALD Y WOLF 59
CONCLUSIONES 61
RECOMENDACIONES 62
BIBLIOGRAFIA

LISTA DE ILUSTRACIONES

pag.

Ilustración 1. Potenciales de corto y largo alcance 5
Ilustración 2. SIMD 8
Ilustración 3. MIMD 8
Ilustración 4. Diagrama de Flujo 9
Ilustración 5. Pasos en la creación de un programa en paralelo 11
Ilustración 6. Descomposición del Dominio 12
Ilustración 7. Descomposición Funcional 13
Ilustración 8. Comunicación global en sumatoria de 8 tareas 14
Ilustración 9. Agrupación 15
Ilustración 10. Hipercubo de nodos 16
Ilustración 11. Cálculo de fuerzas 35

DESARROLLO E IMPLEMENTACIÓN DE CÓDIGO EN PARALELO DE DINÁMICA MOLECULAR PARA EL ESTUDIO DE ZEOLITAS *

HERNANDO RECAMANN CHAUX **

Palabras claves. *Dinámica Molecular, Zeolitas, Variables Termodinámicas, programación en paralelo, MPI, cluster Beowulf.*

La Dinámica Molecular (DM) es una técnica computacional que monitorea de manera determinística la posición de las partículas de un sistema termodinámico en función del tiempo. Esto se logra mediante la integración de las ecuaciones clásicas de movimiento de Newton. En la DM las fuerzas de interacción entre las partículas se calcula a través de los potenciales de corto (van der Waals y enlazantes) y largo alcance (Coulomb). Es bien conocido que el cálculo de la fuerza es la etapa mas demandante de recursos computacionales en los cálculos de DM, este cálculo representa aproximadamente 90% del tiempo que un procesador tardaría en cada ciclo. En este trabajo se presentan los resultados de la implementación de un código de DM en paralelo para el estudio de zeolitas, escrito en C++, que utiliza las librerías MPI y que corre en un "cluster" de computadores del tipo Beowulf. Después de verificar la validez de los resultados mediante la comparación con los obtenidos con el código en serie se procedió a hacer el análisis de rendimiento y eficiencia de la paralelización a través de la evaluación del modelo, el método, la comunicación, la eficiencia y el costo de paralelización, obteniendo datos muy satisfactorios. Por otro lado, se ha utilizado una nueva metodología para el cálculo de las interacciones electrostáticas, que disminuyen sustancialmente los tiempos de ejecución.

---

* Tesis de grado

** Facultad de Ingenierías Fisico-mecánicas. Escuela de Ing. de sistemas e Informática
Maestría en Ingeniería Informática. Director: Cristian Blanco Tirado

DEVELOPMENT AND IMPLEMENTATION OF CODE IN PARALLEL OF MOLECULAR DYNAMICS FOR THE STUDY OF ZEOLITES *

HERNANDO RECAMANN CHAUX **

Keywords. *Molecular Dynamics, Zeolites, Termodynamics Variables, programming in parallel, MPI, Beowulf Cluster.*

Molecular Dynamics (MD) is a computational technique used to determine the position of all particles within a thermodynamic system as a function of time. Particles' evolution in time is achieved by integrating Newton's equations of motion. Interacting forces among particles are computed via the calculation of short range (van der Waals and bonding energy) and long range (Coulomb) potentials. These calculations are the most demanding in terms of computing resources and represent about 90% of the time spent each cycle by a single processor machine. Here we present the results of developing a parallel MD code to study zeolites. This code was written in C++ and it uses the MPI Library to control messages passing. The code runs in a Beowulf cluster of computers. After checking the validity of the results by means of the comparison with the obtained ones with the code in series one proceeded to do the analysis of performance and efficiency of the paralelización across the evaluation of the model, the method, the communication, the efficiency and the cost of paralelización, obtaining very satisfactory information. In addition, we used a new methodology for the calculation of the electrostatic interactions in order to reduce execution times.

---

* thesis of degree

** Faculty of engineerings physicist mechanics.School of systems engineering and computer
Magister in computer engineering. Director. Cristian Blanco Tirado

INTRODUCCIÓN

El modelaje molecular ha sido utilizado ampliamente en las últimas décadas como una herramienta mas en el estudio de sistemas moleculares, a tal punto que se ha convertido en fuente de información imprescindible en la descripción de diversos procesos fisicoquímicos. Las técnicas principales de modelaje molecular son: mecánica cuántica y clásica, dinámica cuántica y clásica y los métodos de Monte Carlo.

En los últimos años, gracias al desarrollo en la ciencia de los computadores, se han hecho avances cuantitativos en la capacidad y versatilidad de estas herramientas. Es así que hoy en día, es fácil encontrar por lo menos una docena de aplicaciones de Dinámica Molecular (DM). Muchas de estas aplicaciones siguen los diseños totalmente funcionales desarrollados en los años 1960's, que consistían básicamente en hacer el algoritmo para ejecución en serie. Sin embargo últimamente ha tomado fuerza la idea de desarrollar estos programas en paralelo. Aunque existe por lo menos 4 de estas aplicaciones en Dinámica Molecular, no se encuentra en la literatura un programa robusto de DM que sea útil en el estudio de las zeolitas y que implemente adecuadamente los nuevos desarrollos en los diseños para el cálculo de las fuerzas intermoleculares de largo alcance.

El proyecto se centra en el desarrollo de la aplicación para lo cual se ha utilizado como código base un programa de Dinámica Molecular llamado Dizzy (escrito en Fortran 77) diseñado en serie. Este programa se convirtió en paralelo utilizando Programación Orientada a Objetos. La aplicación es robusta y altamente portable, sirviendo como base en el desarrollo de nuevas aplicaciones y en un mejor entendimiento de los procesos fisicoquímicos mediados por las zeolitas.

1. PLANTEAMIENTO DEL PROBLEMA

La Dinámica Molecular, es una técnica computacional ampliamente utilizada en diversas ramas de la ciencia. Su importancia radica básicamente en la posibilidad de determinar la trayectoria de las partículas de un sistema molecular mediante la solución de ecuaciones diferenciales sencillas, *i. e.* las ecuaciones de movimiento de Newton. Permitiendo reproducir los experimentos de laboratorio y brindando la libertad de variar los parámetros utilizados en la simulación.

La trayectoria de las partículas nos permite determinar estadísticamente otras propiedades termodinámicas, por ejemplo temperatura, energía interna, entre otras, definiendo así el estado termodinámico en el cual se encuentra el sistema molecular. Todo cálculo de las trayectorias moleculares involucra la determinación de las fuerzas de interacción intra e intermoleculares, las cuales podemos descomponer a su vez en fuerzas de corto y de largo alcance.

Las fuerzas de corto alcance se pueden calcular de forma directa y convergen rápidamente en función de la distancia $(1 / r^6)$, mientras que las de largo alcance convergen muy lentamente debido a que su dependencia con la distancia es proporcional a $(1 / r^2)$. Para solucionar el problema de convergencia en el cálculo de las fuerzas de largo alcance, se han desarrollado metodologías que descomponen el problema en sumas en espacio real y recíproco, sumas de Ewald; estas metodologías aunque convergen rápidamente son costosas computacionalmente debido a que exigen calculos intensivos, almacenamiento masivo, alta velocidad y alta capacidad de transferencia. Para solucionar este problema computacional se utilizan los sistemas de procesamiento en paralelo.

El cálculo computacional de las fuerzas de largo alcance es el factor limitante en el desarrollo de programas de Dinámica Molecular, debido a que el computador gasta la mayor parte del tiempo en su realización. Para eliminar el tiempo de cálculo de las fuerzas de largo alcance, y modelar sistemas moleculares más grandes, se han propuesto soluciones encaminadas a desarrollar metodologías que calculen la fuerza de forma eficiente y a desarrollar los códigos en paralelo para ser utilizados en un cluster que permita aumentar el rendimiento, la disponibilidad y la escalabilidad del modelamiento.

Sin embargo, no existe en la actualidad sistemas en paralelo de DM que utilicen estos dos mecanismos para mejorar la rapidez y eficiencia en el cálculo de trayectorias dinámicas. Para contribuir en la solución del problema, este trabajo desarrolla e implementa un programa de DM que conjuga el cálculo de fuerzas intra e intermoleculares en paralelo junto con la adaptación de metodologías eficientes en el cálculo de las fuerzas de largo alcance.

2. MARCO TEÓRICO

2.1 DINÁMICA MOLECULAR

En los últimos años muchos investigadores han intentado entender y desarrollar procesos físicos y químicos basados en materiales nanoporosos con la intención de aplicar ese conocimiento en catálisis, separación, detección y hasta liberación controlada de drogas[1]. Entre estas estructuras nanoporosas se encuentran las zeolitas, que se usan comúnmente en las industrias de separación y de petróleos[2]. Estos materiales son también utilizados como desecantes, intercambiadores iónicos y como detergentes ambientales amigables[3,4].

La versatilidad estructural y química que ofrecen las zeolitas sugiere que aún existen muchas aplicaciones industriales por descubrir. Por ejemplo, las zeolitas son candidatas a ser usadas en novedosos sistemas de enfriamiento y en nuevos convertidores catalíticos para el arranque de los automotores en frío[4-6]. Por otra parte, materiales nanoporosos con la misma estructura, se planean usar para la separación de biomoléculas y la elaboración de materiales optoelectrónicos con un confinamiento cuántico sustancial[7] . Las zeolitas, mediante cristalización controlada, se usan para la elaboración de nano películas con gran potencial en la separación de mezclas complejas, como las corrientes de xilenos en las refinerías de petroleo[8,9].

Estas novedosas aplicaciones muestran que la ciencia y tecnología de absorción, difusión y reacción en zeolitas, así como el estudio de la formación de cristales, siguen siendo áreas de intensa investigación y desarrollo[10].

Los usos industriales de las zeolitas son determinados por la estructura básica del material, tamaño de los poros, la carga del enrejado, la geometría de los canales y el área de composición de las superficies internas y externa[10]. En general, para los procesos de separación y catálisis, las moléculas orgánicas se deben absorber sobre la superficie de la zeolita para luego difundirse a través de los canales y permitir su separación o llegar hasta los sitios catalíticamente activos para realizar la reacción deseada.

Mediante estudios experimentales y teóricos se han logrado describir de manera exitosa los procesos de difusión en zeolitas y de igual manera se ha investigado sobre la relación entre la geometría y la energía de absorción de moléculas orgánicas dentro de los cristales[11] . Sin embargo, en muchas otras aplicaciones los fundamentos físicos y químicos relacionados con la reactividad, la difusión y la absorción en zeolitas son desconocidos. Esto es debido en parte a la dificultad de observar experimentalmente los procesos mediados por las zeolitas.

En años recientes se han producido considerables avances en el estudio de las zeolitas. Varias técnicas computacionales han sido implementadas exitosamente en el análisis de la estructura, y las propiedades fisicoquímicas de estos materiales, tales como absorción, difusión, acidez, y catálisis. Entre estas técnicas se encuentra la Dinámica Molecular, la cual ha sido ampliamente utilizada en el estudio de la difusión y la absorción en zeolitas[12].

Las simulaciones con Dinámica Molecular han tenido un impacto considerable en los últimos años en el estudio de las zeolitas. Estas simulaciones han sido de vital importancia en el entendimiento de experimentos de difusión de forma cualitativa y cuantitativa.

La técnica de DM se basa en la segunda ley de Newton $F_i = m_i * a_i$, donde F_i es la fuerza ejercida sobre cada átomo i, m_i es la masa y a_i la aceleración de cada partícula i. Gracias a que la energía potencial es una función local en un sistema molecular, la fuerza ejercida por todas las partículas del sistema sobre una en particular se puede determinar a partir de la energía potencial mediante la expresión: $F_i = -\frac{dV_i}{dr_i}$.
Si se conoce la fuerza en cada átomo se puede determinar la aceleración de las partículas en el sistema y por ende la nueva posición. Cuando se integran las ecuaciones de movimiento se obtienen las trayectorias de cada partícula en términos de la variación de las posiciones, velocidades y aceleración con el tiempo. Al conocer las trayectorias de cada átomo el estado termodinámico se puede predecir en el futuro o pasado. Mediante la trayectoria se pueden predecir la temperatura, la presión, energía total y otras variables termodinámicas de interés[12,16].

La Dinámica Molecular también permite determinar cantidades dinámicas como las funciones de autocorrelación de la velocidad (utilizadas para encontrar frecuencias vibracionales a las cuales resuenan los sistemas en estudio). Las constantes de difusión y los factores dinámicos de dispersión[12]. Para las simulaciones de sistemas zeolíticos se utilizan dos tipos de funciones disponibles para hacer cálculos, los de enlace de valencia y los iónicos [13]. Mientras que los potenciales de enlace de valencia consideran el sistema como una colección de interacciones de corto alcance entre dos , tres y cuatro cuerpos (ver ilustración 1) , los potenciales iónicos representan el sistema como un conjunto de cargas que interactúan vía fuerzas de corto y largo alcance[14]. Las fuerzas intermoleculares entre las zeolitas y los adsorbatos se describen mediante interacciones de Lennard-Jones y términos Coulómbicos, los cuales se obtienen a través de cálculos mecanocuánticos o de datos espectroscópicos, al igual de la descripción de los enlaces, ángulos y torsiones moleculares. En la siguiente ilustración se

observan diferentes tipos de interacciones que existen entre las partículas de un sistema molecular.

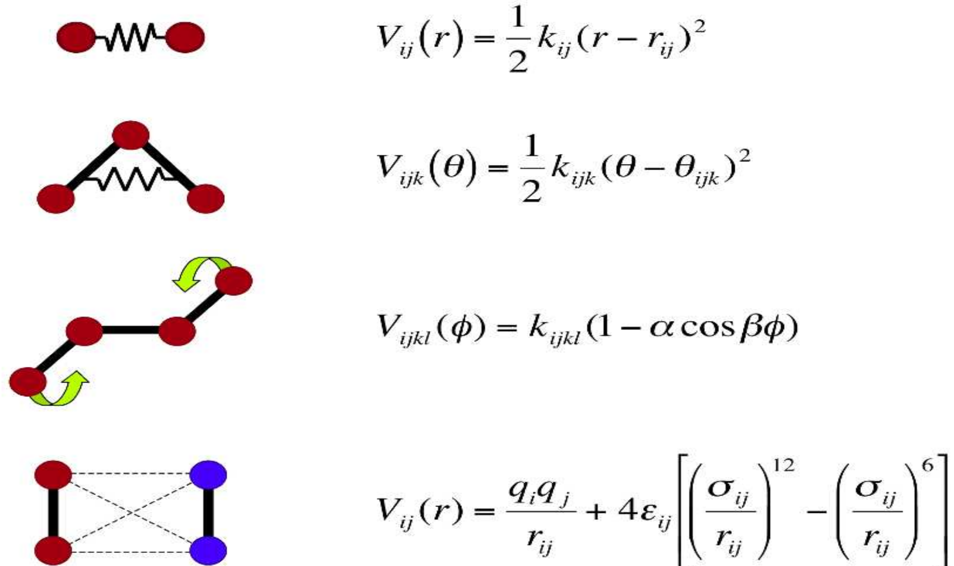


Ilustración1. Potenciales de corto y largo alcance
Fuente: Modeling Equilibrium and nonequilibrium dinamics in Zeolites

Siguiendo un orden descendente, la primera figura de la ilustración representa un enlace entre dos atomos y su fórmula matemática basada en la distancia. La segunda figura simboliza un enlace formado por tres átomos, cuya fórmula utiliza el ángulo existente entre ellos. La tercera Figura representa el ángulo de torsión entre cuatro átomos acompañado de su ecuación matemática. La cuarta figura simboliza el potencial entre dos moleculas y sus dos átomos, en donde la fórmula matemática incluye las cárgas atómicas.

Se observa matemáticamente que al aumentar el número de partículas en el sistema, el cálculo de los potenciales y por ende de las fuerzas de iteración se hace más costosos computacionalmente, debido en principio, a que las interacciones entre las partículas se incrementan en forma factorial.

El potencial total de interacción se puede calcular mediante la siguiente sumatoria:

$$V_T = \sum_i \sum_j k_{ij} (r - r_{ij})^2 + \sum_i \sum_j \sum_k k_{ijk} (\theta - \theta_{ij})^2 + \sum_i \sum_j \sum_k \sum_l k_{ijkl} (1 - \alpha \cos \beta\phi) + \sum_i \sum_j 4\varepsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \sum_i \sum_j \frac{q_i q_j}{r_{ij}}$$

Donde el primer término esta relacionado con todos las partículas unidas entre si, el segundo involucra tríos de partículas unidas mediante enlaces, la tercera

expresión implica que hay cuatro partículas unidas, la cuarta expresión representa las interacciones no enlazantes, es decir, las interacciones entre partículas que no pertenecen a la misma especie química. Todas estas interacciones son de corto alcance, convergen muy rápidamente y, como su nombre lo indica, a distancias cortas entre partículas. Finalmente tenemos las interacciones electrostáticas, las cuales por naturaleza son de largo alcance, *i.e.* en la sumatoria se debe considerar la interacción de cada partícula con las demás. Debido a que este potencial es inversamente proporcional a la distancia, esta sumatoria no converge. Para solventar este problema, históricamente se han desarrollado una serie de algoritmos para el cálculo de las energías electrostáticas en materiales periódicos tridimensionales. Entre estos algoritmos tenemos el método de Ewald,[15,17] el cual, a pesar de haber sido utilizado ampliamente, es costoso computacionalmente debido a su implementación (O(N2)). Adicionalmente, el método de Ewald converge muy pobremente para sistemas bidimensionales. La implementación del metodo de Ewald aprovecha la periodicidad de los sistemas moleculares para hacer un cambio en los límites de la sumatoria, de manera que la ecuación de Coulomb se convierte en:

$$E^{tot} = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{n=0}^{\infty}{}' \frac{q_i q_j}{\left|\vec{r}_{ij} + \vec{n} L\right|}\left[erfc\left(\alpha\left|\vec{r}_{ij} + \vec{n} L\right|\right) + erf\left(\alpha\left|\vec{r}_{ij} + \vec{n} L\right|\right)\right]$$

La cual converge más rápidamente, aunque es necesario hacer una sumatoria adicional sobre vectores recíprocos del enrejado. Esta ecuación se convierte en la ecuación de trabajo para el cálculo de la energía electrostática en DM.

Recientemente, Wolf y colaboradores desarrollaron un algoritmo que promete mejorar sustancialmente tanto la interpretación física como la eficiencia de los cálculos electrostáticos.[18.]

Es precisamente el cálculo de las fuerzas de largo alcance el factor limitante en el desarrollo de programas de DM, debido a que el computador gasta la mayor parte del tiempo realizando el cálculo de estas fuerzas. Con la finalidad de la disminución del tiempo de cálculo en las fuerzas de largo alcance, y para modelar sistemas moleculares más grandes, se han propuesto soluciones encaminadas a: desarrollar metodologías que calculen la fuerza de forma eficiente o a desarrollar los códigos en paralelo.[18,19] .

La mayoría de programas de DM han sido desarrollados por investigadores en física y química para cumplir funciones específicas o simular sistemas particulares; estos han sido desarrollados a través de programación serial, lo cual requiere gran capacidad de computo a la par de gran cantidad de memoria para realizar simulaciones de sistemas moleculares de mas de 100 átomos[15,16].

Existen algunas implementaciones de programas de DM en procesamiento paralelo como NAMD, GROMACS, estos programas aunque han demostrado estar bien desarrollados, han sido básicamente creados para estudiar sistemas moleculares biológicos [19] .

La ventaja de desarrollar e implementar algoritmos de DM en paralelo radica principalmente en la posibilidad de simular sistemas moleculares cada vez más grandes que brinden la estadística necesaria para hacer comparaciones significativas con los datos experimentales.

2.2 PARALELIZACIÓN

El objetivo principal de la programación en paralelo es disminuir el tiempo de ejecución de una aplicación mediante la utilización de métodos de programación que permitan la utilización de sistemas físicos de cómputo que posean varios procesadores [20].

Existen básicamente dos paradigmas en la ejecución de programas en paralelo, los de memoria compartida y los de memoria distribuida. Los primeros funcionan con un solo bloque de memoria que es utilizado por todos los procesadores en la misma tarjeta o máquina. Los segundos consisten en un conjunto de computadores intercomunicados que permiten la transferencia de información de una máquina a otra en forma eficiente. De manera que cada procesador hace uso autónomo de su memoria. En este segundo grupo de sistemas se encuentran los *clusters del tipo de Beowulf* [20,21].

Las herramientas utilizadas para desarrollar programas en paralelo dependen básicamente del tipo de máquinas. En el caso de los sistemas de memoria compartida se utilizan compiladores que incluyen directrices para programación con multiprocesadores. Tal es el caso de las librerías OpenMP o sencillamente MPI . Por otra parte, los sistemas de memoria distribuida, del tipo Beowulf, utilizan una serie de bibliotecas que se ensamblan en los compiladores tradicionales, e. g. Fortran, C++, para facilitar la comunicación entre los procesadores, de esta manera se facilitan las comunicaciones entre procesadores, la administración y control de las tareas que cada procesador realiza durante la ejecución del programa [22].

Los algoritmos más conocidos que cumplen esta función son Message Passing Interfase (MPI) y Parallel Virtual Machine (PVM). Como se mencionó anteriormente, estas interfases han sido ensambladas exitosamente en los compiladores más populares. De las dos, MPI es tal vez la interfaz mas utilizada en la actualidad para desarrollos de programas en paralelo. Estas bibliotecas llevan a cabo el proceso de paralelización en tres formas distintas, las cuales se utilizan dependiendo del análisis del problema y del programa a desarrollar. Estos tipos son:

1. Múltiple Secuencia de Instrucciones una Secuencia de Datos (MISD): Transmite una secuencia de datos a un conjunto de procesadores, cada uno de los cuales ejecuta una secuencia de instrucciones diferente.

2. Instrucción Simple Múltiples Datos (SIMD): en la que cada elemento de proceso tiene una memoria asociada, de forma que cada instrucción es ejecutada por cada procesador, con un conjunto de datos diferentes. Los procesadores matriciales y vectoriales pertenecen a esta categoría. Su representación gráfica es:

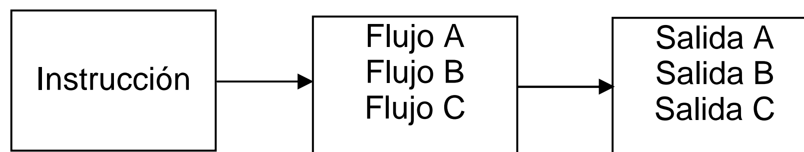


Ilustración 2. SIMD.
Fuente:Introducción a la programación paralela

Múltiples Instrucciones-Múltiples Datos (MIMD): donde un conjunto de procesadores (clusters) ejecutan simultáneamente secuencias de instrucciones diferentes con conjuntos de datos diferentes. Su función se representa en la siguiente ilustración:

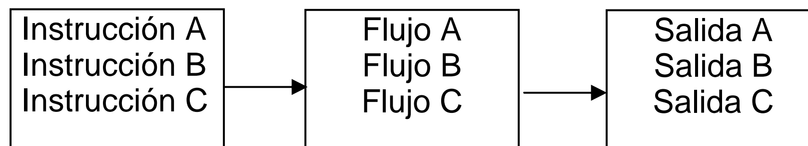


Ilustración 3. MIMD
Fuente:Introducción a la programación paralela

Para obtener una buena respuesta en la computación paralela hay que tener en cuenta la optimización de las tareas a través de los tiempos de cómputo, tiempos libre y tiempo de computación [24,25]. En este estudio en particular se desarrolla e implementar un programa de Dinámica Molecular para el estudio de la superficie

de los materiales del tipo Zeolitas[26,27] . El diagrama de flujo de los procesos que requieren de mayor tiempo de cómputo se describe a continuación:

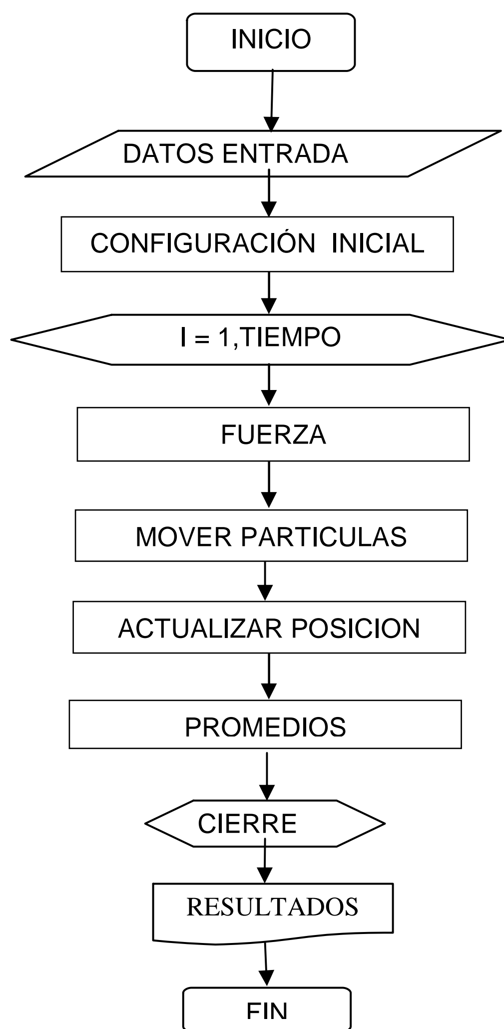


Ilustración 4. Diagrama de Flujo

Debido a la complejidad del sistema molecular, se ha determinado que el programa de DM gasta un 95% del tiempo en el cálculo de la fuerza (de corto y largo alcance).

2.3 TIPO DE TRABAJO

El trabajo de investigación se encuentra enmarcado dentro de un estudio descriptivo y de investigación tecnológica aplicada.

A nivel descriptivo se estudiaron las características de las paralelización, como su arquitectura y tipos, también se soporta en los principios fundamentales de los campos de fuerza y su gama de aplicaciones en el mundo científico.

El trabajo de investigación tecnológica aplicada corresponde al diseño, desarrollo y pruebas con una retroalimentación de los resultados generados con pruebas químicas del laboratorio. Se realizará el código en paralelo utilizando el modelo MIMD que maximiza el trabajo por medio de clusters, a través de la descomposición del dominio en sus procesadores y el uso de la memoria compartida.

3. CREACIÓN DE UN PROGRAMA EN PARALELO

En la creación de un programa en paralelo se requiere tener claro los siguientes conceptos:

Tarea: Conjunto de instrucciones para realizar una actividad definida.
Proceso: Conjunto de tareas.
Nodo: Unidad de conexión de red.
Computación distribuida: Modelo de tratamiento de problemas de computación masiva en la cual se utiliza una infraestructura compuesta por diferentes computadores y el código se "distribuye" entre ellos.
Escalabilidad: Es la capacidad que tiene un sistema de adaptarse a un número de usuarios cambiantes, sin perder calidad en los servicios.
Clusters: Sistema de procesamiento paralelo o distribuido que consiste en una colección interconectada de máquinas secuenciales que trabajan cooperativamente como un único recurso computacional integrado. Existen cluster de Alta disponibilidad (OpenMosix) y de alto desempeño (*Beawulf*).
Balanceo de carga: se refiere a la distribución de las tareas en forma tal que se asegura más eficientemente la ejecución en paralelo.

La mayoría de los autores coinciden en la creación de programas en paralelo se utilizan cuatro pasos fundamentales cuya ilustración es:

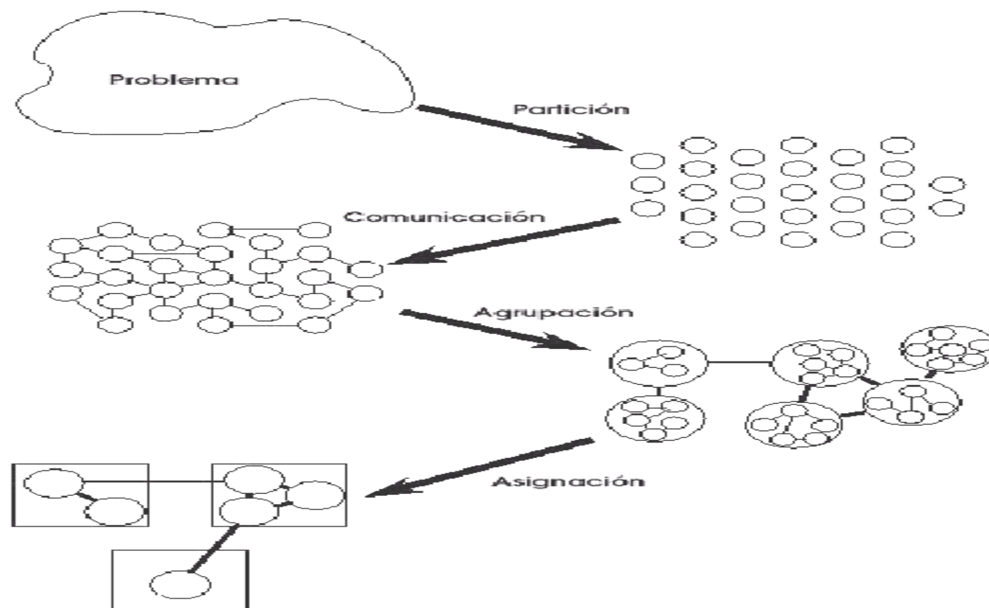


Ilustración 5. Pasos en la creación de un programa en paralelo

Fuente: Proyecto cumulus

En la primera etapa se subdivide el problema en un gran número de tareas en orden de rendimiento, buscando que la granularidad sea fina y sin incremento del proceso de comunicación. La granularidad es la propiedad que garantiza la descomposición de un proceso en tareas simples. Se analiza el problema y se observa si una solución paralela es buena o no por medio de la identificación de variables, su relación y los ciclos utilizados en ella, la subrutinas de DM paralelizadas que obtuvieron buenos resultados son las del cálculo de fuerzas y energía, aplicando la metodologías de Ewald y Wolf, representados a través del archivo de salida en el ítem descrito como *time for real space calculation*. Se paralelizan otras rutinas pero los resultados no eran satisfactorios, por este motivo se descartan.

En la bibliografía se explican 2 técnicas para realizar la descomposición del problema:

La descomposición del problema por descomposición del dominio, donde los datos son divididos en secciones del mismo tamaño que son organizadas para diferentes procesadores; cada procesador realiza independientemente su tarea y posteriormente comunica sus resultados al procesador de control. Ver siguiente figura[26,29].

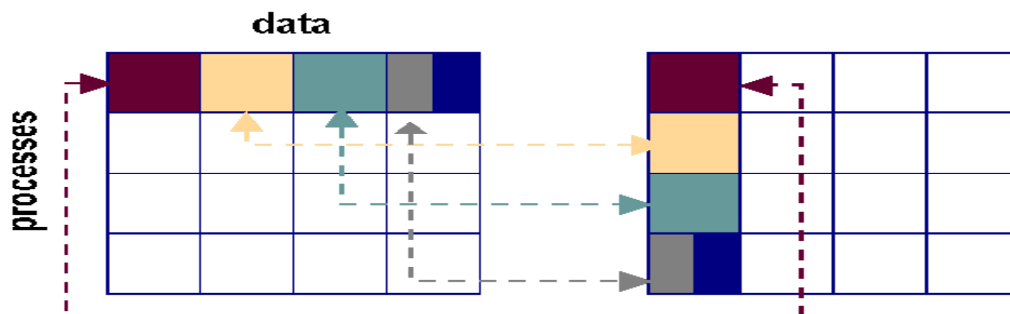


Ilustración 6. Descomposición del Dominio
Fuente:MPI course Introduction

También se puede usar la descomposición funcional, en el que un procesador maestro asigna los trabajos de acuerdo a su tamaño y orden a los procesadores esclavos.

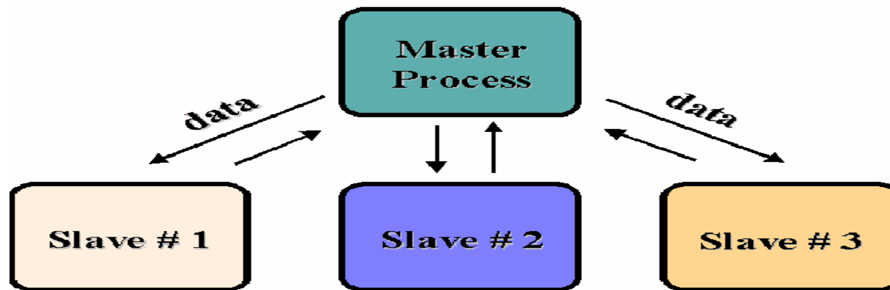


Ilustración 7. Descomposición funcional
Fuente:MPI course Introduction

En cuanto a la memoria se puede utilizar las arquitecturas de distribución de memoria para que cada nodo tenga un acceso rápido a su memoria local y a otros nodos por medio de los enlaces de red, trabajando unidos en la solución de un problema. Además se puede usar la división de memoria, permitiendo que varios procesadores (entre 2 a 16) ejecuten tareas simultáneamente, limitados por el ancho de banda del bus de memoria [29, 30].

En el sistema Dizzy se implementó la descomposición funcional en donde se dividieron los cálculos en tareas separadas exitosamente, la granularidad es fina.

3.2 COMUNICACIÓN:

Los cálculos ejecutados por una tarea requerirán datos asociados a otras. Por lo tanto durante la computación, los datos deben ser transferidos o compartidos entre tareas y esto se diseña en la fase de comunicación.

En nuestro ambiente de memoria distribuida cada tarea tiene una identificación única y las tareas interactúan enviando y recibiendo mensajes hacia y desde tareas especificas, todo esto es permitido por la codificación utilizando las funciones de la librería MPI. Durante el proyecto se codificaron diferentes modelos de mensajes en MPI y seleccionó la Mensajería asíncrona con Bloqueo MPI_Send y MPI_Recv por el nivel de seguridad y menores tiempos de respuesta.

Existen cuatro tipos de comunicación:

1. Local/Global: En la comunicación local, cada tarea se comunica con un pequeño grupo de otras tareas vecinas. En cambio la comunicación global, cada tarea se comunica con otras tareas.
2. Estructurada/No Estructurada: La comunicación estructurada toma una forma regular, de árbol o malla. Por el contrario la No Estructurada forman

grafos no arbitrarios.

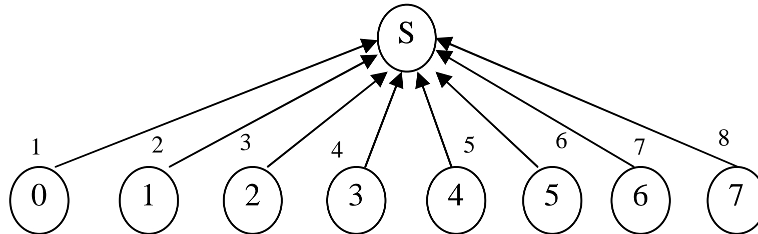


Ilustración 8. Comunicación Global en sumatoria de 8 números con Log(N) pasos

Fuente: Proyecto Cumulus

3. Estática/Dinámica: En la comunicación estática, la identidad de los pares de comunicación no cambian con el tiempo. En cambio, en la Dinámica la identidad puede ser cambiada en tiempo de ejecución por medio de cálculos.
4. Sincrónica/Asincrónica: En la comunicación sincrónica los productores (proporcionan la información) y los clientes(requieren información) cooperan en la transferencia de datos. Entre tanto en la Asincrónica se presenta que un clientes obtenga información sin la cooperación del productor [31].

Según las características anteriormente explicadas las sumatorias y demás procesos paralelizados utilizan la comunicación Global, Estructurada, Estática y Asincrónica.

Se aplicaron estrategias para solucionar problemas recursivamente, la estrategia de *divide y conquista* es la que mejor resultado genera, su algoritmo es el siguiente:

```
Procedimiento divide_y_conquista
Inicio
   particione el problema en subproblemas L y R
   resuelva el problema L y subdivida y conquista
   resuelva el problema R y subdivida y conquista
   combine la solución de los problemas L y R
Fin
```

El anterior algoritmo requiere de tres pasos para cada nivel de división. Primero se divide el problema en subproblemas. Segundo se resuelven los subproblemas de la manera más sencilla (conquista). Tercero se combina la solución de los subproblemas, resolviendo así el problema original.

3.3 ORQUESTACIÓN, SINCRONIZACIÓN, AGRUPACIÓN O AGLOMERACIÓN:

En esta fase se realiza una revisión de las decisiones tomadas en las 2 fases anteriores con el objetivo de obtener un algoritmo paralelo eficiente. Las tareas y las estructuras de comunicación se evalúan con respecto a los requerimientos de ejecución y los costos de implementación. Si es necesario, las tareas se redefinen para combinarse o agruparse en tareas mas grandes para optimizar la ejecución y el desarrollo ver ilustración 9.

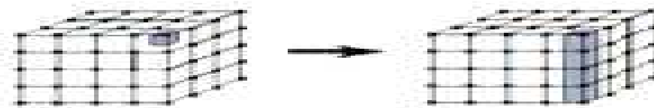


Ilustración 9. Agrupación
Fuente: Proyecto Cumulus

Además se evalúa si las tareas frecuentemente se bloquean en la espera de datos remotos siendo así necesaria la asignación de varias tareas a un procesador. "El número optimo de tareas es determinado por una combinación de modelamiento analítico y estudios empíricos". En esta fase se reducen costos de comunicación y costos de ingeniería del software, además se mejora la flexibilidad.

3.4 LOCALIZACIÓN, MAPEO O ASIGNACIÓN:

En esta fase se especifica en que procesador se ejecutara cada tarea minimizando los costos de comunicación y sincronización. El principal objetivo de esta fase, es diseñar algoritmos que minimicen el tiempo de ejecución total. Actualmente no se han documentado mecanismos de mapeo siendo un problema interesante de estudio[30-32].

4. TOPOLOGIA DE LA RED PARALELA

La topología se refiere a la forma como se enlazan los procesadores en un computador en paralelo, su importancia radica en que permite minimizar el acceso de memoria. Los criterios para evaluar el diseño son:

- ✓ Diametro de la red. Es la mayor distancia entre dos nodos, entre menor se el diametro, menor es el tiempo de comunicación.
- ✓ La conectividad de la red. Permite reducir el tiempo de comunicación al poder viajar por caminos alternos, evitando o reduciendo la congestión entre los node de la red.
- ✓ La flexibilidad. Permite ejecutar una amplia variedad de algoritmos, en donde la interconexión puede variar durante la ejecución de los programas.
- ✓ El retrazo en la comunicación. Requerido por algunas tareas en muchos algoritmos como computación de productos internos,multiplicación entre una matriz y un vector, etc.

4.1 TIPOS DE TOPOLOGIAS

Arreglo de procesadores lineal. Permite comunicación bidireccionales entre cada uno de los procesadores que forman un elemento del vector.

Anillo. Cada nodo se conecta al siguiente y el último al primero formando un anillo.

Arbol. Existe un nodo llamado raíz, en donde cada nodo *i* se conecta por un solo camino a otro nodo en grupos de dos hasta llegar al nodo raíz.

Matriz. Arreglos de procesadores conectados en *d* dimensiones.

Hipercubo. Los nodos son agrupados en dos cubos desplazados en un cuarto eje dimensional, utilizando 32 conexiones entre ellos [33].

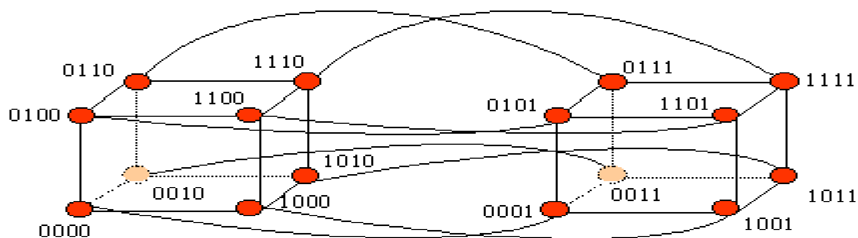


Ilustración 10. Hipercubo de nodos
Fuente: Parallel and Distributed Computation Numerical Methods

5. MPI

La interfaz de paso de mensajes MPI es un estandar API, altamente portable utilizado principalmente en sistemas SIMD. MPI puede trabajar con lenguajes de programación como FORTRAN y lenguaje C++. En C++ se compila MPI utilizando el siguiente comando: *mpiCC programa.c -o ejecutable*

Para ejecutar el compilado se utiliza la siguiente linea de comando:
mpirun -np numero_procesadores ejecutable

5.1 FUNCIONES

MPI utiliza192 funciones, de las cuales las utilizadas en el proyecto son las siguientes:

MPI::Init(argc,argv);
Es la primera llamada de cada uno de los procesos, esta función establece el entorno de programación que solamente se realiza una vez en la aplicación.

MPI::Finalize();
Termina la ejecución en MPI, al liberar los recursos utilizados.

MPI::COMM_WORLD.Get_size();
Determina el numero de procesos que participan en la aplicación.

MPI::COMM_WORLD.Get_rank();
Permite que cada proceso averigüe su dirección (identificador) dentro de la colección de procesos que componen la aplicación.

MPI::COMM_WORLD.Send(buf, count, datatype, dest, tag);
Envio de un mensaje con bloqueo a una determinada dirección.

MPI::COMM_WORLD.Recv(buf, count, datatype, source, tag);
Recepción de un mensaje de una o cualquier fuente con bloqueo.

MPI::COMM_WORLD.Isend(buf, count, datatype, dest, tag, request);
Envio de un mensaje sin bloqueo con un objeto request para determinar si la operación solicitada ha terminado o no.

MPI::COMM_WORLD.Irecv(buf, count, datatype, source, tag, request);
Recepción de un mensaje sin bloqueo con el objeto request para determinar si la operación solicitada ha terminado o no.

MPI::Wait();
Evalúa y espera que la operación se complete.

MPI::Test();
Devuelve una bandera(flag) indicando si la operación se ha completado.

LLAMADAS COLECTIVAS

MPI::COMM_WORLD.Barrier();
Finaliza los procesos asegurándose de que todos hayan terminado antes de dar comienzo al siguiente.

MPI::COMM_WORLD.Bcast(void* buffer, int count, MPI_Datatype datatype ,int root);
Sirve para que el proceso raíz envíe un mensaje (variable, vector, matriz) a todos los miembros del comunicador.

MPI::COMM_WORLD.Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root);
Devuelve la información de los nodos esclavos al vector del nodo raiz.

MPI::COMM_WORLD.Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root);
Distribuye un buffer de datos en partes y los entrega a un grupo de procesos(esparcir).

MPI::COMM_WORLD.Reduce(void* sendbuf, void * recvbuf, int count, MPI::Datatype datatype, MPI_OP op, int root);
Operación realizada de forma cooperativa entre todos los procesos de un comunicador, de tal forma que se obtiene un resultado final que se almacena en el proceso raíz. Las operaciones a realizar pueden ser sumatorias, máximos, minimos, entre otras.

5.2 EJEMPLOS

Ejemplo 1.

Calcular el mínimo de un arreglo de datos, empleando el paradigma maestro-esclavo (descomposición funcional). En el maestro se crea el arreglo de datos dinámicamente y a cada proceso hijo se le asigna el cálculo del valor mínimo de una porción del arreglo.

```
#include "stdafx.h"
#include "mpi.h"
```

```
#include <stdio.h>

float min(float *x, int n)
{
 int i;
 float min=x[0];
 for(i=1;i<n;i++)
   if(x[i]<min)
      min=x[i];
 return min;
}

int main(int argc, char* argv[])
{
  MPI_Status  status;
  int numProcs;
  int myRank;
  float *x;
  float  y[1000];
  float  mini,z;
  int    I,n;
   MPI_Init(&argc,&argv); //Inicializo MPI
   MPI_Comm_size(  MPI_COMM_WORLD,  &numProcs);//  Determino  el  número  de
procesos
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank);// Determino el número de proceso
con el que estoy ejecutando
   if(myRank==0)
   {    n = numProcs*1000;
        x = new float[numProcs*1000];     // Creo el vector
      for(i=0;i<n;i++)
        x[i]=(float)i+10;
      for(i=1;i<numProcs;i++)
        MPI_Send(&x[(i-1)*1000],1000,MPI_FLOAT,i,1,MPI_COMM_WORLD);     // A cada
uno de los procesos se le asigna el trabajo a realizar
       mini=1e4;
       for(i=1;i<numProcs;i++)
      {
MPI_Recv(&z,1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,
&status);         //Recibe el valor mínimo de cada proceso y calcula el min global
         if(z<mini)
           mini=z;
        }
     printf("El valor minimo es:%f\n",mini);
   }
else{
MPI_Recv(&y[0],1000,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_W
ORLD,&status);      // Recibe el arreglo de datos
```

```
        z=min(y,1000);
        MPI_Send(&y,1,MPI_FLOAT,0,1,MPI_COMM_WORLD);       // Regresa el resultado
   }
   MPI_Finalize(); //Finalizo MPI
  return 0;
}
```

Ejemplo 2.

Determinar el valor de pi empleando la integral de 4/(1+x*x) en el intervalo -1/2 a +1/2. El método es sencillo, la integral se aproxima por suma de intervalos (Riemann). Se pide al usuario el número de intervalos y con base al número de procesos se divide la tarea.

```
#include  "stdafx.h"
#include  "mpi.h"
#include  <stdio.h>
#include  "math.h"
#ifndef M_PI
  #define M_PI  3.14159265358979323846
#endif

int main(int argc, char* argv[])
{
  int numProcs;
  int myRank;
  int i,n;
  double sum,h,x;
  double pi,mypi;
   MPI_Init(&argc,&argv);
   MPI_Comm_size( MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank( MPI_COMM_WORLD, &myRank);
   if(myRank==0){
     printf("Numero de intervalos :");
     fflush(NULL);
    scanf("%d",&n);
   }

   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);// Comparte la información con todos
los procesos
   h   = 1.0 / (double) n;//    Determina el tamaño del paso global
   sum = 0.0;
// Calcula la integral, por ejemplo numProcs=3, se tienen 3 casos
// de ejecucion en paralelo
// a) for (i = 1; i <= n; i +=3)
// b) for (i = 2; i <= n; i +=3)
// c) for (i = 3; i <= n; i +=3)
   for (i = myRank + 1; i <= n; i += numProcs)
```

```
  {
     x = h * ((double)i - 0.5);
     sum += 4.0 / (1.0 + x*x);
  }
// Ajusto la integral
  mypi = h * sum;
//  A todos los procesos se les colecta el valor que tienen en mypi, cada valor es sumado y
almacenado en pi.
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
  if (myRank == 0){
    printf("\n");
     printf("PI es aproximadamente %.18f\n",pi);
     printf("La diferencia | PI-PI_M | es:  %e\n",fabs(pi-M_PI));
   }
  MPI_Finalize();
 return 0;
}
```

Ejemplo 3.

Llenar la matriz A de tamaño *mxn* y B de *nxl*, y realizar la multiplicación entre ellas utilizando el método de descomposición funcional. El resultado debe ser almacenado en la matriz C.

```
/** AUTHOR: Ros Leibensperger / Blaise Barney. Converted to MPI: George L. Gusciora */
#include "mpi.h"
#include <stdio.h>
#define NRA 62                /* número de filas de la matriz A */
#define NCA 15                /* número de columnas de la matriz A */
#define NCB 7                 /* numero de columnas de la matriz B */
#define MASTER 0              /* identificador de la tarea */
#define FROM_MASTER 1         /* tipo de mensaje */
#define FROM_WORKER 2         /*tipo de mensaje */

int main(int argc,char * argv[ ])
{
int numtasks, taskid, numworkers, source,  dest, mtype, rows;
int averow, extra, offset, i, j, k, rc;
double  a[NRA][NCA],  b[NCA][NCB], c[NRA][NCB];          /* Matriz resultante C */
MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  numworkers = numtasks-1;
/*************************** Tareas procesador raiz ***********************************/
  if (taskid == MASTER)
  {
```

```
    printf("Numero de tareas = %d\n",numworkers);
    /* llenado de la matriz A y B */
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        a[i][j]= i+j;
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        b[i][j]= i*j;

    /* envio de la matriz de datos a los procesadores esclavos */
    averow = NRA/numworkers;
    extra = NRA%numworkers;
    offset = 0;
    mtype = FROM_MASTER;
    for (dest=1; dest<=numworkers; dest++)
    {
      printf("   enviando %d columnas a la tarea %d\n",rows,dest);
      MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
     MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
      MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
              MPI_COMM_WORLD);
      MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
      offset = offset + rows;
    }

    /* Recepción de los resultados de las tareas asignadas a los procesadores esclavos*/
    mtype = FROM_WORKER;
    for (i=1; i<=numworkers; i++)
    {
      source = i;
      MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
    }
    printf("Matriz resultante C\n");
    for (i=0; i<NRA; i++)
    {
      printf("\n");        for (j=0; j<NCB; j++)
        printf("%6.2f   ", c[i][j]);
    }
    printf ("\n");
  }
/************************** tareas de los procesadores esclavos*****************************/
  else if (taskid > MASTER)
  {
    mtype = FROM_MASTER;
```

```
/* Recepción de las tareas asignadas por el procesador Master*/
     MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,
&status);
     MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,
&status);
/* Calculo de la multiplicación de matrices */
     for (k=0; k<NCB; k++)
       for (i=0; i<rows; i++)
       {
         c[i][k] = 0.0;
         for (j=0; j<NCA; j++)
           c[i][k] = c[i][k] + a[i][j] * b[j][k];
       }
   mtype = FROM_WORKER;
/*Envio de resultados de la tarea asignada */
     MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  }
  MPI_Finalize();
return 0;
}
```

La multiplicación de las anteriores matrices requiere de la evaluación de *ml* productos internoe en *n* vectores, se puede solucionar en un tiempo de [log *n*] +1 usando *nml* procesadores. Para el caso en el que $m=n=l$ se utiliza n^3 procesadores. Si re requiere multiplicar una matriz con un vector el tiempo de computo es O(*log n*) utilizando n^3 procesadores.

5.3 APLICACIONES

Aplicación 1.

Programa principal del sistema Dizzy en donde inicia y termina la paralelización.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <time.h>
#include "common_dizzy.h"  //variables globales del sistema
#include "dizzy.h" //funciones del sistema
int main(int argc, char* argv[])
{
  MPI_Init(&argc,&argv);
  MPI_Comm_size( MPI_COMM_WORLD, &num_procs);
```

```
   MPI_Comm_rank( MPI_COMM_WORLD, &my_id);
   printf("proceso iniciado : %d",my_id);
   time(&start);
           initdiz ();
           rd_dump ();
           rd_pars ();
           params_diz ();
      mains();
    printf("TERMINACION  %d:",my_id);
    MPI_Finalize();
return 0;
}
```

Aplicación 2.

Calcular las velocidades iniciales en forma aleatoria aplicando la descomposición funcional, en donde el procesador maestro asigna el trabajo a los procesadores esclavos que retornan la tarea calculada.

```
void velocidades_iniciales()
{double Px,Py,Pz,CHI,x,y,z,sumax,sumay,sumaz; //variables locales
#define GAS 8.3145
srand(time(NULL)); //semilla
sumax=0;sumay=0;sumaz=0;
Vx = new double[dimension]; //Asignación dinámica del tamaño del vector
Vy = new double[dimension];//segun el número de procesos y partículas
Vz = new double[dimension];
for (i=0; i<totpart;i++)     //generación de la campana de gauss para iniciar velocidades
{         for (j=1; j<=12;j++)
          {  x= ((double)rand()/(double)(RAND_MAX )) ;
            sumax= sumax + x;
          y= ((double)rand()/(double)(RAND_MAX));
          sumay= sumay + y;
          z= ((double)rand()/(double)(RAND_MAX));
          sumaz= sumaz + z;
          }
  sumax=sumax-6;sumay=sumay-6;
  sumaz=sumaz-6;
  Vx[i]=sumax;Vy[i]=sumay;
  Vz[i]=sumaz;
}
Px=0; Py=0;Pz=0;
for (i=0;i<totpart;i++)
{         Px=Px+Vx[i]*masas[i];
          Py=Py+Vy[i]*masas[i];
          Pz=Pz+Vz[i]*masas[i];
}
```

```
  Px=Px/totpart;  Py=Py/totpart;  Pz=Pz/totpart;
KE=0;
if (my_id==0)       // hilo del procesador master
{//determino el trabajo a realizar a los procesadores slaves
   for (i=1;i<num_procs;i++)
  {  MPI::COMM_WORLD.Send(&Vx[(i-1)*asigno_part],asigno_part,MPI::DOUBLE,i,1);
     MPI::COMM_WORLD.Send(&Vy[(i-1)*asigno_part],asigno_part,MPI::DOUBLE,i,1);
     MPI::COMM_WORLD.Send(&Vz[(i-1)*asigno_part],asigno_part,MPI::DOUBLE,i,1);
   } //a cada proceso slave le detemino el numero de posiciones que va a utilizar para los
calculos
     MPI::COMM_WORLD.Recv(&CHI,1,MPI::DOUBLE,MPI::ANY_SOURCE,MPI::ANY_
TAG) ;
     // recibe la informacion de los procesadores slaves
          for (k=0;k<tot_part;k++)
          {         Vx[k]=Vx[k]*CHI;          Vy[k]=Vy[k]*CHI;          Vz[k]=Vz[k]*CHI;
                     v2 = pow(Vx[k],2) + pow(Vy[k],2)+ pow(Vz[k],2);
                    KE= KE + 0.5*10000000*v2*masas[k];
          }//calculo de energia cinetica y velocidades en cada eje
          CURRTEMP=(2*KE)/(3*totpart*GAS);   //Temperatura
          std::cout<<"TEMPERATURA: "<<CURRTEMP<<"\n";
 }
else
{         //hilos slaves
MPI::COMM_WORLD.Recv(&Vx[0],asigno_part,MPI::DOUBLE,MPI::ANY_SOURCE,MPI::
ANY_TAG);
MPI::COMM_WORLD.Recv(&Vy[0],asigno_part,MPI::DOUBLE,MPI::ANY_SOURCE,MPI::
ANY_TAG);
MPI::COMM_WORLD.Recv(&Vz[0],asigno_part,MPI::DOUBLE,MPI::ANY_SOURCE,MPI::
ANY_TAG); //cada procesador slave recibe su tarea a realizar
  for (k=0;k<asigno_part;k++)
  { Vx[k]=Vx[k]-(Px/masas[k]);
    Vy[k]=Vy[k]-(Py/masas[k]);
    Vz[k]=Vz[k]-(Pz/masas[k]);
    v2 = pow(Vx[k],2) + pow(Vy[k],2)+ pow(Vz[k],2);
    KE= KE + 0.5*10000000*v2*masas[k];
  }
          CURRTEMP=(2*KE)/(3*totpart*GAS);
          CHI=sqrt(PARAMETROS.temperatura/CURRTEMP);
           MPI::COMM_WORLD.Send(&CHI,1,MPI::DOUBLE,0,1);  //retorna el chi al
                    procesador master “0”
 } }
```

Aplicación 3.

Paralelización de la primera parte del algoritmo de velocity verlet.

```
#include "common_dizzy.h"
#include "dizzy.h"
#include "stdio.h"
#include "mpi.h"
```

```
#include "stdlib.h"
#include "time.h"
#include "stdbool.h"
void movea_vv()
{     int   atnum,j;
      double axia , ayia , azia;
      double shiftx , shifty , shiftz;
      if ( options_.trace[0]==TRUE[0]  && my_id==0 ) printf("\nentering movea_vv");
      int ini1=1,cond1,I1,rc,master=0;
      if ( options_.freesp[0]==TRUE[0] || options_.fixedlatt[0]==TRUE[0] )
         ini1 = atpoint_.guestpt[1];
      else
         ini1 = 1;
   MPI_Comm_size( MPI_COMM_WORLD, &num_procs);
// Calculo del segmento del vector a asignar a cada procesador esclavo
if ( cmin_.dynregion[0]==TRUE[0] )
{       I1= cmin_.totdynatoms/(num_procs-1);
      if (cmin_.totdynatoms % I1 != 0) I1++;
}
else
{
 I1= atpoint_.npt/(num_procs-1);
      if ( atpoint_.npt % I1 != 0) I1++;
 }
double tcrdx2[atomlim],tcrdx[atomlim],tcrdy[atomlim],tcrdz[atomlim],tacrdx[atomlim],
tacrdy[atomlim],tacrdz[atomlim],tvx[atomlim],tvy[atomlim],tvz[atomlim];
   MPI_Comm_rank( MPI_COMM_WORLD, &my_id);
  if (my_id !=0)
  {ini1=((my_id-1)*I1)+1;
  cond1=(I1)*(my_id); }
if (my_id==0)  //Procesador master o raiz
 {  //envio de tareas a los procesadores esclavos
  for (i=1;i<num_procs;i++)
        {
       rc=MPI_Send(&coords_.crdx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i,MPI_COMM_WORLD);
        if (rc != MPI_SUCCESS) //validación de error en el envio del mensaje
        printf("%d: Send failure on round %d\n", my_id, i);
rc=MPI_Send(&coords_.crdy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+30,MPI_COMM_WORLD);
        if (rc != MPI_SUCCESS)
        printf("%d: Send failure on round %d\n", my_id, i+30);
rc=MPI_Send(&coords_.crdz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+60,MPI_COMM_WORLD);
 if (rc != MPI_SUCCESS)
        printf("%d: Send failure on round %d\n", my_id, i+60);
rc=MPI_Send(&coords_.acrdx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+90,MPI_COMM_WORLD);
        if (rc != MPI_SUCCESS)
        printf("%d: Send failure on round %d\n", my_id, i+90);
rc=MPI_Send(&coords_.acrdy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+120,MPI_COMM_WORLD
```

```
);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+120);
rc=MPI_Send(&coords_.acrdz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+150,MPI_COMM_WORLD
);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+150);
rc=MPI_Send(&velocity_.vx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+180,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+180);
rc=MPI_Send(&velocity_.vy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+210,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+210);
rc=MPI_Send(&velocity_.vz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+240,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+240);
rc=MPI_Ssend(&force_.fx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+500,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+500);
rc=MPI_Ssend(&force_.fy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+530,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+530);
rc=MPI_Ssend(&force_.fz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+560,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+560);
rc=MPI_Ssend(&mass_.amass,typelim,MPI_DOUBLE,i,i+590,MPI_COMM_WORLD);
       if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", my_id, i+590);  }
 for(i=1;i<num_procs;i++)     //Recepción de tareas asignadas
 {
 rc=MPI_Recv(&coords_.crdx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+300,MPI_COMM_WORLD,
&status );
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+300);
rc=MPI_Recv(&coords_.crdy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+330,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+330);
rc=MPI_Recv(&coords_.crdz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+360,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+360);
rc=MPI_Recv(&coords_.acrdx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+390,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+390);
rc=MPI_Recv(&coords_.acrdy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+420,MPI_COMM_WORLD,
```

```
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+420);
rc=MPI_Recv(&coords_.acrdz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+450,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+450);
rc=MPI_Recv(&velocity_.vx[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+480,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+480);
rc=MPI_Recv(&velocity_.vy[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+510,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+510);
rc=MPI_Recv(&velocity_.vz[((i-1)*I1)+2],I1,MPI_DOUBLE,i,i+540,MPI_COMM_WORLD,
&status);
       if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", my_id, i+540);  }
   }
 else
  {
// Procesadores Slaves- Recepción de tareas en variables temporales
rc=MPI_Recv(&tcrdx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id,
MPI_COMM_WORLD,&status);
 if (rc != MPI_SUCCESS)
    printf("%d: Receive failure on round %d\n", my_id, i);
rc=MPI_Recv(&tcrdy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+30,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
    printf("%d: Receive failure on round %d\n", my_id, my_id+30);
rc=MPI_Recv(&tcrdz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+60,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
    printf("%d: Receive failure on round %d\n", my_id, my_id+60);
rc=MPI_Recv(&tacrdx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+90,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)printf("%d: Receive failure on round %d\n", my_id, my_id+90);
rc=MPI_Recv(&tacrdy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+120,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
    printf("%d: Receive failure on round %d\n", my_id, my_id+120);
rc=MPI_Recv (&tacrdz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+150,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
    printf("%d: Receive failure on round %d\n", my_id, my_id+150);
rc=MPI_Recv (&tvx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+180,
```

```
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+180);
rc=MPI_Recv (&tvy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+210,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, i+210);
rc=MPI_Recv (&tvz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+240,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+240);rc=MPI_Recv
(&force_.fx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+500,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+500);
rc=MPI_Recv (&force_.fy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+530,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+530);
rc=MPI_Recv (&force_.fz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+560,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+560);
rc=MPI_Recv (&mass_.amass,typelim,MPI_DOUBLE,master,my_id+590,
MPI_COMM_WORLD,&status);
if (rc != MPI_SUCCESS)
   printf("%d: Receive failure on round %d\n", my_id, my_id+590);
//Realización de tareas asignadas
        for (atnum = ini1;atnum<=cond1;atnum++)
         {
          j = cmin_.dynatoms[atnum];
           // determine the accelerations
          axia = force_.fx[j] / mass_.amass[atpoint_.atmtype[j]];
          ayia = force_.fy[j] / mass_.amass[atpoint_.atmtype[j]];
          azia = force_.fz[j] / mass_.amass[atpoint_.atmtype[j]];
          shiftx = params_.delt * tvx[j] + cons_.conew * axia;
          shifty = params_.delt * tvy[j] + cons_.conew * ayia;
          shiftz = params_.delt * tvz[j] + cons_.conew * azia;

          tcrdx[j]  = tcrdx[j]  + shiftx;
          tcrdy[j]  = tcrdy[j]  + shifty;
          tcrdz[j]  = tcrdz[j]  + shiftz;
          tacrdx[j] = tacrdx[j] + shiftx;
          tacrdy[j] = tacrdy[j] + shifty;
          tacrdz[j] = tacrdz[j] + shiftz;
          tvx[j] = tvx[j] + 0.5 * cons_.conewh * axia;
          tvy[j] = tvy[j] + 0.5 * cons_.conewh * ayia;
```

```
          tvz[j] = tvz[j] + 0.5 * cons_.conewh * azia;
         }
//Envio de resultados en cada procesador esclavo
rc=MPI_Ssend(&tcrdx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+300,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+300);
rc=MPI_Ssend(&tcrdy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,master,my_id+330,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+330);
rc=MPI_Ssend(&tcrdz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+360,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+360);
rc=MPI_Ssend(&tacrdx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+390,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+390);
rc=MPI_Ssend(&tacrdy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+420,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+420);
rc=MPI_Ssend(&tacrdz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+450,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+450);
rc=MPI_Ssend(&tvx[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+480,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+480);
rc=MPI_Ssend(&tvy[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+510,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+510);
rc=MPI_Ssend(&tvz[((my_id-1)*I1)+2],I1,MPI_DOUBLE,0,my_id+540,
MPI_COMM_WORLD);
if (rc != MPI_SUCCESS)
   printf("%d: Send failure on round %d\n", my_id,my_id+540);
 }
}
```

6. RESULTADOS

6.1 DESCRIPCIÓN DE PROGRAMAS

En el proceso de desarrollo se utilizó el compilador de Intel para Sistema Operativo Linux (I.C.C. Versión 9.0) el cual es gratuito y nos permite trabajar con las librerías del MPI.

El archivo ejecutable del proyecto se denomina dizzy_nombre_equipo por ejemplo: dizzy_cluster. Este ejecutable es el resultado de dos pasos:

1. Una compilación de cada uno de los programas fuente, generando un archivo de igual nombre al programa con extensión .o .

2. El enlace del programa con las librerías MPI, generando un ejecutable (dizzy_cluster) con todos los símbolos necesarios.

La creación del programa objeto se realiza a través de un archivo Makefile como el siguiente:

```
CODE = $(shell basename dizpar)
BINDIR = $(HOME)/recaman/simulaciones/marco/paralelo/
CC = mpicc -Iincludes #-I/opt/lam-7.0.6/include
CFLAGS = #-O #-showme:compile
LIBS = -L/opt/lam-7.0.6/lib -lmpi
INCLUDES = -Iincludes

TRUESRC = buck4r.c   buck.c          \
$(CODE): $(CC) $(CFLAGS) $^ -o $(BINDIR)$(CODE)_'host_name' $(LIBS)
```

El Makefile anterior guarda los comandos de compilación con todos sus parámetros que permiten encontrar los archivos binarios (bindir), las bibliotecas(libs), el listado de los programas fuentes (truesrc), entre otros. Este shell evita escribir largas líneas de compilación con multiples opciones que tendríamos que aprender de memoria. Con el Makefile sólo se necesita hacerlo una vez y el durante su ejecución creará el programa objeto dizpar*_host_name*. Existen en el proyecto otros Makefile que se crearón con argumentos que permiten crear los ejecutables para diferentes máquinas como IBM, Hewlett Packard, Sun, Silicon Graphics, entre otras. Además se realizaron Makefile para enlazar aplicativos de Fortran y Lenguaje C, sin embargo el alto volumen de programas complicaban el control de enlaces entre los dos tipos de código (common_blocks) .

En el directorio Include estan las siguientes directivas del preprocesador:

- Common_dizzy.h: Archivo con la declaración de las constantes, variables globales y las estructuras de datos u objetos.
- Common_error.h: Contiene la declaración de las variables globales utilizadas para el control de los errores en los parámetros de entrada.
- Common_parse.h: Directiva que contiene la estructura parse que controla los tipos de términos capturados en los parámetros de entrada.
- Dizzy.h: Archivo con la declaración de las funciones.

Los archivos fuera del directorio includes son programas fuentes de lenguaje C.

En el Diagrama de Flujo de Datos (DFD) del capítulo 4.3 se describe gráficamente el flujo de la información. En el DFD se modelaron cuatro grupos de procesos diferenciados por un color en la escala de grises. A continuación se describe cada uno de estos cuatro procesos:

1. INICIALIZACION DE DATOS

En esta sección se comienza la ejecución del sistema dizzy y se inicializan los objetos que forman los parámetros globales y que contienen las banderas, vectores y matrices.

2. LECTURA DE PARÁMETROS Y CREACIÓN DE LA CONFIGURACIÓN INICIAL

Se realiza la lectura del archivo secuencial dump.old, acorde con la estructura de datos explicada en el archivo secuencial (fort51).

El flujo de datos continúa con la lectura de los parámetros suministrados al archivo de entrada pars_*name*.dat. En este archivo se captura secuencialmente la información y se valida cada parámetro para conocer la cantidad de términos ingresados y su correspondiente tipo de dato (lógicos, enteros, reales y alfanuméricos). Acorde con esta información se le asigna su correspondiente variable global.

El Flujo continúa con la medición y comparación de las etiquetas de cada átomo con las ya existentes. Este proceso se realiza según el tipo de relación entre los átomos que pueden ser un enlace (2 átomos), un ángulo (tres átomos) y una torsión (cuatro átomos).

En el caso que exista un error en los datos de entrada anteriormente mencionados el sistema genera el mensaje descriptivo correspondiente,

acompañado de la línea que causa la terminación del proceso e ejecución. Sino es el caso se avanza con la definición de los parámetros resultantes de cálculos numéricos en los datos de entrada y se obtiene un mínimo de energía que permita determinar las ubicaciones atómicas más probables de los átomos.

Realiza la configuración de los vectores recíprocos para ser utilizados en la sumatoria de Ewald, configura los parámetros de restricción y efectúa la función del potencial de Lennard Jones

3. CALCULO DE FUERZAS, ENERGÍAS Y OTRAS VARIABLES TERMODINÁMICAS DE INTERÉS

Este bloque de programas es el más importante del sistema dizzy porque consume el mayor tiempo de ejecución al realizar los cálculos aplicando el método de Velocity Verlet para simular moléculas de forma más estable y de convergencia más rápida.

Comienza con la inicialización de las variables utilizadas para contar y acumular energía, temperatura y calor. En caso que se requiera las velocidades y posiciones iniciales de los átomos pueden ser determinadas por medio del método de distribución Gaussiana tomando como base el parámetro de la temperatura inicial y fundamentada en el Numerical Recipes.

El sistema Continúa con la evaluación del tipo de interacción a realizar y enlaza las rutinas que calculan la energía y fuerza en los átomos del sistema. En este módulo se encuentra el ciclo principal del sistema Dizzy donde se determina el tiempo de ejecución del modelo.

El cálculo de las fuerzas y energía para todos los átomos se realiza con base en las ecuaciones que se referencia en la siguiente figura:

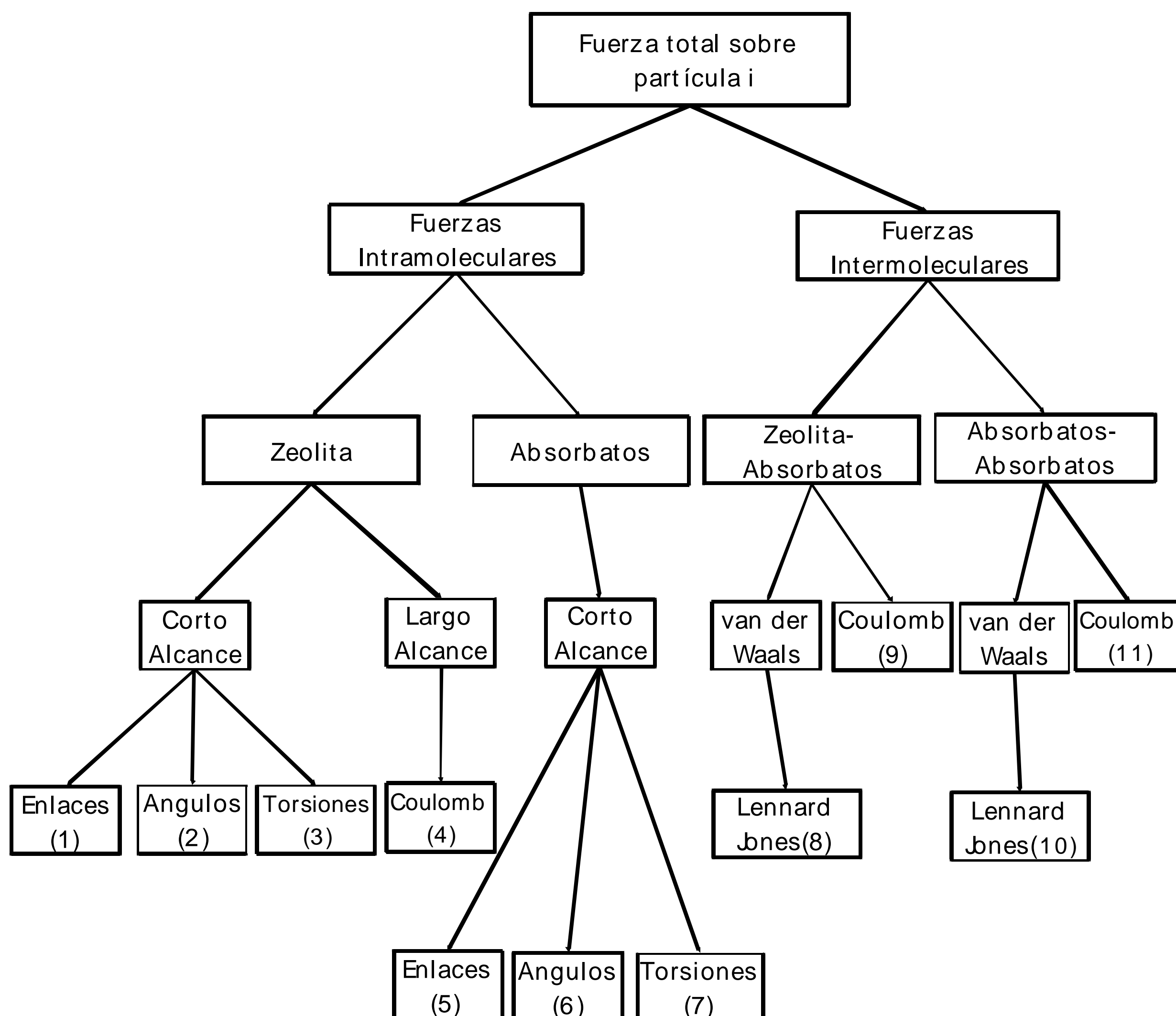


Ilustración 11. Cálculo de la Fuerza

$$F_{total} = \sum_{i=1}^{11} F_i$$

$$F_1 = \sum_{i=1}^{j} F_{ij}$$

todos los pares de átomos en las zeolitas

$$F_2 = \sum_{i=1}^{j} \sum_{j \neq i}^{k} F_{ijk}$$

(3) $F_3 = \sum_{i=1}^{j} \sum_{j=1}^{k} \sum_{k=1}^{l} F_{ijkl}$ *ijkl* es el conjunto de 4 átomos que toma una torsión en la zeolita

(4) $F_4 = \sum_{i=1}^{j} F_{ij}$ calculado mediante suma de Ewald con espacio real y recíproco

(5) $F_5 = \sum_{i=1}^{j} F_{ij}$ para todos los pares de átomos enlazados en las moléculas de adsorbatos

(6) $F_6 = \sum_{i=1}^{j} \sum_{j=1}^{k} F_{ijk}$ para todos los tríos de átomos que forman un ángulo en los adsorbatos

(7) $F_7 = \sum_{i=1}^{j} \sum_{j=1}^{k} \sum_{k=1}^{l} F_{ijkl}$ para los grupos de 4 átomos que forman una torsión enlos adsorbatos

(8) $F_8 = \sum_{i=1}^{j} F_{ij}$ todas las interacciones entre átomos zeolitas y átomos adsorbatos tipo van der Waals

(9) $F_9 = \sum_{i=1}^{j} F_{ij}$ todas las interacciones entre átomos zeolitas y átomos adsorbatos Coulomb

(10) $F_{10} = \sum_{i=1}^{j} F_{ij}$ todas las interacciones entre los átomos de los adsorbatos y de los otros adsorbatos tipo van der Waals

(11) $F_{11} = \sum_{i=1}^{j} F_{ij}$ todas las interacciones entre los átomos de los adsorbatos y los otros átomos adsorbatos Coulomb

Al cálculo de cada posición de las partículas se les aplica las condiciones periódicas de frontera en cada uno de los átomos que superan las coordenadas de la celda unitaria.

4. REGISTRO DE RESULTADOS EN ARCHIVOS SECUENCIALES.

Finalmente se imprimen los siguientes cálculos:

1. Tiempo de proceso en cada iteración que se realiza.
2. Parámetros iniciales de simulación como Lennard Jones, Buckingham, Morse, torsiones, entre otros.
3. Información resultante en forma periódica y acorde a la estructura del archivo secuencial Fort.51.
4. Valores de seguimiento de energía resultantes del proceso en un periodo determinado (archivo .log) .
5. Valores de energía resultantes en el archivo secuencial fort.23.
6. Registra los resultados siguiendo la estructura de datos del archivo secuencial fort.19.

6.2 DESCRIPCIÓN DE ARCHIVOS SECUENCIALES

Programa: wrt_txt.c
Archivo: fort.19
Ejemplo:
12
1
C 10.86939 6.37268 12.71688
Descripción:

12:	Número de átomos por molécula huésped.
1:	Número del ciclo de modelamiento.
C:	Etiqueta del átomo.
10.86939:	Coordenada del átomo en el eje X.
6.37268:	Coordenada del átomo en el eje Y.
12.71688:	Coordenada del átomo en el eje Z.

Programa: wrt_enefile.c
Archivo: fort.23
Ejemplo:
1 0.115703 -8770.545697 0.008779 -8770.536918064600715
Descripción:

1:	Número del ciclo de modelamiento.
0.115703:	Temperatura.
-8770.545697:	Energía Potencial.
0.008779:	Energía Cinética.
8770.536918064600715:	Energía Total del Sistema.

En la descripción del siguiente archivo secuencial es fundamental interpretar los

siguientes conceptos:

Molécula Huésped: Partícula formada por un conjunto de átomos que producen vibraciones moleculares. Una molecula es considerada huésped cuando es alojada por otra.
Celda Unitaria: Arreglo espacial de átomos que se repite en el espacio tridimensional definiendo la estructura de cristal. Se caracteriza por tres vectores que definen las tres direcciones independientes del sistema de coordenadas de la celda.

Programa: wrt_dump.c
Archivo: fort.51
Ejemplo:

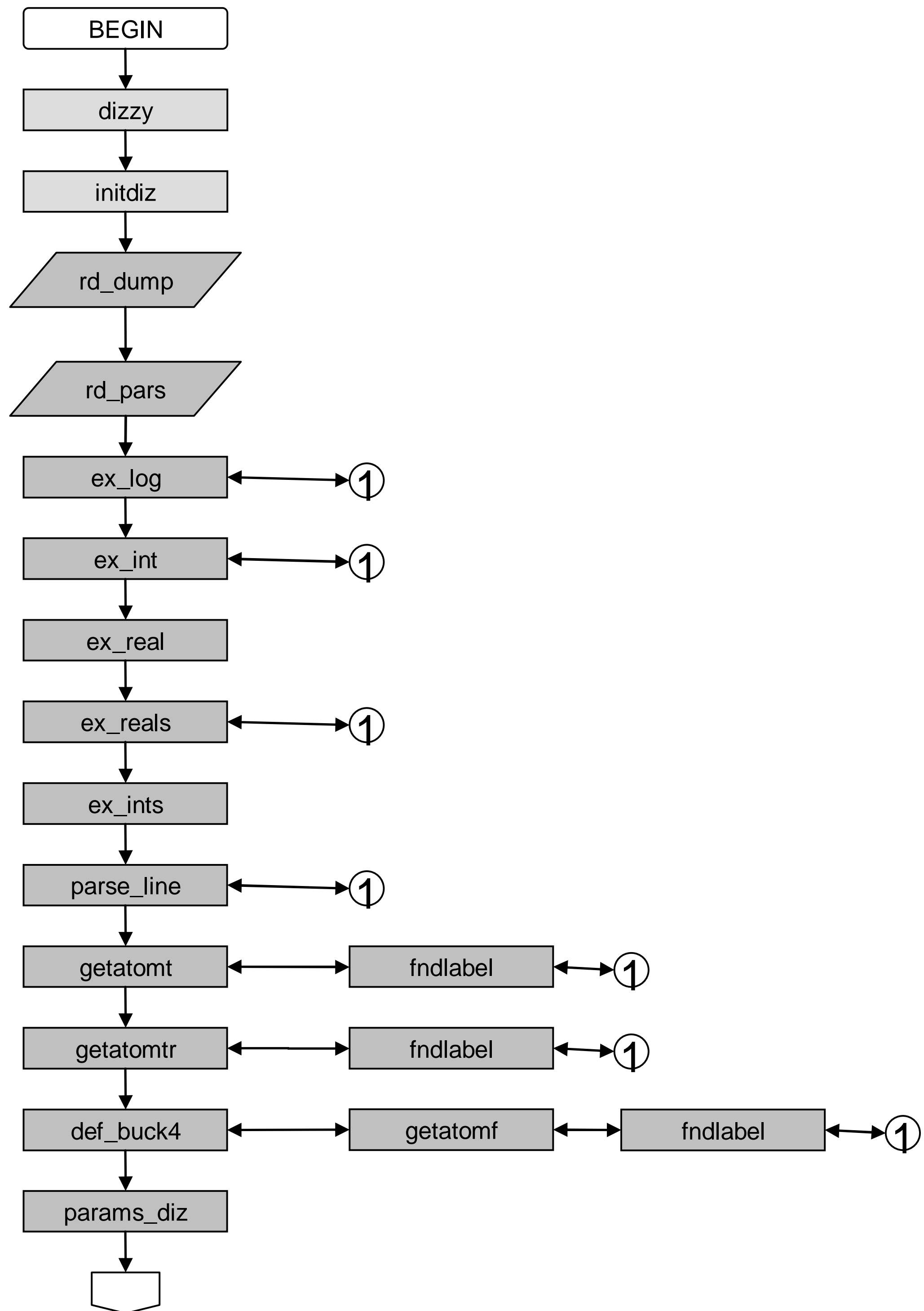
```
6.32
 T F F F T T 1
 588 1  1 12
 2  1
 193
 3
 8.70538259500000    1.20552811300000    8.62353372800000
        1
-1.02500000000000
-1.97545405927285E-005  2.33161378660247E-005 -1.05549659614350E-006
100     600.00     0.00
 Si  28.09 0
20.0220000000000       19.8990000000000       26.7660000000000
8.70538259500000       1.20552811300000       8.62353372800000
1170   1152
193    1       305
2
577        581
24
578        582        579        580
```

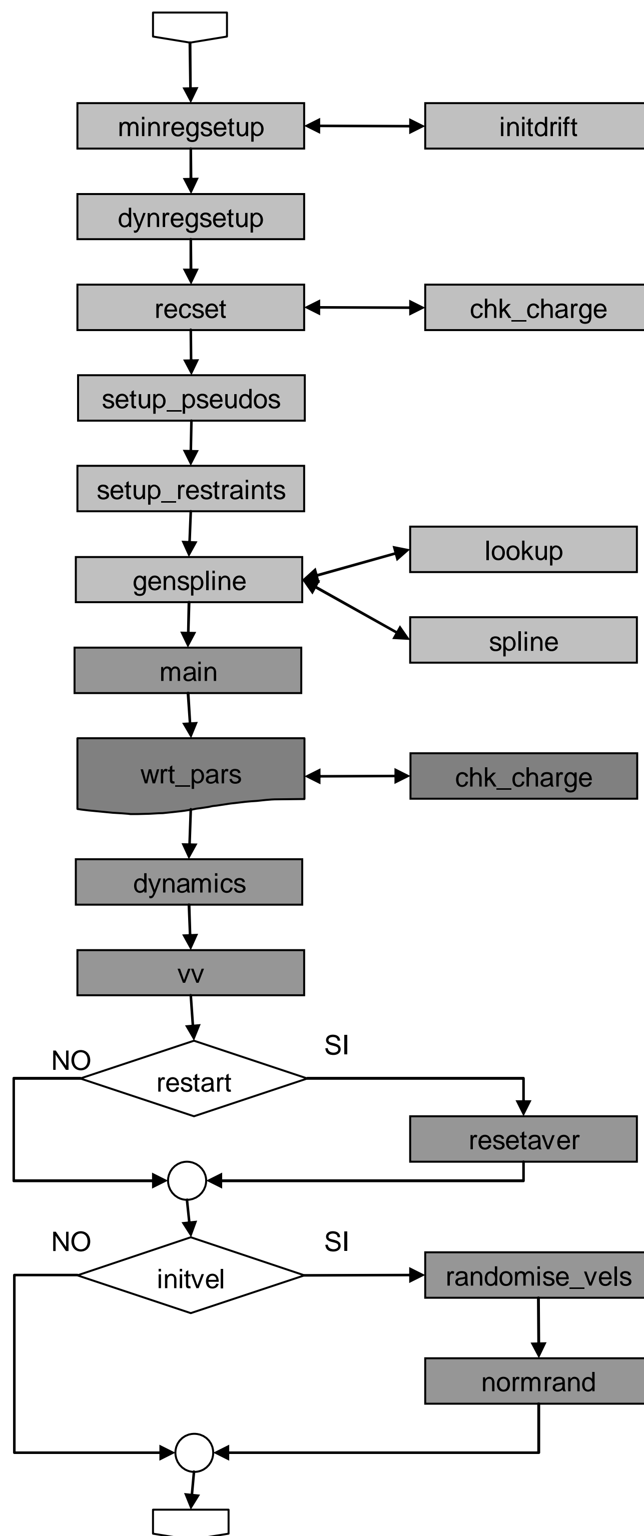
Descripción:

6.32:	Número de la versión del sistema Dizzy.
T:	Bandera para trabajar o nó con la carga del átomo.
F:	Bandera para imprimir o nó condiciones indirectas.
F:	Bandera para imprimir o nó condiciones directas.
F:	Bandera para átomos fijos o nó .
T:	Bandera para calcular o nó la fuerza en tres átomos enlazados
T:	Bandera para trabajar o nó con molécula huésped.
1:	Número de fronteras del huésped.
588:	Número total de partículas a modelar.

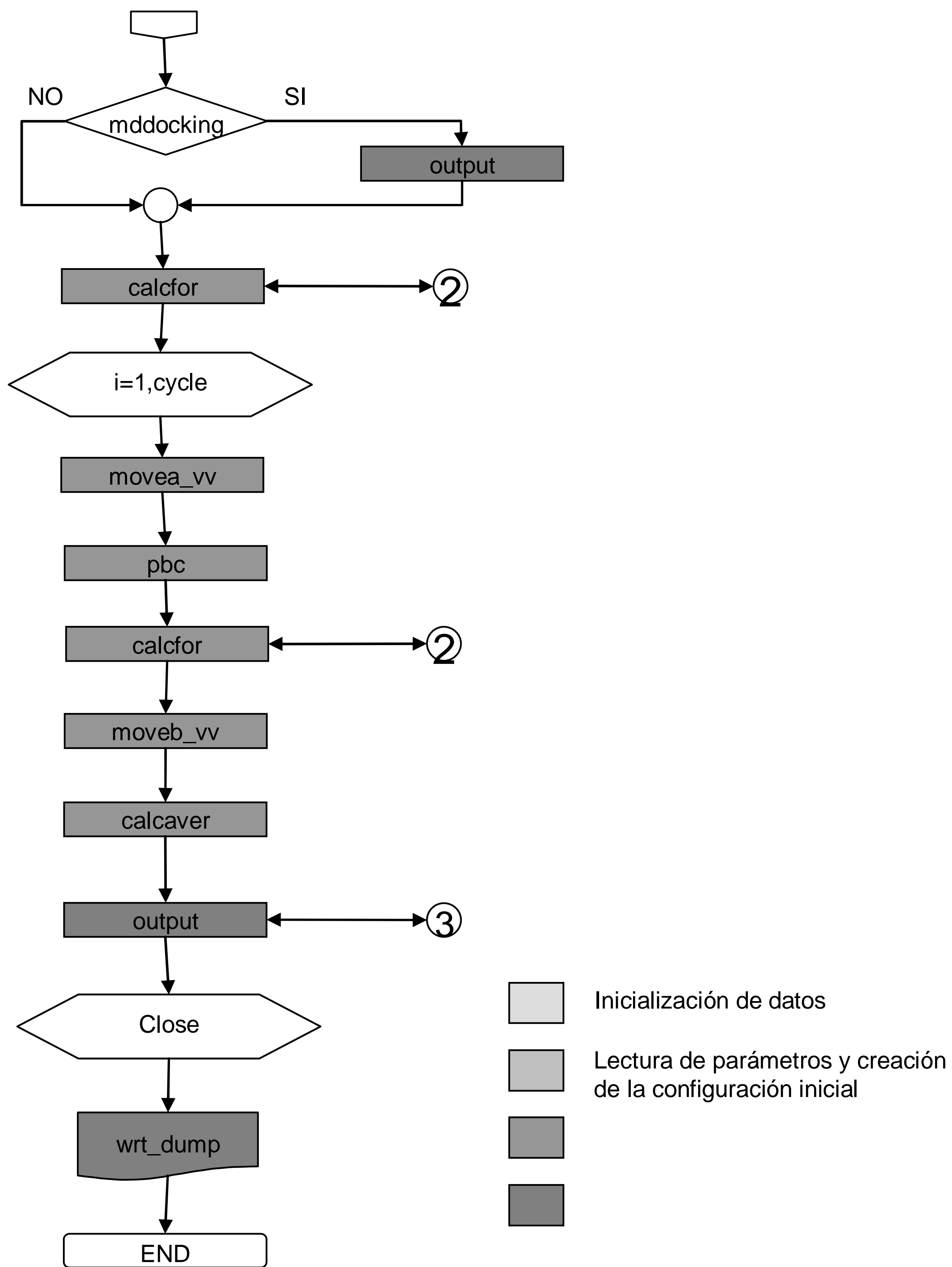
1: Número total de moléculas a modelar.
1: Número total moléculas huésped.
12: Número de átomos por molécula huésped.
1: Número total de átomos de zeolita.
12: Número total de átomos huésped.
2: Número total de zeolitas y huésped.
1: Puntero uno del arreglo para zeolitas.
193: Puntero dos del arreglo para huésped.
3: Tipos huésped.
8.70538259500000 : Coordenada del eje X en cada átomo.
1.20552811300000 : Coordenada del eje Y.
8.62353372800000 : Coordenada del eje Z.
1 : Tipo de átomo.
-1.02500000000000: Carga del átomo.
-1.97545405927285E-005 2.33161378660247E-005 -1.05549659614350E-006:
Velocidad en el eje X,Y, Z de cada átomo.
100: Número de ciclos de modelamiento.
600.00: Temperatura.
0.00: Periodo de tiempo para el registro de la información.
Si: Etiqueta del átomo.
28.09: Masa del átomo.
0: Tipo de átomo.
20.0220000000000 19.8990000000000 26.7660000000000:
Tamaño de la celda unitaria en X,Y,Z.
8.70538259500000 1.20552811300000 8.62353372800000:
Coordenada X,Y,Z de cada átomo.
1170: Número total de átomos enlazados en grupos de tres-tripletes.
1152: Número total tripletes de átomos en la zeolita.
193 1 305 : Número correspondiente a cada átomo O-Si-O.
2: Número total de fronteras para el huésped.
577 581: Número de átomos que forman un par.
24: Número total de torsiones para la molécula huésped.
578 582 579 580: Número de los átomos enlazados en grupos de cuatro.

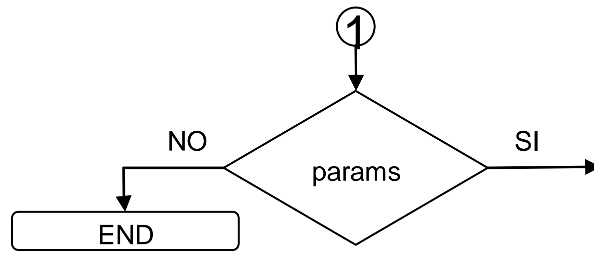
6.3 DIAGRAMA DE FLUJO

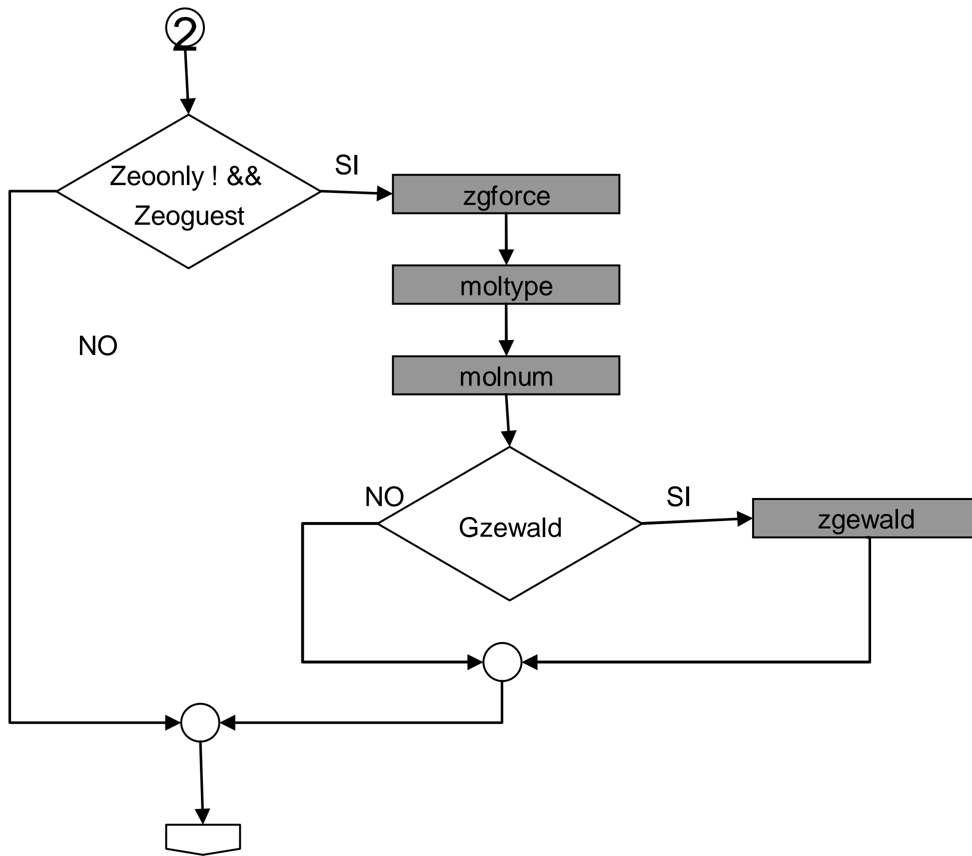


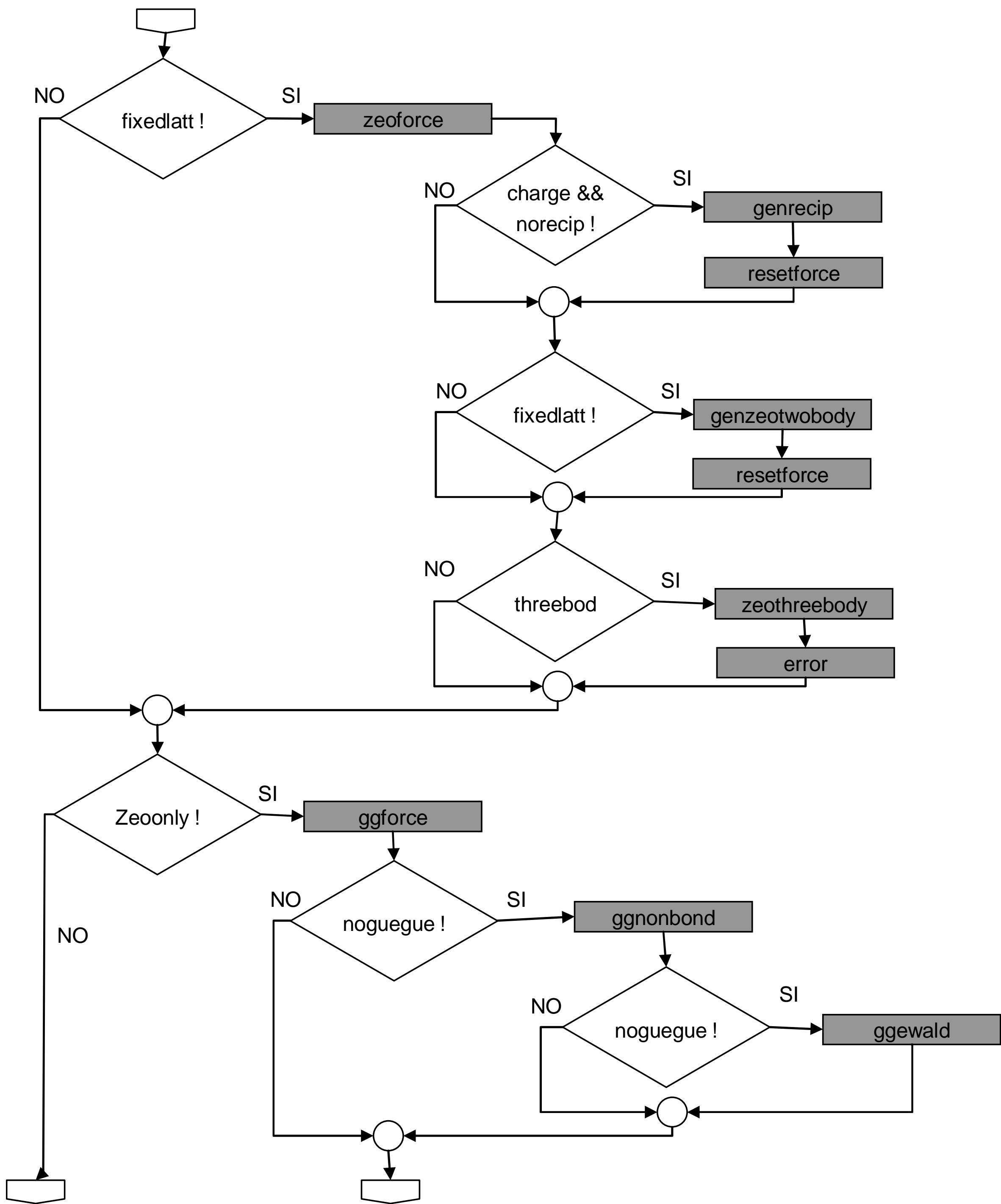


minregsetup
initdrift
dynregsetup
recset
chk_charge
setup_pseudos
setup_restraints
lookup
genspline
spline
main
wrt_pars
chk_charge
dynamics
vv
NO
restart
SI
resetaver
NO
initvel
SI
randomise_vels
normrand

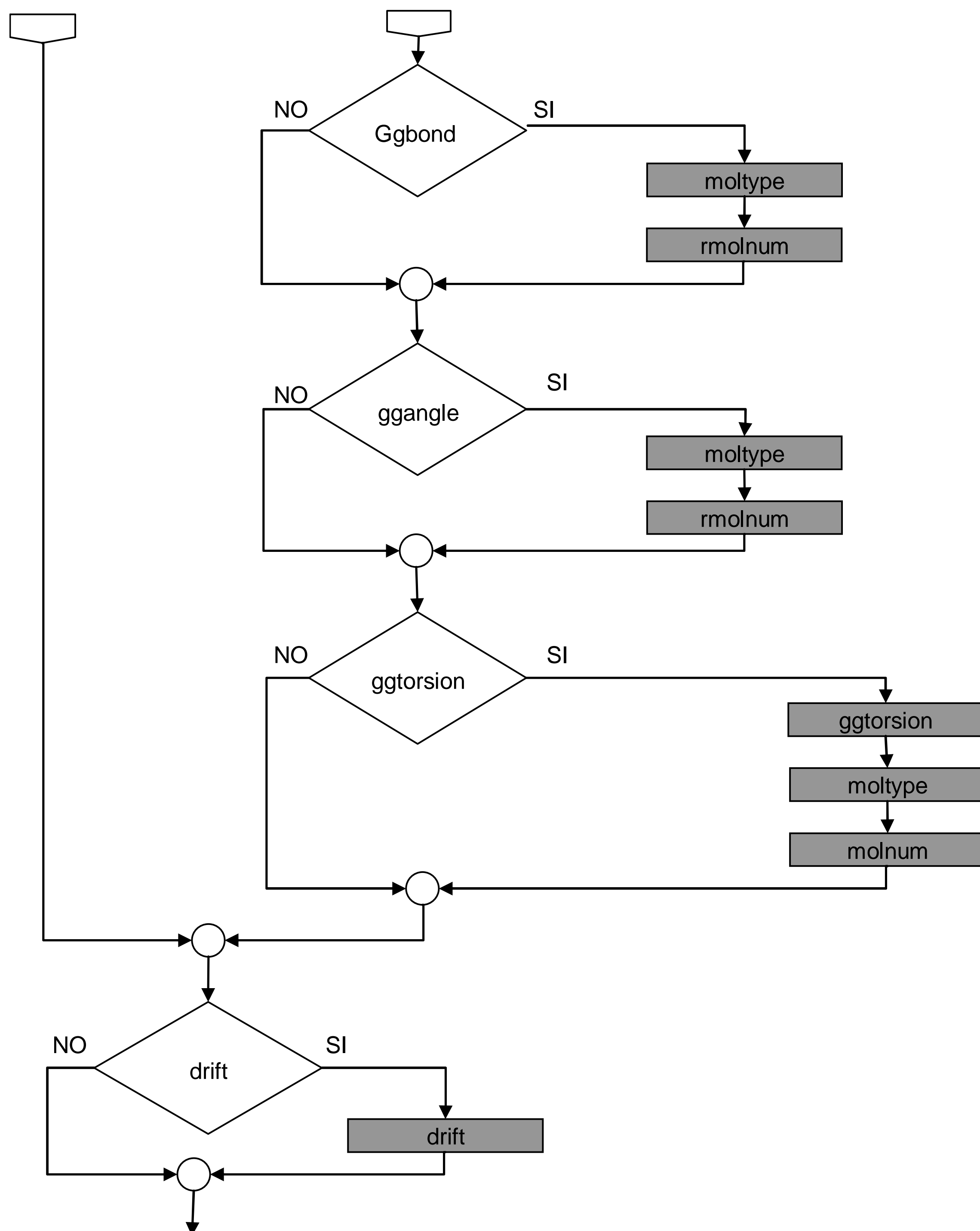

NO
SI
mddocking
output
calcfor
2
i=1,cycle
movea_vv
pbc
calcfor
2
moveb_vv
calcaver
output
3
Close
wrt_dump
END
Inicialización de datos
Lectura de parámetros y creación
de la configuración inicial


1
NO
params
SI
END

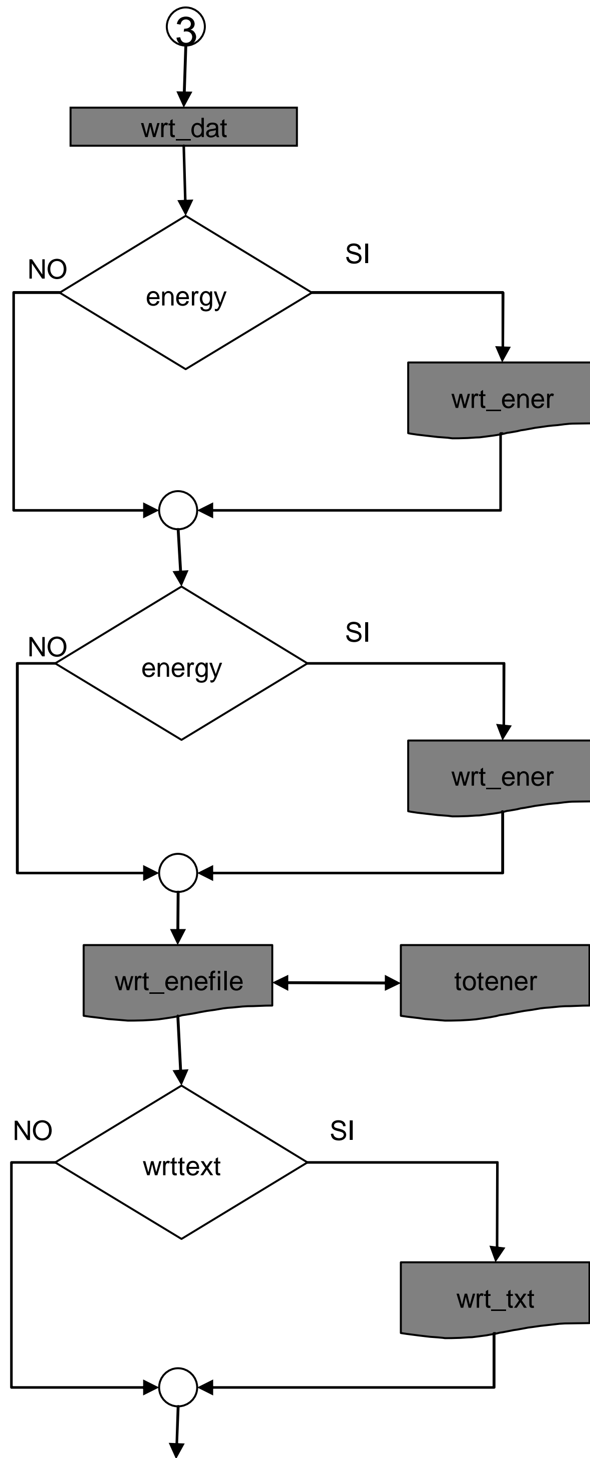

2
Zeoonly ! &&
Zeoguest
SI
zgforce
moltype
NO
molnum
NO
Gzewald
SI
zgewald



fixedlatt !
NO
SI
zeoforce
charge &&
norecip !
NO
SI
genrecip
resetforce
fixedlatt !
NO
SI
genzeotwobody
resetforce
threebod
NO
SI
zeothreebody
error
Zeoonly !
SI
NO
ggforce
noguegue !
NO
SI
ggnonbond
noguegue !
NO
SI
ggewald



NO
Ggbond
SI
moltype
rmolnum
NO
ggangle
SI
moltype
rmolnum
NO
ggtorsion
SI
ggtorsion
moltype
molnum
NO
drift
SI
drift


3
wrt_dat
NO
energy
SI
wrt_ener
NO
energy
SI
wrt_ener
wrt_enefile
totener
NO
wrttext
SI
wrt_txt

6.4 GRAFICACION DE RESULTADOS

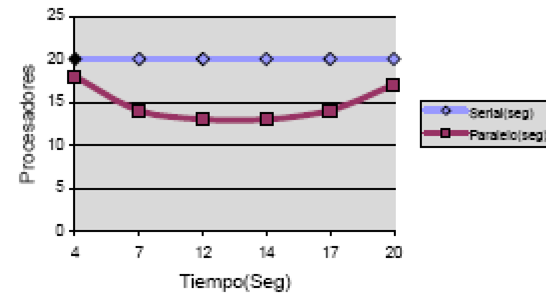
Iteraciones Totales

Tiempo total de simulacion

588 ATOMOS Y 100 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	20	18
7	20	14
12	20	13
14	20	13
17	20	14
20	20	17

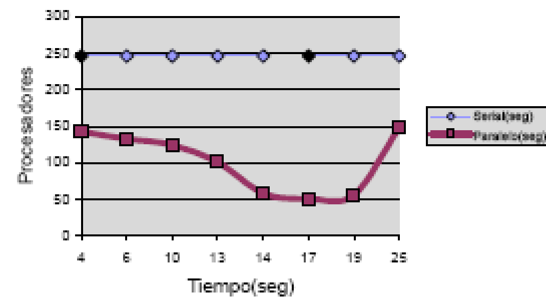
/home/giftex/recaman/simulaciones/paralelo/
/home/giftex/recaman/simulaciones/serial/ciclo100



15552 ATOMOS y 10 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	246	143
6	246	133
10	246	124
13	246	101
14	246	58
17	246	51
19	246	56
25	246	148

/home/giftex/recaman/simulaciones/marco/paralelo/
/home/giftex/recaman/simulaciones/marco/serial/ciclo10

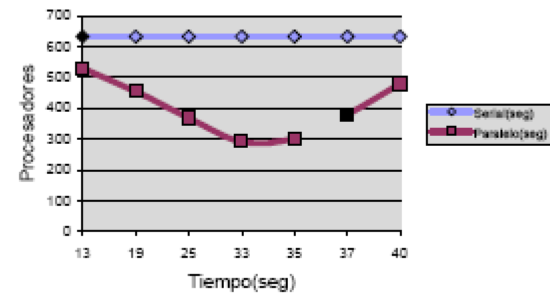


Iteraciones Totales

27678 ATOMOS y 10 iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
13	634	530
19	634	455
25	634	369
33	634	291
35	634	302
37	634	380
40	634	480

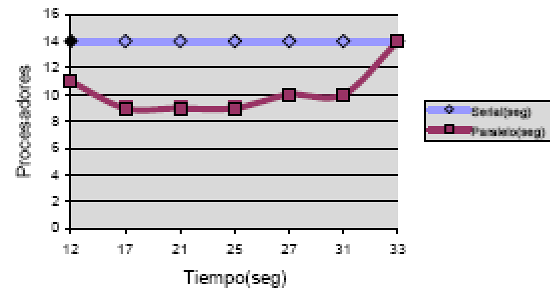
/home/giftex/recaman/simulaciones/marco/pwolf/
/home/giftex/recaman/simulaciones/marco/wolf/ciclo10



2160 ATOMOS y 10 iteracion WOLF

Procesadores	Serial(seg)	Paralelo(seg)
12	14	11
17	14	9
21	14	9
25	14	9
27	14	10
31	14	10
33	14	14

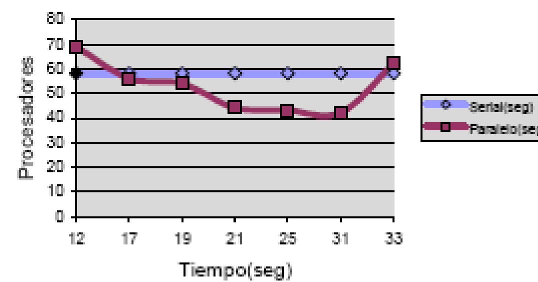
/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ciclo10



Iteraciones Totales

2160 ATOMOS y 10 iteracion EWALD

Procesadores	Serial(seg)	Paralelo(seg)
12	58	69
17	58	56
19	58	54
21	58	44
25	58	43
31	58	42
33	58	62



/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ewald10

En las anteriores 6 graficas del numero de procesadores Vs Tiempo observamos un comportamiento lineal para el programa en serie, mientras en el programa en paralelo es una función parabólica con valores inferiores al serie.

La fila sombreada es aquella que consume menos tiempo de cómputo en paralelo. El path corresponde a la ubicacion de los archivos planos resultantes que producen resultados iguales en todas las ejecuciones.

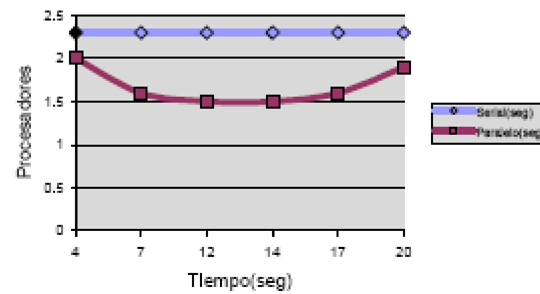
La evaluación del desempeño del programa en paralelo con respecto al serial se realiza en el tema configuración del sistema (ver capitulo 4.5).

Primera Iteración

588 ATOMOS Y 100 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	2.3	2
7	2.3	1.6
12	2.3	1.5
14	2.3	1.5
17	2.3	1.6
20	2.3	1.9

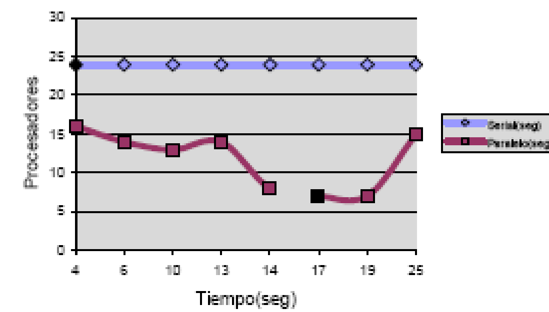
/home/giftex/recaman/simulaciones/paralelo/
/home/giftex/recaman/simulaciones/serial/ciclo100



15552 ATOMOS y 10 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	24	16
6	24	14
10	24	13
13	24	14
14	24	8
17	24	7
19	24	7
25	24	15

/home/giftex/recaman/simulaciones/marco/paralelo/
/home/giftex/recaman/simulaciones/marco/serial/ciclo10

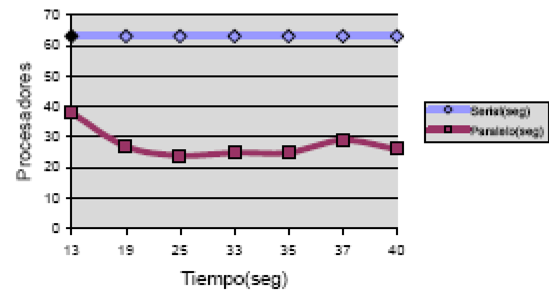


Primera Iteración

27678 ATOMOS y 10 iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
13	63	38
19	63	27
25	63	24
33	63	25
35	63	25
37	63	29
40	63	26

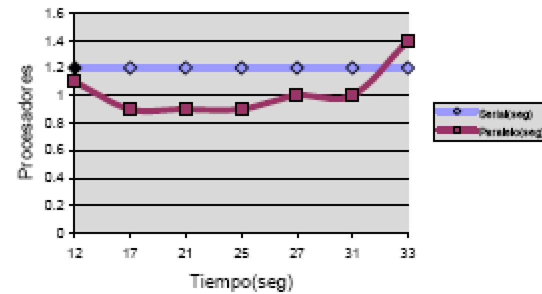
/home/giftex/recaman/simulaciones/marco/pwolf/
/home/giftex/recaman/simulaciones/marco/wolf/ciclo10



2160 ATOMOS y 10 iteracion WOLF

Procesadores	Serial(seg)	Paralelo(seg)
12	1.2	1.1
17	1.2	0.9
21	1.2	0.9
25	1.2	0.9
27	1.2	1
31	1.2	1
33	1.2	1.4

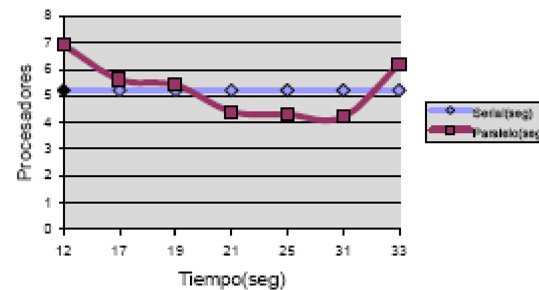
/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ciclo10



Primera Iteración

2160 ATOMOS y 10 iteracion	EWALD	
Procesadores	Serial(seg)	Paralelo(seg)
12	5.2	6.9
17	5.2	5.6
19	5.2	5.4
21	5.2	4.4
25	5.2	4.3
31	5.2	4.2
33	5.2	6.2

/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ewald10



Para la primera iteracion se observan el comportamiento anteriormente mencionado. Podemos analizar que la primera iteración es la que requiere de un mayor tiempo de cómputo debido a que calcula los parámetros para utilizarlos en las demas iteraciones. Además la cantidad de procesadores en donde se obtiene el menor tiempo de procesamiento varia con respecto al número de átomos utilizados en la Dinámica Molecular.

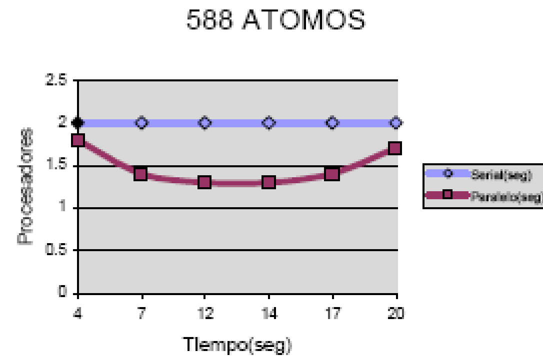
Iteraciones Centrales

Tiempo promedio Iteraciones

588 ATOMOS Y 100 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	2	1.8
7	2	1.4
12	2	1.3
14	2	1.3
17	2	1.4
20	2	1.7

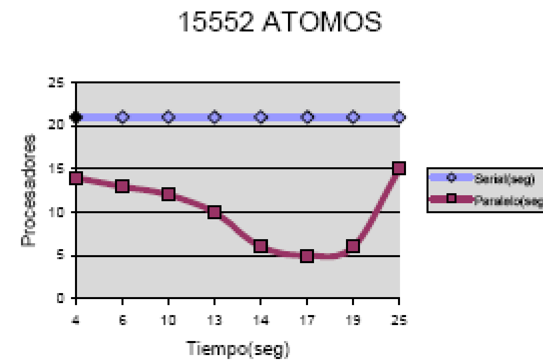
/home/giftex/recaman/simulaciones/paralelo/
/home/giftex/recaman/simulaciones/serial/ciclo100



15552 ATOMOS y 10 Iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
4	21	14
6	21	13
10	21	12
13	21	10
14	21	6
17	21	5
19	21	6
25	21	15

/home/giftex/recaman/simulaciones/marco/paralelo/
/home/giftex/recaman/simulaciones/marco/serial/ciclo10

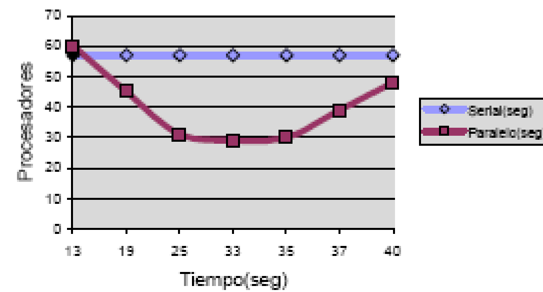


Iteraciones Centrales

27678 ATOMOS y 10 iteraciones

Procesadores	Serial(seg)	Paralelo(seg)
13	57	60
19	57	45
25	57	31
33	57	29
35	57	30
37	57	39
40	57	48

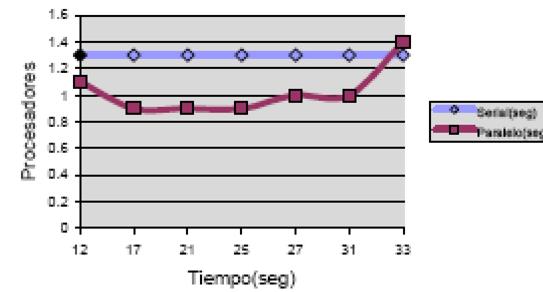
/home/giftex/recaman/simulaciones/marco/pwolf/
/home/giftex/recaman/simulaciones/marco/wolf/ciclo10



2160 ATOMOS y 10 iteracion WOLF

Procesadores	Serial(seg)	Paralelo(seg)
12	1.3	1.1
17	1.3	0.9
21	1.3	0.9
25	1.3	0.9
27	1.3	1
31	1.3	1
33	1.3	1.4

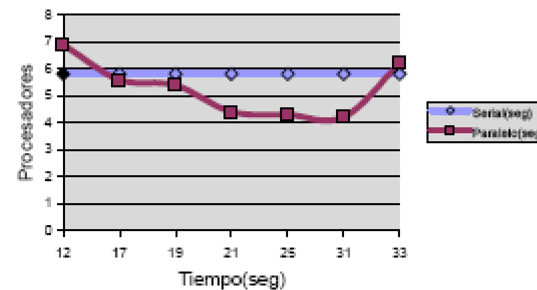
/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ciclo10



Iteraciones Centrales

2160 ATOMOS y 10 iteracion EWALD		
Procesadores	Serial(seg)	Paralelo(seg)
12	5.8	6.9
17	5.8	5.6
19	5.8	5.4
21	5.8	4.4
25	5.8	4.3
31	5.8	4.2
33	5.8	6.2

/home/giftex/recaman/simulaciones/marco/torsionpar/
/home/giftex/recaman/simulaciones/marco/torsion/ewald10



Las iteraciones centrales requieren de menor tiempo de cómputo que la primera iteración por la utilización de parámetros previamente calculados.

La cantidad de procesadores en las que se consume menos tiempo varia con respecto al número de átomos (dominio de ejecución). Si el dominio en ejecución aumenta, el número de procesadores que puedo utilizar para obtener un mejor resultado también aumenta, la razón es la mayor cantidad de cálculos a realizar en cada uno de ellos.

Se analiza que el comportamiento en paralelo es parabólico con sentido descendente al aumentar el número de procesadores, alcanzando un mínimo y posteriormente cambia de sentido debido al costo en la comunicación. El aumento en la cantidad de mensajes es directamente proporcional con la cantidad de procesadores y el tiempo de ejecución.

6.5 CONFIGURACIÓN DEL SISTEMA

El programa Dizzy en paralelo se desarrolló en un *cluster* del tipo *Beowulf*. Este equipo consta de:

- 19 nodos computacionales, Hewlett-Packard, Proliant DL140, Doble procesador Intel Xeon 3.0 GHz HT, 1GB RAM, DD 80 GB SATA.
- 1 nodo principal, DELL PowerEdge 1850, Doble procesador Intel Xeon 3.4 GHz HT, 2 GB RAM, 2 DD 146GB SCSI.
- 1 *switch* 3COM 3870 manejable de 24 puertos capa L3, 1 GB ancho de banda.

El sistema operativo es RedHat Enterprise Linux 4, utiliza un administrador de *cluster* Open Source Cluster Aplication Resource (OSCAR), la interfaz de paso de mensajes MPI y los lenguajes de programación C++ de Intel (ICC) y FORTRAN de Intel.

La evaluación del desempeño de un programa en paralelo se realiza de la siguiente forma:

6.5.1 MODELO Y MÉTODO:

Modelo matemático --> Análisis Numérico
Software (Proyecto Dizzy) --> *Simulación de eventos discretos*
Prototipo --> Observación

6.5.2 COMUNICACIÓN:

En la evaluación del rendimiento, la comunicación determina el tiempo de transmisión y recepción dinámica de los mensajes generados en el código paralelizado. La comunicación en el cluster se evalúa a través del ancho de banda y la latencia. Para evaluar estos valores se ejecuta un programa en paralelo que envíe 100 mensajes entre 2 nodos y se cambia el tamaño del mensaje obteniendo la siguiente información promedio:

Tamaño	Ancho de Banda (MB/s)	Latencia (microsegundos)
50K	11.440966	190.500000
100K	11.456341	188.000000
1MB	11.456842	128.000000

El ancho de banda es la cantidad en bytes de datos que se puede transmitir en

una unidad de tiempo. Se observa en los anteriores resultados que el ancho de banda se mantiene estable ante el cambio del tamaño del mensaje. La latencia es el tiempo necesario para que un paquete de información viaje desde la fuente hasta su destino. En los resultados notamos que el tiempo de latencia varía acorde al tamaño del mensaje transmitido al formar diferentes colas de transmisión.

6.5.3 ACELERACIÓN Y EFICIENCIA:

Para evaluar estos indicadores se explican 2 muestras representativas.

Primera muestra representativa: Código en serie y paralelo que produce la siguiente información:

METODO WOLF

Procesadores	Serial(s.)	Paralelo (s.)
12	14	11
17	14	9
21	14	9
25	14	9
27	14	10
31	14	10
33	14	14

METODO EWALD

Procesadores	Serial(s.)	Paralelo (s.)
12	58	69
17	58	56
21	58	44
25	58	43
27	58	43
31	58	42
33	58	62

Para evaluar la anterior información se determina la aceleración, considerada como la razón del tiempo que tarda el programa secuencial y el tiempo que tarda el correspondiente algoritmo paralelo en ser ejecutado en el mismo computador utilizando *p* procesadores. El valor resultante en Wolf es:

	P=12	P=17	P=21	P=25	P=27	P=31	P=33
Aceleración	1.27	1.55	1.55	***1.55***	1.4	1.4	1

Para Ewald se obtuvo:

	P=12	P=17	P=21	P=25	P=27	P=31	P=33
Aceleración	0.84	1.03	1.31	1.34	1.34	***1.38***	0.93

Otro indicador es la aceleración en términos de porcentaje de código paralelizado, calculado a través de la *ley de ahmdal* para la cual siendo T el tiempo del algoritmo secuencial, si f es la fracción de operaciones del algoritmo que hay que ejecutar secuencialmente ($0 < f < 1$), entonces la aceleración máxima obtenible ejecutando el programa en p procesadores es:

$$Ap = \frac{T}{T\left(f + (1-f)/p\right)} = \frac{1}{\left(f + (1-f)/p\right)}$$

Cuando f -> 0, entonces Ap= p. *Este límite nunca es obtenible debido a que inevitablemente parte del algoritmo es secuencial y la comunicación y sincronización de procesos consume tiempo*. Aplicando la ecuación obtenemos para Wolf el 37% y para Ewald el 27% (*Ley de Ahmdal*).

La eficiencia en términos de porcentaje de código paralelizable y número de procesadores utilizados es la aceleración dividida por el número de procesadores *p.* En un mismo porcentaje de código paralelizable el aumento en el número de procesadores produce una reducción de este parámetro (ver tabla).

(1-f)%	***P=1***	***P=2***	***P=4***	***P=8***	***P=16***	***P=32***	***P=64***	***P=128***
0	1.00	0.50	0.25	0.13	0.06	0.03	0.02	0.01
10	1.00	0.53	0.27	0.14	0.07	0.03	0.02	0.01
30	1.00	0.59	0.32	0.17	0.09	0.04	0.02	0.01
50	1.00	0.67	0.40	0.22	0.12	0.06	0.03	0.02
70	1.00	0.77	0.53	0.32	0.18	0.10	0.05	0.03
80	1.00	0.83	0.63	0.42	0.25	0.14	0.07	0.04
90	1.00	0.91	0.77	0.59	0.40	0.24	0.14	0.07
100	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

La eficiencia en Wolf es 0.062 y en Ewald es 0.044.

Segunda muestra representativa: se realiza para 27678 átomos y 10 iteraciones produciendo los siguientes datos:

METODO WOLF

Procesadores	Serial(s.)	Paralelo (s.)
13	634	530
19	634	455
25	634	369
32	634	291
35	634	302
37	634	380
40	634	480

METODO EWALD

Procesadores	Serial(s.)	Paralelo (s.)
13	2538	3000
19	2538	2400
25	2538	2280
32	2538	1790
35	2538	1605
37	2538	1580
40	2538	2612

La aceleración resultante en Wolf es:

	P=13	P=19	P=25	P=32	P=35	P=37	P=40
Aceleración	1.19	1.39	1.71	***2.18***	2.10	1.66	1.32

Para Ewald se obtiene:

	P=13	P=19	P=25	P=32	P=35	P=37	P=40
Aceleración	0.85	1.06	1.11	*1.41*	1.58	**1.61**	0.97

En los resultados de las muestras se observa que la aceleración aumenta con el incremento en el número de átomos, debido al aumento de la diferencia en tiempo serial Vs tiempos paralelos. Al cambiar la aceleración varían los siguientes indicadores.

La aceleración en términos de porcentaje de código paralelizado para 32 procesadores en Wolf es de 56% $\frac{1}{(0.44 + (1 - 0.44)/32)} = 2.18$. Entre tanto en Ewald se obtiene el 40% (*Ley de Ahmdal*). Este resultado es verificado con la *ley de Gustafon* en la cual la proporción de código paralelo crece al aumentar el tamaño del problema.

Para esta muestra la eficiencia en el método de Wolf es 0.068 y con Ewald 0.043.

6.5.4 COSTO DE PARALELIZACION

Para el cual se asume que el código sea 100% paralelizable y por lo tanto la aceleración es A*p* = *p* despreciando el tiempo de comunicación entre procesadores. Considerando el tiempo de comunicación o latencia *T*, la

aceleración decrece aproximadamente a: $Ap = \frac{T}{(T(\sqrt{P + Tc}))} < P$ Y para que la aceleración no sea afectada por la latencia de comunicación necesitamos que: $\frac{T}{(P)} \gg Tc \rightarrow P \ll \frac{T}{(Tc)}$

Esto significa que a medida que se divide el problema en partes más y más pequeñas para poder correr el problema en más procesadores, *llega un momento en que el costo de comunicación Tc se hace muy significativo y desacelera el cómputo*. El costo de paralelización es 0.17059.

6.5.5 RAPIDEZ ENTRE EWALD Y WOLF

Para obtener este parámetro se compara la muestra representativa paralelizada entre Ewald Y Wolf con 2160 átomos.

Para las iteraciones centrales se produce:

Procesad.	Ewald (seg)	Wolf (seg)	Diferencia (veces)
17	5.1	0.8	6.3
21	3.8	0.8	4.75
25	3.7	0.8	4.62
27	3.7	1	3.7
31	3.7	1	3.7
33	5.2	1.3	4

La diferencia promedio en las iteraciones *paralelas centrales* es de *4.5* veces más rápida en Wolf con respecto a Ewald. En el programa en *serie* Ewald necesita de un tiempo de 5.3 seg. y en Wolf 1.3 seg., produciendo una diferencia de *4.07* veces.

Con respecto a la primera iteración (donde se realizan más cálculos matemáticos que en las demás iteraciones) se obtiene:

Procesad.	Ewald (seg)	Wolf (seg)	Diferencia (veces)
17	9.9	2	4.95
21	9.5	1.8	5.27
25	9	0.8	11.25
27	9	1	9
31	8	1	8
33	15.3	2	7.65

La diferencia promedio en la *primera iteración* es de *7.6* veces más rápido en Wolf con respecto a Ewald. Entre tanto en *serie* Ewald requiere de un tiempo de 11 seg. y Wolf de 2.4 seg., generando una diferencia de *4.5* veces.

Se puede observar un rendimiento superior a 4 veces del método de Wolf con respecto a Ewald, estos resultados del programa en paralelo son similares al programa en serie.

CONCLUSIONES

- La paralelización del código que calcula las fuerzas permite la duplicación de la aceleración con respecto al código serie.

- La implementación de soluciones utilizando la Interfaz de pasos de mensajes MPI genera diferentes tiempos de respuesta según el tipo de comunicación que se utilice.

- En la codificación de un programa en paralelo es necesario crear el código en serie para comparar si el tiempo que dicho proceso produce es menor que la solución en serie.

- Los mejores resultados en tiempos de respuesta para el programa dizzy en paralelo se obtienen cuando el dominio es muy grande y el número de procesadores es determinado por la cantidad de mensajes que se producen durante su ejecución.

- La aceleración generada por el sistema dizzy cumple con los lineamientos de la ley de Amhdal y Gustafon.

- En el sistema dizzy el rendimiento del método de Wolf es superior a 4 veces con respecto al metodo de Ewald. Los valores obtenidos en la fuerza, la temperatura, la presión, energía total y otras variables termodinámicas de interés son similares.

RECOMENDACIONES

- El sistema Dizzy puede continuar su crecimiento a través del desarrollo en forma organizada de módulos de librerías.

- Para ejecutar el programa Dizzy en paralelo y obtener un menor tiempo de respuesta, se recomienda utilizar el cluster cuando exista una mínima carga de tareas para los procesadores, para ello se debe verificar la cola de tareas.

- Para la disminución en los costos se recomienda la utilización del sistema dizzy bajo plataformas de software libre.

BIBLIOGRAFÍA

1. N.Lane, National Nanotechnology Initiative, National Science and Technology Council, 2000, <http://www,nsf.gov/home/crssprgrm/nano/nani.pdf>.
S. M. Auerbach, K. A. Carrado, and P. K. Prabir, editors, Marcel Dekker, Inc., New York, 2003.
2. J. M. Newsam, Zeolites, in Solid State Chemistry: Compounds, edited by A. K. Cheetham and P. Day, pags 234-280, Oxford University Press, Oxford, 1992.
3. S. M. Kuzniki, K. A. Thrush, and H. M. Garfinkel, 1993, US Patent (pending).
4. P. S. Zurer, Chem. and Eng. News , 1996, pags 74, 29.
5. C.T. Kresge, M. E. Leonowicz, W. J. Roth, J. C. Vartuli, and J. S. Beck, Nature 1992, pags 359, 710.
6. C. C. Freyhardt et al., Nature,1996, pags 381, 295.
7. M. C. Lovallo and M. Tsapatsis, Aiche Journal,1996, pags 42, 3020.
8. S. Nair and M. Tsapatsis, J. Phys. Chem B ,2000, pags 104, 8982.
9. M. Cook and W. C. Conner, How big are the pores of zeolites? , in Procedings of the 12th International Zeolite Conference, edited by M.M. J. Treacy, B.K. Marcus, M. E. Bisher, and J. B. Higgins, pags 409-414, Material Research Society, Warrendale, PA, 1999.
10. F. J. Keil, Chemische Technik 1994, Pags 46, 7.
11. D. Frenkel and B. Smit, Understanding Molecular Simulation, Academic Press, San Diego.
12. S. M. Auerbach, Int. Rev. Phys. Chem. 2000, Pags 19,155.
13. C. Blanco and S. M. Auerbach, J. Phys. Chem. B , 2003, pags 107, 2490.
14. M. P. Allen and D. J. Tildesley , Computer Simulation of Liquid, Oxford Science Publications, Oxford ,1997.
15. D.C. Rapaport, The Art of Molecular Dynamics Simulation, Cambridge University Press, 2002.
16. Ewald, P.P. Ann. Phys.(Leipzig), 1921, pags 64, 253.
17. Wolf, D.; Keblinski, P.; Phillpot, S.R. and Eggebrecht, J.; *J. Chem Phys.*, 1999, pags 110, 8254.
18. Theoretical and Computational Biophysics Group, <http://www.ks.uiuc.edu/Research/namd/> , 2005.
19. Official MPI standards documents <http://www.mpi-forum.org/docs/mpi-20-tml/mpi2-report.html>.
20. National Laboratory Argonne, MPI Documentation, <http://www-unix.mcs.anl.gov/
21. mpi/tutorial/gropp/talk.html>
22. Maui High Performance Computing Center, Parallel Programming Workshop MPI, <http://www.mhpcc.edu/training/workshop/mpi/MAIN.html>
23. Blaise Baney, Tutorial OpenMP,

<http://www.llnl.gov/computing/tutorials/openMP>.
24. Cluster Beowulf - Implementaciones en C++ y Fortran, <http://twiki.im.ufba.br/pub/Main/TiagoVaz/apresentacao_beowulf_1.0.pdf>
25. GRUPO PACS, MPI course Introduction, Universidad de Illinois, 2006.
26. GRUPO PACS, MPI course Intermediate, Universidad de Illinois, 2006.
27. C. Blanco, Modeling Equilibrium and nonequilibrium dinamics in Zeolites and zeolites-guest systems, Umass, 2004.
28. C. Blanco, C Saravanan, M. Allen, and S. M. Auerbach, J. Chem. Phys. 2000, pags 113, 9778.
29. H. Herbert, Introducción a la Computación Paralela, Universidad de los Andes, 2000.
30. P. Guillén, Introducción a la programación paralela, Cenatav,cuba,2005.
31. C. J. Hernandez, J. C. Casallas, Lineamientos para la construcción de programas de procesamiento en paralelo, proyecto Cumulus,UIS, 2002.
32. I. Llorente, Computación Científica sobre Sistemas Multiprocesador, Universidad Complutense de Madrid, 2001.
33. D. Bertsekas, J. Tsitsiklis, Parallel and Distribuited computation Numerical Methods, Prentice Hall, 2002.