UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

EVALUACIÓN DE CLUSTERS TIPO BEOWULF CON BASE AL PARADIGMA DE FARMING

T E S I S
QUE PARA OBTENER EL TÍTULO DE
INGENIERO ELÉCTRICO ELECTRÓNICO
ÁREA : COMUNICACIONES
P R E S E N T A

LUIS FRANCISCO GARCÍA JIMÉNEZ

Y PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN
P R E S E N T A

ANGEL RAÚL MONROY JIMÉNEZ

DIRECTOR: ING. FRANCISCO JAVIER CÁRDENAS FLORES

México, D.F. Mayo,2005

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Agradecemos el apoyo brindado a cada una de las personas que laboran en el Departamento de Ingeniería de Sistemas Computacionales y Automatización, del Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas de la UNAM.

Especialmente el apoyo y confianza brindada por la Dra. Lucía Medina Gómez por contribuir en la realización de esta tesis, así como al Ing. Francisco Javier Cárdenas Flores, ya que sin la ayuda de ambos este proyecto no se hubiera realizado.

Asimismo, las aportaciones y recomendaciones que surgieron durante la revisión de este trabajo para mejorarlo por parte de: Dra. Katya Rodríguez Vázquez, M. en C. Jorge L. Ortega Arjona, Dr. Héctor Benítez Pérez y al M. en I. Roberto Tovar Medina.

Índice General

1 Introducción. **9**

1.1 Objetivo General. 10
1.2 Objetivos Particulares. 10
1.3 Metodología. 10
1.4 Descripción del Contenido. 11

2 Introducción a los Sistemas Paralelos. **13**

2.1 Introducción. 13
2.1.1 Alcance de los Sistemas Paralelos. 14
2.2 Aspectos del Diseño en Sistemas Paralelos. 14
2.3 Modelos de Sistemas Paralelos. 15
2.3.1 Taxonomía de las Arquitecturas Paralelas. 16
2.4 Topologías de Red para Multicomputadoras. 21
2.4.1 Topología de Estrella. 21
2.4.2 Topología de Malla 2D. 22
2.5 Cluster Beowulf. 23
2.6 Programación Paralela. 24
2.6.1 Paso de Mensajes. 24
2.6.2 Paradigmas. 25
2.7 Medición de Desempeño en los Sistemas Paralelos. 28

3 Casos de Estudio para evaluar los sistemas Beowulf. **31**

3.1 Caracterización de la Distribución de Presión Acústica. 31
3.2 Método de la Respuesta al Impulso. 32
3.2.1 Teoría Acústica. 32
3.2.2 Condiciones de Frontera. 34
3.2.3 Respuesta al Impulso para un Transductor Circular. 34
3.3 Método Basado en el Principio de Huygens. 36
3.4 Medición de la Presión Acústica. 38

4 De Secuencial a Paralelo. **39**

4.1 Simulación de la Respuesta al Impulso Secuencial. 39
4.1.1 Resultados de la Respuesta al Impulso Secuencial. 40
4.1.2 Paralelización de la Respuesta al Impulso. 41
4.2 Simulación del Método de Huygens Secuencial. 43
4.2.1 Resultados del Método de Huygens Secuencial. 43

4.2.2 Paralelización del Método de Huygens. 45

5 Pruebas de Desempeño. 47
5.1 Automatización de Pruebas de Desempeño. 47
5.2 Punto Óptimo. 48
5.2.1 Determinación del Número Óptimo de Trabajadores. 48
5.2.2 Comparación de Clusters. 52
5.3 Métricas. 53
5.4 Análisis de Resultados. 54

6 Conclusiones. 59
6.1 Trabajos Futuros. 60

A Sistemas Digitales 61
A.1 Sistemas lineales invariantes con el tiempo 61

B Ondas Mecánicas 63
B.1 Superposición 63
B.2 Interferencia 64
B.3 Difracción 64

C Cluster Beowulf. 65
C.1 Socket. 65
C.2 Makefile 65
C.3 PyFarm. 66

D Programas. 69
D.1 Respuesta al Impulso Secuencial. 69
D.2 Respuesta al Impulso Paralelo. 71
D.3 Función baffpar 72
D.4 Huygens Secuencial. 74
D.5 Huygens Paralelo 75
D.6 Mensaje en Huygens 76
D.7 Clase Farmer 77
D.8 Makefile 79
D.9 Hace Makefile 79
D.10Parámetros 80

E Tablas de las Pruebas. 81

Índice de Figuras

2.1 Evolución de una computadora serial . . . 15
2.2 Multiprocesadores . . . 17
2.3 Memoria compartida con múltiples módulos . . . 18
2.4 Multicomputadora . . . 19
2.5 Formato de un Mensaje . . . 20
2.6 Topología estrella . . . 22
2.7 Topología Malla-2D . . . 23
2.8 Paradigma de Farming. . . . 26

3.1 Campo cercano y lejano de un transductor (T/R). . . . 32
3.2 La proyección de **r**: a) $r_{xy} \leq a$ y b) $r_{xy} > a$ . . . 35
3.3 Principio de Huygens. . . . 37
3.4 Arreglo de Transductores. . . . 37

4.1 Velocidad del pistón obtenida experimentalmente. . . . 39
4.2 Distribución de presión acústica . . . 40
4.3 Plano central de la distribución de presión acústica . . . 41
4.4 Presión acústica para diferentes planos . . . 43
4.5 Distribución Volumétrica . . . 44
4.6 Plano central de la presión acústica en Huygens . . . 44
4.7 Planos de la distribución volumétrica en Huygens . . . 45
4.8 Distribución volumétrica en Huygens . . . 46

5.1 Latencias de la tecnología SCI y ethernet. . . . 49
5.2 Respuesta al impulso y Huygens en Onomatopoeiia . . . 50
5.3 Respuesta al impulso ethernet VS SCI en Cuaco . . . 51
5.4 Trabajadores para Huygens en Cuaco . . . 52
5.5 Huygens y respuesta al impulso para ambos clusters . . . 53
5.6 Respuesta al Impulso para eficiencia y speed-up en Cuaco . . . 54
5.7 Huygens para eficiencia y speed-up en Cuaco . . . 55
5.8 Respuesta al Impulso para eficiencia y speed-up en Onomatopoeiia 55
5.9 Huygens para eficiencia y speed-up en Onomatopoeiia . . . 55
5.10 Respuesta al impulso para eficiencia y speed-up en Onomatopoeiia y Cuaco . . . 56
5.11 Tiempo serial y paralelo en un algoritmo. . . . 56

A.1 Sistema digital. . . . 61

Índice de Tablas

3.1 Límites numéricos en la evaluación de $h(\mathbf{r}, t)$ 35

E.1 Latencias para ethernet y SCI 82
E.2 Huygens ethernet VS SCI con desviación estándar 82
E.3 Impulso ethernet VS SCI con desviación estándar 83
E.4 Huygens comparación de clusters 83
E.5 Respuesta al impulso comparación clusters 84
E.6 Huygens para ethernet en Cuaco 84
E.7 Huygens para SCI en Cuaco 84
E.8 Respuesta al Impulso para ethernet en Cuaco 84
E.9 Respuesta al Impulso para SCI en Cuaco 85
E.10 Respuesta al Impulso en Onomatopoeiia 85
E.11 Huygens en Onomatopoeiia 85
E.12 Impulso en Onomatopoeiia para 4 procesadores 85

Capítulo 1

Introducción.

El cómputo paralelo tiene un papel central en muchas aplicaciones, que tienen una alta demanda computacional, tales como la predicción meteorológica, modelos económicos-financieros, algoritmos de optimización, señales e imágenes biomédicas, robótica, etc. Algunos de estos problemas pueden requerir días de cómputo (o meses) en una computadora que cuente con un único procesador, pero podrían realizarse en fracciones de hora en una arquitectura paralela.

Un tipo de sistema paralelo es el *cluster*, que nace de la idea de conectar varias computadoras de escritorio con procesadores *Pentium Dx4* a través de una red ethernet. De tal manera que se logra una computadora de alto desempeño a bajo costo. Por ello la rápida divulgación de estos sistemas en varias ramas científicas e ingenieriles. En este trabajo se evalúan dos *clusters* de tipo *Beowulf*.

El procesamiento paralelo es una tecnología que se está desarrollando rápidamente. Una de las razones principales que ha impulsado este desarrollo, es la motivación de crear algoritmos capaces de mejorar el desempeño en cuanto a tiempo de ejecución a través de lenguajes y/o herramientas. Una manera de usar estos sistemas es en el campo de diseño, al generar simulaciones que permitan reducir tiempos y costos. La caracterización de transductores ultrasónicos, es un ejemplo de este tipo de simulaciones que demandan gran poder de cálculo computacional y son importantes, ya que tienen aplicaciones en varios campos de investigación científica y de ingeniería.

La caracterización de transductores ultrasónicos permite conocer la distribución de las presiones acústicas en el medio de propagación, así como las regiones de máxima amplitud que están relacionadas con la concentración de energía.

El conocimiento detallado del patrón de radiación de los transductores ayuda a diferenciar el área de aplicación de un transductor, encontrar óptimamente los cambios que este patrón sufre cuando existen inhomogeneidades en el medio de propagación o bien calibrar los transductores.

Ejemplos de aplicación del ultrasonido sobran, se mencionan algunos y por qué son importantes. Supóngase una empresa que realiza pistones. Al fundir el acero para su construcción estos deben de tener un máximo de burbujas dentro del material. Cada burbuja no debe sobrepasar un cierto diámetro de

seguridad, ya que si es rebasado podría romperse o fracturarse en condiciones de uso extremo. Analizar su estructura y observar sus fallas de construcción proporciona información para saber si el pistón es útil o no. El ultrasonido ayuda a realizar dicho análisis y *el dispositivo que genera la onda ultrasónica debe ser caracterizado*.

Otro ejemplo podría ser el análisis en tejidos biológicos, donde una potencia mayor (*generada por el transductor*) de la que el tejido soporta, dañaría al mismo. Así se podrían mencionar varias aplicaciones que tiene el ultrasonido en muchos tipos de sistemas pero se deja a la imaginación del lector. . .

1.1 Objetivo General.

Analizar el desempeño de dos *clusters Beowulf* con diferente construcción y topologías de red, a través del uso de dos experimentos numéricos para caracterizar la distribución de presión acústica de un transductor ultrasónico.

1.2 Objetivos Particulares.

- A través del uso del paradigma Farming pasar del programa secuencial al paralelo.
- Comparar diferentes tecnologías de comunicación de redes.
- Usar métricas de desempeño de sistemas paralelos, para encontrar el punto óptimo en la ejecución de las simulaciones.
- Comparar los *clusters*.

1.3 Metodología.

Para lograr el objetivo general de la tesis, se parte de dos métodos que caracterizan la distribución de presión acústica de un transductor ultrasónico (benchmark). El primero de ellos es el método de la respuesta al impulso, que consiste en excitar al transductor con una señal ultrasónica, obteniéndose como respuesta el movimiento del transductor. El segundo, utiliza el principio de *Huygens*, para conocer el frente de onda secundario a partir del frente de onda primario ya conocido. Con ello se puede obtener la distribución de presión acústica.

Una vez implementados estos dos métodos secuencialmente, se modifican dividiéndose en procesos iguales e independientes que se pueden ejecutar simultaneamente, utilizándose el paradigma de *Farming*.

Posteriormente se toman métricas de desempeño como la *eficiencia*, *speed-up y fracción serial*, que permiten analizar el desempeño de los *Clusters Beowulf* y obtener conclusiones.

1.4 Descripción del Contenido.

- En el capítulo 2 se presenta una introducción a los sistemas paralelos.
- En el capítulo 3 se estudia la base físico-matemática para la caracterización de la presión acústica en transductores ultrasónicos como casos de uso para evaluar los sistemas paralelos.
- En el capítulo 4 se implementan los métodos estudiados en los capítulos 3 y 2 en forma secuencial y paralela. En el apéndice se muestran los programas implementados.
- En el capítulo 5 se realizan mediciones de desempeño de ambos sistemas paralelos.
- En el capítulo 6 se presentan el análisis de los resultados y las conclusiones obtenidas. Finalmente se proponen trabajos futuros asociados al tema de tesis.

Capítulo 2

Introducción a los Sistemas Paralelos.

2.1 Introducción.

Desde el nacimiento de las computadoras seriales, su velocidad ha sido incrementada de tal manera que han cubierto las aplicaciones más emergentes. Con la tecnología actual, es posible construir un sistema con miles de microcircuitos, donde cada circuito puede ejecutar millones de instrucciones por segundo (MIPS), para alcanzar una frecuencia total de miles de millones de instrucciones por segundo. Si un único procesador pudiera alcanzar 50000 MIPS, tendría que ejecutar una instrucción en $0.002\ ns$, recordando que el periodo es el inverso de la frecuencia. Ninguna máquina serial existente llega a acercarse a esta cifra, además de que consideraciones teóricas establecen improbable que alguna lo logre. La teoría de la relatividad de Einstein establece que nada puede viajar más rápido que la luz, que en el periodo en cuestión ($2\ ps$) cubriría una distancia de $0.6\ mm$. Desde el punto de vista práctico, una computadora de esa velocidad, generaría un calor tal que se fundiría. La tendencia reciente muestra que el desempeño de las computadoras seriales ha comenzado a saturarse. Una manera de eludir la saturación es usar un conjunto de procesadores realizando una tarea en común o sistema paralelo.

Es posible encontrar sistemas paralelos en la vida cotidiana, piénsese en una librería, en la que se debe ordenar un gran número de libros y sólo existe un bibliotecario para realizar dicha tarea. Llevar cada libro al lugar específico que le corresponde y regresar por otro y acomodarlo en su lugar, es una tarea sumamente tediosa, además tomaría demasiado tiempo. ¿Por qué no dividir la tarea con muchos bibliotecarios? Esto reduciría el tiempo, sin embargo los bibliotecarios (*trabajadores*) deben recorrer toda la biblioteca para organizar un libro y regresar por otro. Tal vez no es la mejor forma de optimizar la tarea. Así que ¿por qué no asignar un mismo número de libros a cada trabajador y además asignarle una área de trabajo, donde no deba moverse para acomodar los libros? Así cada libro que no pertenece a su área, sólo debe pasarlo al trabajador que le corresponde dicho libro, con una etiqueta que le diga al trabajador que ese libro le corresponde acomodarlo. De esta manera no inte-

rrumpe a los demás trabajadores. Sin duda es la mejor manera de optimizar una tarea tan común.

La idea anterior muestra como una tarea puede ser agilizada al dividirla en subtareas asignadas a diferentes trabajadores, donde estos cooperan para realizar la tarea. La semejanza con un sistema paralelo es: dividir una tarea entre trabajadores al asignarles un monto de libros es la forma de ***dividir procesos***, pasar los libros hacia diferentes trabajadores es un ejemplo de ***comunicación entre tareas o paso de mensajes*** [1].

2.1.1 Alcance de los Sistemas Paralelos.

El procesamiento paralelo ha tenido un gran impacto en muchas áreas de computación. Con el alto poder de cómputo que tienen estos sistemas, es posible asignarles aplicaciones que no se lograban con técnicas tradicionales en los sistemas seriales. Muchas aplicaciones como el monitoreo de la contaminación, predicción del clima, requieren que el tiempo de cálculo sea lo más rápido posible, ya que de otra manera no tendría sentido realizar la prueba.

Otras aplicaciones que pueden ser beneficiadas por los sistemas paralelos son el modelado de materiales semiconductores, modelado oceanográfico, diseño de vehículos, análisis biomédicos, exploración petrolera, consultas en bases de datos, etc. Muchas de las aplicaciones mencionadas son consideradas como problemas científicos o de ingeniería que tienen un impacto económico, y su solución eficiente sólo se puede realizar con técnicas de procesamiento en paralelo.

2.2 Aspectos del Diseño en Sistemas Paralelos.

Para el uso eficiente de un sistema paralelo debe tenerse en cuenta los siguientes puntos:

- **Diseño de computadoras paralelas**: Es importante el diseño, ya que éstos deben de ser escalables a un número grande de procesadores y capaces de soportar comunicación rápida de datos entre procesadores. Lo anterior es uno de los aspectos de mayor importancia en dichos sistemas.

- **Diseño eficiente en algoritmos**: Un sistema paralelo es poco productivo, a menos que cuente con un algoritmo eficiente, aprovechando al máximo los recursos.

- **Lenguajes de sistemas paralelos**: Los algoritmos paralelos deben construirse usando lenguajes lo suficientemente flexibles que permitan implementaciones fáciles. Además se debe usar un paradigma que logre el éxito del mismo.

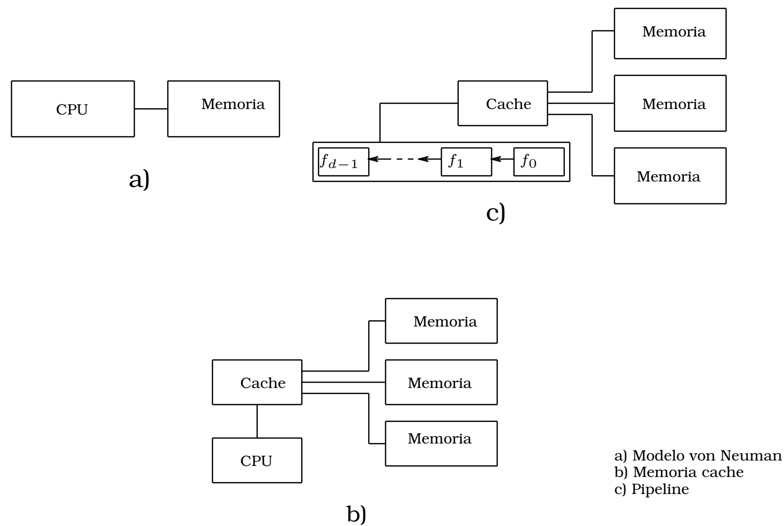


Figura 2.1: Evolución de una computadora serial

- **Portabilidad de programas paralelo**: La portabilidad es uno de los mayores problemas del cómputo paralelo. Típicamente, un programa escrito en un sistema paralelo requiere modificaciones excesivas para ser ejecutado en otro sistema paralelo. Ello es un punto que requiere mucha atención.

2.3 Modelos de Sistemas Paralelos.

2.3.0.1 La Máquina Clásica de von Neuman.

La computadora tradicional está basada en el modelo de von Neuman. Este modelo se divide en un CPU y memoria principal (Figura 2.1 a)). El CPU es dividido en una unidad de control y una unidad lógica aritmética (ALU). La memoria almacena instrucciones y datos. La unidad de control dirige la ejecución del programa, mientras la ALU efectúa los cálculos llamados en el programa. Cuando los datos y las instrucciones son usados por el programa, estos se almacenan en un localidad de memoria rápida, llamada **registros**. Una memoria rápida implica un mayor costo, por lo que existen pocos registros.

Los datos y las instrucciones se mueven dentro de la memoria y los registros en el CPU. La vía por donde pasan se llama **bus**, básicamente es una colección de cables paralelos que están unidos y algún hardware que controla el acceso al bus.

La máquina de von Neuman necesita algunos componentes adicionales que la hacen útil como: dispositivos de entrada y salida, dispositivos grandes de almacenamiento o disco duro.

El gran problema de este modelo es la transferencia de datos e instrucciones entre la memoria y el CPU. La velocidad de ejecución de los programas

está limita por el grado en el que puedan ser transferidos. Por ello se añadió una memoria intermedia, que es más rápida que la memoria principal y los registros, llamada memoria **cache** (Figura 2.1 b)). Otra forma de incrementar la eficiencia, es sobreponer la ejecución de una instrucción con la operación de "fetch" de la próxima instrucción a ser ejecutada, llamado **pipeline** de instrucciones (Figura 2.1 c)).

2.3.1 Taxonomía de las Arquitecturas Paralelas.

Existen varias maneras para construir sistemas paralelos. Algunas de estas formas son el mecanismo de control, organización de memoria, interconexión de red y granularidad de procesadores.

2.3.1.1 Mecanismo de Control.

La primera clasificación basada en el mecanismo de control, la realizó Flynn [2], quien en 1966 clasificó a los sistemas de acuerdo al número de secuencia de instrucciones y el número de secuencia de datos:

- **Single-Instruction Single-Data** (**SISD**): La máquina de von Neuman es un ejemplo de los sistemas SISD, ya que trabaja una única secuencia de datos y opera una única secuencia de instrucciones
- **Single-Instruction Multiple-Data** (**SIMD**): Se refiere a un modelo de computadora, que tiene un único componente encargado exclusivamente del control, y un número grande de ALUs, cada una con su propia memoria. Durante cada ciclo de instrucción, la unidad de control transmite una instrucción a todas sus ALUs, y todas al mismo tiempo realizan la instrucción o esperan la instrucción, lo que implica una completa sincronización en la ejecución de las declaraciones. En otras palabras, en un instante dado de tiempo, todas las instrucciones están "activas" y realizando exactamente el mismo proceso, o todas esperan alguna instrucción.
- **Multiple-Instruction Multiple-Data** (**MIMD**): Las unidades de procesamiento actúa independientemente, es decir, constan de su propia unidad de control y ALU. Los procesadores son autónomos y capaces de ejecutar sus tareas a su propia velocidad. En contraste con los sistemas **SIMD**, los **MIMD** son asíncronos. No hay un reloj global, y ningún procesador debe sincronizar con otro.

2.3.1.2 Organización de Memoria.

El uso de múltiples procesadores paralelos en un mismo sistema, necesita de nuevos requerimientos para su construcción. Para lograr que muchos procesadores puedan trabajar juntos sobre el mismo problema computacional, deben compartir datos y comunicarse uno con el otro. Existen dos tipos de

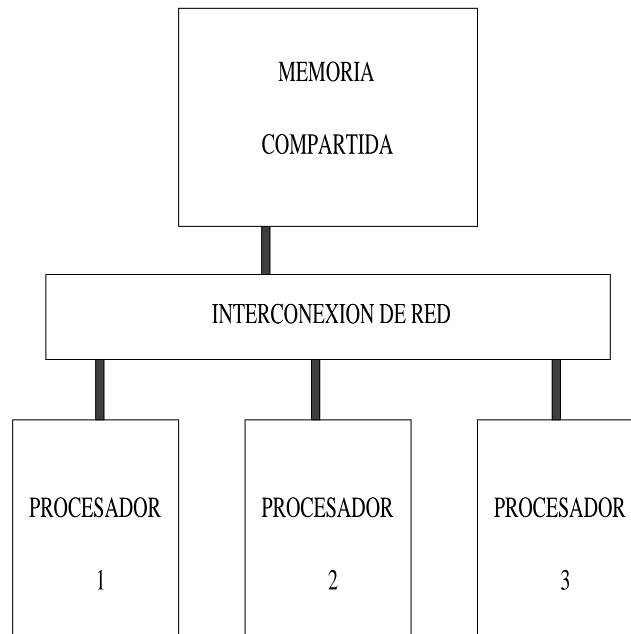


Figura 2.2: Multiprocesadores

arquitectura que aprovechan al máximo estos requerimientos: la arquitectura de *memoria compartida* y *arquitectura de memoria distribuida* [1]. En la arquitectura de memoria compartida o multiprocesadores, todas los procesadores tienen acceso a un banco común de memoria a través de una interconexión de red, que permite compartir los mismos datos y estructuras almacenadas en la memoria. En la arquitectura de memoria distribuida o multicomputadora cada procesador tiene su propia memoria local y comparte datos con paso de mensajes [3] a través de una conexión de red.

- **Multiprocesadores**:

  Los procesadores pueden trabajar en paralelo y pueden acceder a un banco central de memoria a través de una interconexión de red (ver Figura 2.2). La conexión de red es responsable del control entre peticiones simultáneas a memoria por varios procesadores, y asegurarse de que todos los procesadores tengan respuesta en el tiempo más corto posible. La actividad en cada uno de los procesadores es similar al comportamiento de una computadora serial. El desempeño de estos sistemas es el uso en paralelo de los procesadores, es decir, si el sistema tiene n procesadores, su desempeño será n veces comparado con un solo procesador. Cuando muchos procesadores quieren acceder a la memoria compartida en un período de tiempo muy pequeño, la memoria no puede acomodar todas las peticiones simultáneamente, y algunos procesadores deben esperar mientras otros son atendidos.

  Una forma de ayudar a reducir la saturación de peticiones a la memoria es con el uso de memoria cache. Si cada procesador tiene su memoria

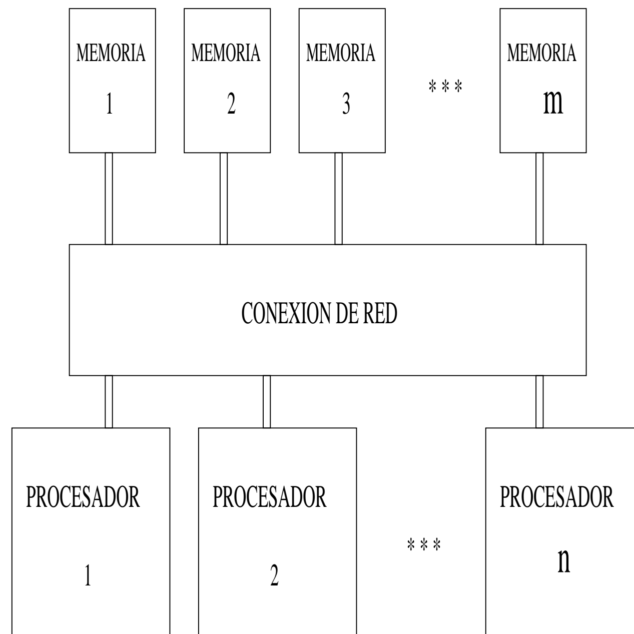


Figura 2.3: Memoria compartida con múltiples módulos

cache permite reducir las colisiones entre peticiones en la memoria.

Otra técnica para reducir las colisiones de peticiones a memoria, es dividir el banco de memoria en pequeños módulos, donde los procesadores puedan acceder a su propio módulo. Esto reduce la probabilidad de peticiones simultáneas al mismo módulo de memoria. La Figura 2.3, muestra la organización de la arquitectura multiprocesadores con múltiples módulos de memoria compartida. Un punto importante, que no debe olvidarse, es que la división de memoria es invisible para el programador.

- **Multicomputadoras**:

  Otra forma de reducir la colisión de petición a memoria en un sistema paralelo es eliminar la memoria compartida y cambiarla por memoria local para cada procesador. La comunicación se debe hacer a través de una red y por paso de mensajes. Este tipo de computadora de memoria distribuida se muestra en la Figura 2.4. Los procesadores deben ser capaces de ejecutar procesos usando solamente su memoria local. Todos los procesadores pueden recibir o enviar datos a través de una interconexión de red con paso de mensajes.

  La organización básica de estos sistemas es completamente diferente a la de los multiprocesadores, ya que requiere un modelo diferente de programación. Cuando algún procesador quiere intercambiar datos, únicamente lo puede hacer por paso de mensajes. Un procesador tiene únicamente acceso a su memoria local y no a la memoria de otro procesador. Esto no implica que los datos no pueden ser intercambiado entre procesadores.

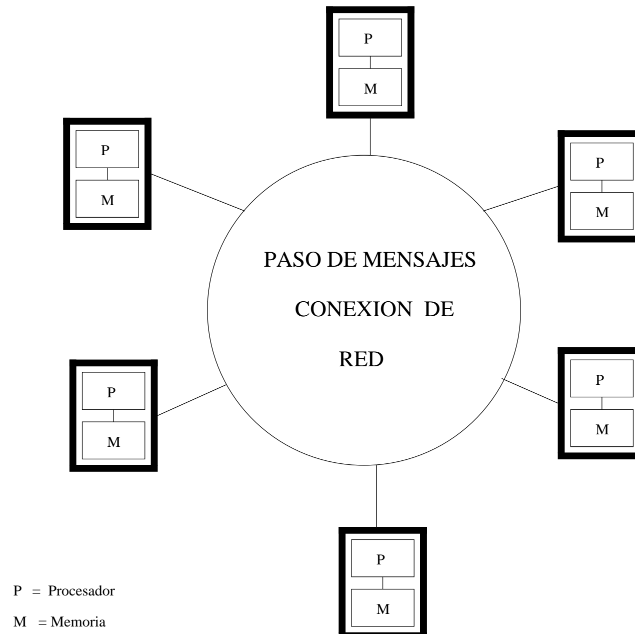


Figura 2.4: Multicomputadora

2.3.1.3 Interconexión de Red.

La forma de comunicación o intercambio de datos en una multicomputadora y en un multiprocesador, se logra con una interconexión de red capaz de comunicar un procesador con otro. Esta interconexión de red debe enviar y recibir datos (*bidireccional*). La conexión de red no es más que la interfaz eléctrica por donde los procesadores intercambian datos, o trasmiten bits de un lado a otro. Es fácil pensar que transmitir datos de un punto a otro toma tiempo, ya que el canal de comunicación tiene un ancho de banda, el cual es el número máximo de datos que pueden ser enviados en una unidad de tiempo. El formato general de un paquete de datos o mensaje se muestra en la Figura 2.5, donde la interfaz que recibe el mensaje, examina el número de procesador destinatario. Cada interfaz conoce su número de procesador y su posición relativa con respecto a los demás procesadores. Si el destino del mensaje pertenece a su procesador, la interfaz de comunicación almacenará el mensaje en memoria y enviará un mensaje de recibido hacia el procesador. Si el mensaje pertenece a otro procesador, la interfaz deberá escoger una vía de conexión que dirija el mensaje al vecino correcto.

Existe una variedad de algoritmos de direccionamiento que son usados por la interfaz como vía de conexión para enviar los datos entre los procesadores distantes. Las dos categorías generales son el direccionamiento *estático* y *dinámico* [4] [1].

El direccionamiento *estático*, se refiere a la conexión entre un par de procesadores cuando usan la misma ruta. Puede escribirse una tabla de direccionamiento con todas las rutas al inicio de la configuración de la interfaz. Siempre que un mensaje sea enviado, solamente debe leerse el encabezado del mensaje

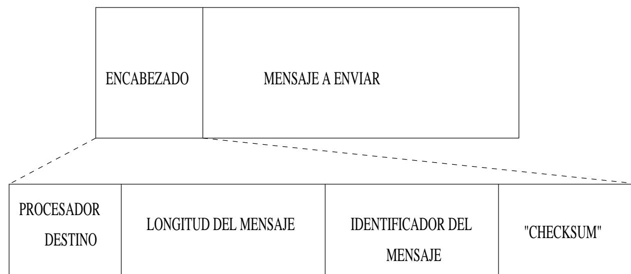


Figura 2.5: Formato de un Mensaje

con el número de procesador, y la ruta está trazada para cualquier número de peticiones, ya que las tablas de direccionamiento nunca cambian. En cambio los direccionamientos *dinámicos* cambian continuamente basándose en la información del congestionamiento o tráfico en la red. Una desventaja de los direccionamientos dinámicos es que un par de mensajes enviados o más, por la misma fuente, pueden llegar en diferente orden del que fueron enviados. Esto puede ser corregido por el procesador que recibe el mensaje, pero requiere de tiempo adicional y memoria para acomodarlo.

El tiempo que le toma a los datos pasar de un procesador a otro, se llama *latencia* y es la suma de tres elementos: *tiempo de transmisión*, *tiempo de procesamiento* y *tiempo de espera* [1]. El *tiempo de espera* es debido al retardo por tráfico y colisiones en el canal de comunicación. El *tiempo de transmisión* es un retardo provocado por la trasmisión física de los bits del mensaje viajando a través de la red. El último, se refiere a un componente de comunicación que debe ejecutar algunos cálculos con el fin de mejorar la comunicación. Los cálculos incluyen detección de errores y decisiones de direccionamiento basado en la fuente y destino entre procesadores.

Dos son las interfaces de red que se ocupan en este proyecto y se describen a continuación:

- Ethernet [5]
  - Soporte en Linux
  - Protocolo de comunicación: “Carrier Sense Multiple Acces”(CSMA, ANSI/IEEE 802.3)
  - Ancho de banda de 100Mb/s
  - Latencia mínima de 80 μs
- “Scalable Coherent Interconnect”(SCI) [5]
  - Soporte en Linux
  - Protocolo de comunicación: ANSI/IEEE 1596-1992
  - Ancho de banda de 4000Mb/s
  - Latencia mínima de 2.7 μs

2.3.1.4 Granularidad de Procesadores.

Los sistemas paralelos pueden ser construidos, ya sea de un número pequeño de procesadores muy potentes o un número muy grande de procesadores de bajo poder. Esta clasificación se puede dividir en tres:

- **Grano Grueso**: Pocos procesadores con un alto nivel de instrucciones de punto flotante por segundo.
- **Grano Medio**: Algunos procesadores con un nivel medio de instrucciones de punto flotante por segundo.
- **Grano Fino**: Muchos procesadores con bajo nivel de instrucciones por segundo.

2.4 Topologías de Red para Multicomputadoras.

En todos los tipos de arquitecturas, existe un costo/desempeño en el intercambio de datos entre la red de comunicación y los procesadores. El patrón de conexión entre los diferentes dispositivos de una red se llama topología de red. Un parámetro importante en la topología es el número de canales incidentes en cada interfaz llamado *conectividad* [1]. Es un factor importante en el costo de la red. Otro parámetro importante es el *diámetro* de la topología, definido como el número máximo de canales para transmitir un mensaje entre los procesadores mas lejanos. El diámetro es un punto importante en el desempeño de la red.

Las topologías simples tienen una conectividad baja y por tanto un alto diámetro. Las topologías más complejas tienen una alta conectividad y un diámetro bajo. Las diferentes topologías representan un costo y desempeño importante en el diseño y en la programación de algoritmos paralelos. Por lo que el programador debe estar familiarizado con las topologías y su comportamiento. Cada topología tiene su propio desempeño y el algoritmo paralelo debe ser apropiado para la topología en particular, con la finalidad de conseguir un mejor desempeño. A continuación se describen dos topologías que se usan en el trabajo presente.

2.4.1 Topología de Estrella.

Sus principales características son:

Todas las estaciones de trabajo están conectadas mediante un enlace bidireccional a un punto central ("Switch") (ver Figura 2.6). El "Switch" asume las funciones de gestión y control de las comunicaciones, proporcionando un camino entre dos nodos que quieran comunicarse. Cada vez que se quiere establecer comunicación entre dos nodos, la información transferida de uno hacia el otro debe pasar por el punto central. Si se rompe un cable sólo se pierde la conexión del nodo que interconectaba. Es fácil detectar y localizar

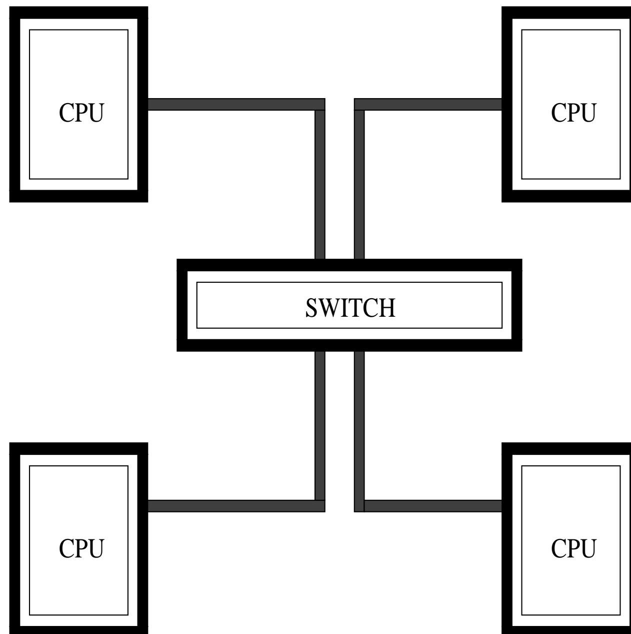


Figura 2.6: Topología estrella

un problema en la red. Si se asume que la distancia física entre algún nodo y el switch es la misma, el tiempo que le toma al mensaje desde su envío hasta la entrega a otro nodo es $2T$, para cualquier conexión entre los nodos. Su diámetro es de 2, debido a que es el número máximo de canales entre dos procesadores.

2.4.2 Topología de Malla 2D.

Al incrementar el número de comunicaciones entre los canales conectados hacia los procesadores, es posible reducir el diámetro de la red y el promedio de retardo en la red. Esta topología se muestra en Figura 2.7. Consiste en un arreglo de dos dimensiones de procesadores interconectados. Un procesador en el renglón i y columna j está conectado con cuatro de sus vecinos; hacia la izquierda, derecha, arriba y abajo, con las posiciones $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ y $(i, j+1)$ respectivamente [1]. Todas las líneas de comunicación son horizontales o verticales, no hay diagonales. Los límites de los procesadores son dos o tres procesadores a su alrededor. Si el retardo entre procesadores adyacentes es T, entonces el tiempo de comunicación es el camino que los une. La conectividad de esta topología siempre es de cuatro. Para una red de m por m, el diámetro es m.

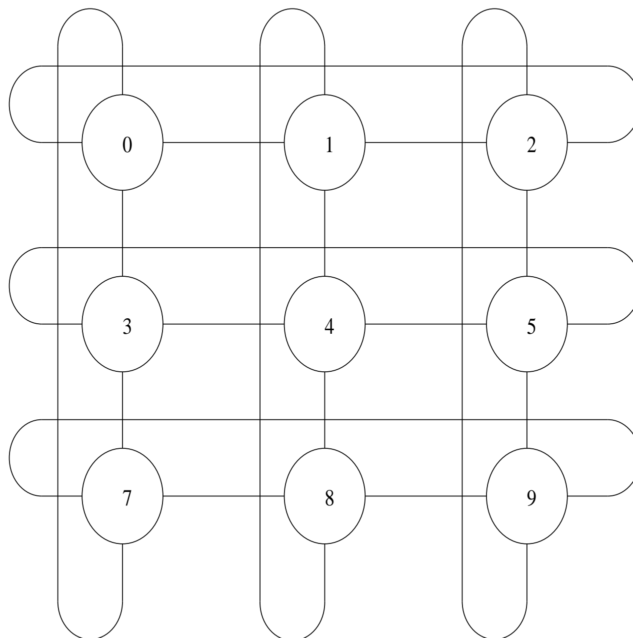


Figura 2.7: Topología Malla-2D

2.5 Cluster Beowulf.

El *cluster Beowulf* posee una arquitectura basada en multicomputadoras que puede ser utilizado para computación paralela. Este sistema consiste de varios nodos de procesamiento o máquinas conectadas a través de una topología de red y construido con componentes de hardware similares a cualquier PC capaz de ejecutar un sistema operativo como Linux, Free-BSD, RT-Linux o Lynx.

Una de las diferencias principales entre *Beowulf* y un *cluster* de *estaciones de trabajo* (COW, cluster of workstations) es el hecho de que el *Beowulf* se comporta más como una sola máquina, a diferencia con COW que son varias estaciones de trabajo, las cuales tienen monitores, teclados etc. En la mayoría de los Beowulf, sus nodos esclavos que no tienen monitores o teclados y son accedidos solamente vía remota o por terminal serial. Para leer un poco de historia de estos sistemas vea el apéndice C.

Existen diferentes formas de construir un *Beowulf*, se mencionan las dos más comunes:

- **Diskless**:

  Un nodo maestro controla el cluster entero y presta servicios de sistemas de archivos a los nodos esclavos vía Network File System (NFS) [6]. Es también la consola del cluster y la conexión hacia el mundo exterior. Las máquinas grandes de Beowulf pueden tener más de un nodo maestro, y otros nodos dedicados a diversas tareas específicas, como por ejemplo, consolas o estaciones de supervisión. En la mayoría de los casos los nodos esclavos son estaciones simples.

Los nodos son configurados por el nodo maestro, ya que éste les proporciona los servicios como DHCPD, FTPBOOT, RSH, PVM etc. Por ejemplo en esta configuración los nodos no conocen su dirección IP, donde están sus archivos, librerías, memoria virtual etc. hasta que el maestro los configura y carga su sistema operativo.

- **Nodo Disk**:

  Todos los nodos tienen disco duro, en donde se encuentra su sistema operativo. Así que solamente necesitan de una interconexión de red para lograr el paso de mensajes.

2.6 Programación Paralela.

Con el nacimiento de un sistema de multicomputadora como el *cluster Beowulf*, se requiere de nuevos contextos en software. Cualquier programa ejecutado en una computadora serial, ejecuta una secuencia de operaciones sobre el procesador en forma concurrente. Un programa paralelo debe proveer una secuencia de operaciones para cada procesador, incluyendo operaciones que coordinen e integren los procesos separados dentro de una ejecución coherente. En un problema específico, el algoritmo debe ser formulado de tal forma que produzca un flujo de operaciones y datos que puedan ser ejecutado en diferentes procesadores. Aunque los sistemas de multicomputadoras proveen de un enorme poder de cómputo a un razonable precio, se requieren lenguajes de programación y/o herramientas que permitan la implementación de algoritmos paralelos.

Para crear programas paralelos, dos herramientas son las más importantes:

- Paso de Mensajes.
- Paradigmas para implementar algoritmos paralelos.

2.6.1 Paso de Mensajes.

Cuando se programa bajo una arquitectura de multicomputadoras, el programador debe ver a su programa como un conjunto de procesos, donde cada uno de ellos tienen sus propias variables locales y además deben ser capaces de intercambiar datos hacia otros procesadores a través de *paso de mensajes* [3, 4]. El paso de mensajes permite el intercambio y la comunicación entre todos los procesadores que compongan la multicomputadora. Esta tarea se puede cubrir con la utilería llamada *Parallel Virtual Machine* (**PVM**).

A continuación se menciona lo más importante de PVM:

PVM es un paquete que engloba servicios del sistema operativo como intercomunicación de procesos y sockets (ver C.1) en el modelo de paso de mensajes. Funciona sobre un conjunto de máquinas que pueden ser heterogéneas

(*distintas arquitecturas*, como Sparc, PPC, Intel etc.), conectados en una o varias redes. Se divide en tres componentes:

1. El demonio (*pvmd3*): es un proceso encargado del control y funcionamiento de las tareas del usuario en la aplicación PVM y coordina las comunicaciones. Debe ser ejecutado en cada procesador que compone la multicomputadora o también llamada por PVM como *máquina virtual*. Mantiene una tabla de información de los procesos ejecutados en la máquina virtual. Los procesos de usuario se comunican unos con otros a través de los demonios, primero se comunican con el demonio local y luego éste manda o recibe mensajes de o hacia los demonios de los nodos remotos.

2. Las librerías (libpvm3.a, libfpvm3.a y libgpvm3.a)

   - libpvm3.a. Es una biblioteca en lenguaje C y son llamadas a funciones que incluye la aplicación paralela. Proporciona la capacidad de:
     - Iniciar y terminar procesos.
     - Empaquetar, enviar y recibir mensajes.
     - Sincronizar.
     - Cambiar la configuración de la máquina virtual paralela (número de procesadores que intervienen en el proceso, en el momento de inicializar la tarea).
   - libgpvm3.a. Se requiere para usar grupos dinámicos, dicho de otra manera, se usa para añadir o quitar procesadores de la máquina virtual sin que ésta pierda algún trabajo, mientras la tarea se ejecuta.
   - libfpvm3.a. Se utiliza en programas con lenguaje Fortran.

3. La última, es una consola gráfica diseñada para X [1], llamada *xpvm*. Actúa a modo de intérprete de comandos. Proporciona una interfaz simple entre el usuario y el demonio pvmd3. Permite configurar: el número de procesadores, inicializar procesos, monitorear la ejecución de los procesos en tiempo real, graficar desempeños, direccionar mensajes y cuantificar tiempos de proceso, espera y comunicación.

PVM opera en el nivel de tareas y proporciona una transferencia de envío de mensajes que el usuario percibe como un sistema unificado, por lo que el paso de mensajes sobre la máquina virtual es transparente al usuario.

2.6.2 Paradigmas.

Un *paradigma* es:

El conjunto de reglas y disposiciones que logran establecer o definir límites e indican cómo comportarse dentro de tales límites para tener éxito.

[1] Sistema de ventanas gráficas en UNIX

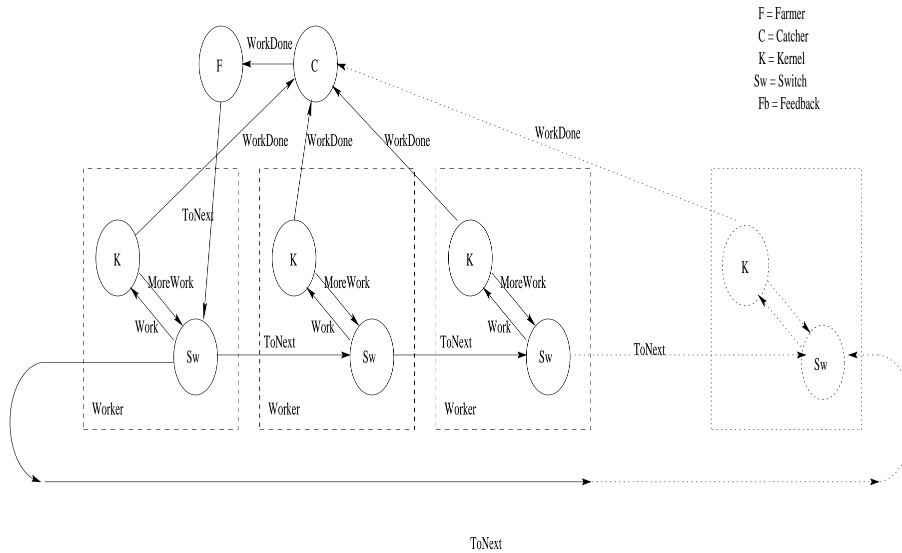


Figura 2.8: Paradigma de Farming.

Con el fin de obtener el mejor desempeño al paralelizar un programa se han clasificado tres tipos de paradigmas de la siguiente manera [3]:

- Algorítmico: Se usa para realizar tareas *quasi independientes*, las cuales se pueden ver como secciones del programa a paralelizar. Cada tarea *no debe ser idéntica a otra.*

- Geométrico: Se usa con el fin de procesar tareas *quasi independientes*. Estas deben de ser *idénticas y además deben de interactuar con sus tareas vecinas*, de acuerdo a la geometría del problema.

- Farming: Su objetivo es procesar tareas totalmente *independientes e idénticas*. Por lo que no importa el orden de su ejecución.

La implementación correcta de los paradigmas conlleva al desempeño del sistema. Por lo que uno de ellos tendrá un mejor desempeño en comparación con otro, dependiendo cual corresponde mejor en la transformación del algoritmo secuencial al paralelo y del lenguaje de programación. Para este trabajo el paradigma que mejor se adapta es *farming*, debido a que se realizará una paralelización de datos donde éstos son independientes unos con otros, a continuación se explica su funcionamiento.

Paradigma de Farming.

Este paradigma consta de tres elementos (Trabajadores **W**, Catcher **C** y Farmer **F**), que logran el control y envío de las tareas e intrínsecamente el balanceo de cargas [2]. La Figura 2.8 muestra el diagrama del paradigma y a continuación se explica:

El farmer, está encargado de inicializar a la línea de trabajadores, el catcher y de enviar las tareas independientes que se van a realizar. Dichas tareas son

[2]Asignarle el mismo monto de trabajo a cada procesador.

enviadas al primer trabajador de la línea de trabajadores. Cada trabajador consta de dos procesos: el switch y el kernel. El kernel es un proceso que ejecuta o realiza las tareas. Además en el momento que termine con una tarea, debe enviársela al catcher y avisar a su switch que terminó la tarea. El switch está capacitado para realizar tres operaciones:

1. Recibe una tarea y verifica si su kernel está ejecutando otra tarea, si el kernel no está ocupado, le manda la tarea para que éste tenga trabajo por hacer.

2. Verifica si puede almacenar una tarea en espera. Esto lo puede hacer siempre y cuando no exista ninguna tarea ya puesta, ya que sólo una tarea puede ocupar ese lugar.

3. Verifica si existe una tarea en ejecución y a la vez una en fila de espera, si es así, debe enviar la tarea hacia su próximo trabajador, si no se cumple la última parte, alguna de las dos anteriores debe cumplir.

Cada switch de la línea de trabajadores se comporta de la misma manera. Como el último trabajador está conectado con el primero, garantiza que no haya ningún trabajo perdido.

El último elemento es el catcher, que recibe todas las tareas procesadas por los n kernels. Una vez que haya recibido todas las tareas, manda dichas tareas al farmer, con el fin de que éste las ordene y presente resultados al usuario.

No debe perderse de vista que todo está basado en el modelo de *paso de mensajes*, por lo que es natural pensar, que cuando existe comunicación entre los diferentes procesos del paradigma de *Farming*, todos se comunican a través de mensajes. Observe la Figura 2.8. el farmer envía un mensaje con la etiqueta **ToNext** al primer switch, avisándole que le envía una nueva tarea. El kernel, envía el mensaje con la etiqueta **MoreWork** a su switch, pidiendo más trabajo, ya que éste terminó el que tenía en ejecución. El switch envía el mensaje **ToNext** al siguiente trabajador, cuando éste tiene un trabajo en ejecución y uno más en espera. El kernel, envía el mensaje con la etiqueta **WorkDone** al catcher cuando termina su trabajo y por último, el catcher envía **Terminate** en el momento que tiene todos los procesos almacenados al farmer.

La división de tareas *independientes* justifica la utilización del paradigma de *farming*, ya que como se verá en el siguiente capítulo, se puede transformar un programa secuencial, en múltiples partes, donde cada una de éstas es independiente de la otra. Por otra parte, se utiliza un *framework* o plantilla, *pyFarm* [3] implementado en *Python*, donde cualquier usuario sin previos conocimiento de procesamiento paralelo pueda usarlo.

Con el fin de entender mejor el contexto del paradigma de *Farming*, véase el siguiente ejemplo:

Supóngase que una universidad debe enviar a sus alumnos un comunicado, que debe ser firmado por cada alumno y devuelto al consejo universitario.

[3] ver apéndice C.3

El proceso resulta muy semejante al método para paralelizar un programa secuencial. Primero la universidad debe conocer perfectamente la dirección y nombre de todos los alumnos, con el fin de que dicho papel llegue a su destino. A continuación, la universidad escribe los comunicados, con las instrucciones que deben realizar los alumnos. Los comunicados se envuelven en un sobre y se etiquetan con el nombre del remitente y dirección del destinatario. A través de un servicio postal se envía el comunicado. El destinatario recibirá la carta y verificará que es para él, si es correcta la dirección y el nombre, abrirá la carta, leerá el contenido y hará alguna acción que diga en dicha carta, en este caso firmarla. Después tendrá que devolver la carta firmada al consejo universitario, para lo que debe meterla en un sobre, etiquetarla con la dirección y nombre del interesado y ponerla en un servicio postal para su entrega.

Véase la similitud del ejemplo con el paradigma de farming. La universidad debe crear un número grande de comunicados, es decir, el programa realiza múltiples veces una tarea, donde esta tarea pueden verse como un mensaje independiente que debe ser enviado a algún destinatario (*procesador*). El farmer genera dichos mensajes o cartas a enviar (ver 2.6.2). Se utiliza *PVM* como el servicio postal, es decir, se encarga de empaquetar, etiquetar y envía el mensaje o tarea a algún destinatario específico dentro de la máquina virtual (ver PVM 2.6.1). *PVM* al igual que en un servicio postal no le interesa que contiene la carta o mensaje, sólo a quién va dirigida y quién la recibe. Cuando llegue el mensaje a su destinatario, abrirá el mensaje y lo entregará. El mensaje debe contener las instrucciones exactas y correctas para que el *procesador* realice dicha tarea. Después cuando el trabajador haya terminado la tarea, llama a *PVM* para empaquetar el resultado y devolverlo a un almacén o *catcher* (ver 2.6.2) donde éste espera a que lleguen todas las tareas resueltas para enviarlas al programa principal *farmer*. Esta última parte, tiene su similitud cuando los alumnos reciben el comunicado, lo firman y envían al consejo universitario, donde éste debe recolectar todos los comunicados firmados y después entregarlos a la universidad.

Entonces la idea del ejemplo anterior, es hacer paquetes independientes, donde dichos paquetes contengan la información necesaria para que en cualquier *procesador* que reciba el paquete, pueda ser ejecutado sin ningún problema. Al conjuntar todos los paquetes procesados, se completa la tarea.

2.7 Medición de Desempeño en los Sistemas Paralelos.

Existen muchas formas de medir el desempeño de algoritmos paralelos. Las métricas más común son el *tiempo de ejecución*, *precio/desempeño*, *“speed-up”* y la *eficiencia* [7].

El *tiempo de ejecución*, es una de las métricas más importantes. Esta depende de las características del sistema paralelo y obviamente de la capacidad del procesador para realizar más instrucciones por segundo.

El *precio/desempeño*, es simplemente el tiempo que le toma a un programa ser ejecutado con respecto al costo de las máquinas que realizan dicho programa.

Las dos mediciones anteriores ayudan en la decisión de cuál máquina conviene comprar. Mientras la *eficiencia* y *speed-up* revelan la eficiencia de las máquinas.

El *speed-up*, es la medición entre el tiempo que le toma a un procesador (T(1)) ejecutar el programa con respecto al tiempo que le tomaría ejecutar el mismo programa en varios procesadores (T(p)). Se define como:

$$s = \frac{T(1)}{T(p)} \tag{2.1}$$

La *eficiencia* es una medición que relaciona el *precio/desempeño* de un sistema y se define:

$$e = \frac{T(1)}{pT(p)} = \frac{s}{p} \tag{2.2}$$

La *eficiencia* cuando es muy cercana a la unidad, quiere decir, que el hardware es usado de manera correcta; baja eficiencia se transforma en desperdicio de los recursos.

Todas las métricas mencionadas tienen sus desventajas. De hecho, existe información importante que no puede ser obtenida de ellas. Es obvio que aumentar el número de procesadores reduce el tiempo de ejecución pero ¿cuánto? Para ello se ocupa la métrica *speed-up*. Si el *speed-up* está muy cerca de ser lineal, implica buenas noticias, pero qué tan cercano a lo lineal es bueno.

La pregunta se puede contestar a través de una métrica llamada *fracción serial* [7], la cual se deduce de la siguiente manera.

La ley de Amdahl [7], se define como:

$$T(p) = T_s + \frac{T_p}{p} \tag{2.3}$$

donde T_s es el tiempo que le toma a la parte del programa que debe ejecutarse serialmente y T_p es el tiempo de la parte paralelizable.

$T(1) = T_s + T_p$. Si se define la *fracción serial* como $f = \frac{T_s}{T(1)}$, entonces la ecuación 2.3 se puede escribir como:

$$T(p) = T(1)f + \frac{T(1)(1-f)}{p} \tag{2.4}$$

o en términos de el *speed-up* s

$$\frac{1}{s} = f + \frac{1-f}{p} \tag{2.5}$$

Despejando f resulta:

$$f = \frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.6}$$

La *fracción serial* cuantifica la componente del programa que es serial. Si se observa la ecuación, al incrementarse p, la *fracción serial* debe tender a cero.

Todas estás métricas, ayudan a entender el comportamiento del algoritmo paralelo ejecutado sobre una multicomputadora. También permiten observar la eficiencia del algoritmo, y con ello localizar errores sobre el programa, con el fin de corregirlos.

Con todos los elementos revisados hasta el momento es posible entender como se construye un sistema paralelo. Para evaluar dichos sistemas es necesario tener un caso de estudio que requiera un gran poder de cómputo y para ello en el siguiente capítulo se presentan dos casos de estudio.

Capítulo 3

Casos de Estudio para evaluar los sistemas Beowulf.

En este capítulo se presentan las bases físico-matemáticas para caracterizar la distribución de presión acústica o campo de radiación acústica empleando dos métodos. Primero, el método de la respuesta al impulso y segundo el método basado en el principio de Huygens, ambos son casos de estudio que permitirán evaluar el desempeño de los clusters Beowulf.

3.1 Caracterización de la Distribución de Presión Acústica.

Una parte importante en la instrumentación de sistemas ultrasónicos es el transductor. Normalmente éste tiene un elemento piezoeléctrico, encargado de convertir una señal eléctrica en vibraciones mecánicas y viceversa. Muchas de las aplicaciones del ultrasonido se generan a través de excitaciones impulsivas o transitorias sobre los elementos piezoeléctricos. Gracias a la utilización de transductores, se ha logrado un mejor entendimiento y desarrollo en temas como: caracterización de materiales, pruebas no destructivas, imágenes biomédicas, análisis de tejidos, etc.

El estudio de las características espaciales y temporales del campo de vibraciones mecánicas o distribución de presión acústica, permite comprender dichos dispositivos. Esto es una buena herramienta para la caracterización y diseño de transductores ultrasónicos.

El análisis de la distribución de presión acústica se divide en dos regiones: el campo cercano o *Fresnel* y el campo lejano o *Fraunhoffer* [8]. Dichas regiones se ilustran en la Figura 3.1.

El campo cercano es la región frente a la cara del transductor y se caracteriza por rápidas variaciones del campo sonoro, debido a la interferencia (ver B.2) de las ondas mecánicas producidas por el movimiento del transductor y las ondas de borde generadas en la frontera del mismo.

El campo lejano o *Fraunhoffer*, tiene la característica que la presión acústica decrece linealmente a lo largo de una línea radial conectada con la fuente [9].

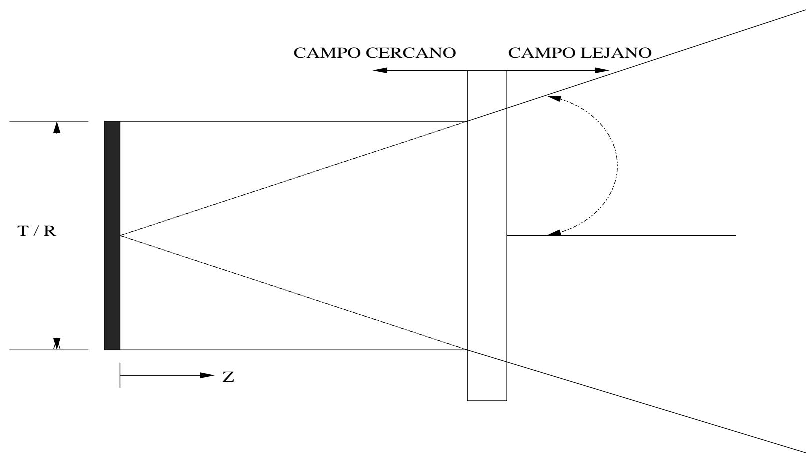


Figura 3.1: Campo cercano y lejano de un transductor (T/R).

Existen varios métodos que se han desarrollado con el fin de calcular la presión del campo acústico y se usan como plantillas o guías en el diseño y caracterización de transductores. En el presente trabajo se utilizan dos métodos para caracterizar el campo de radiación acústica:

- Método de la respuesta al impulso.
- Método basado en el principio de Huygens.

3.2 Método de la Respuesta al Impulso.

El método de la respuesta al impulso [10], proporciona una solución exacta para un transductor cuando éste se considera como un pistón plano sumergido en un baffle infinito, se ha probado que es una buena herramienta para conocer la presión acústica en regiones frente al transductor y con propiedades geométricas específicas.

La respuesta al impulso se define como una función del potencial de velocidad en un punto arbitrario del espacio, resultado de un movimiento impulsivo del pistón. Como resultado de esta transformación, puede obtenerse una expresión para el cálculo de la respuesta al impulso como función de las coordenadas espaciales y temporales, para cualquier geometría del pistón y en cualquier condición de frontera [10].

3.2.1 Teoría Acústica.

La presión acústica depende del tiempo en el espacio medio a partir de $z \geq 0$ (ver Figura 3.1), y es originada por la velocidad de las vibraciones del pistón en un medio isótropo [1]. La velocidad c se considera constante en el medio

[1] No hay variación de densidad

de propagación, también se asume que el pistón está empotrado en un baffle infinito. Esto se puede modelar como un problema clásico de condiciones de frontera.

Al escoger la condición de frontera correcta para la ecuación de onda considerando el potencial de velocidad, $\varphi(\mathbf{r}, t)$, la radiación acústica emitida es:

$$\nabla^2 \varphi(\mathbf{r}, t) - \frac{1}{c^2} \frac{\partial^2 \varphi(\mathbf{r}, t)}{\partial t^2} = 0. \tag{3.1}$$

De la Eq. 3.1 la presión acústica $p(\mathbf{r}, t)$ puede ser obtenida

$$p(\mathbf{r}, t) = \rho_0 \frac{\partial \varphi(\mathbf{r}, t)}{\partial t} \tag{3.2}$$

donde ρ_0 es la densidad del medio.

El potencial de velocidad depende del tiempo y un punto arbitrario de observación $\mathbf{r}$ en el espacio medio y puede ser determinado por

$$\varphi(\mathbf{r}, t) = \int_0^t dt_0 \int_S v(\mathbf{r}_0, t_0) g(|\mathbf{r} - \mathbf{r}_0|, t)\, dS \tag{3.3}$$

donde $v(\mathbf{r}_0, t_0)$ es la velocidad en la cara del pistón, $\mathbf{r}_0(x_0, y_0, z_0)$ es un punto arbitrario sobre la superficie S del pistón, y $g(|\mathbf{r} - \mathbf{r}_0|, t)$ es la función de *Green* de un movimiento armónico simple dependiente de la condición de frontera. Si la velocidad se considera uniforme sobre la cara del pistón y la función de *Green* es una solución del pulso de onda [11], el potencial de velocidad se puede escribir como:

$$\begin{aligned} \varphi(\mathbf{r}, t) &= \int_0^t v(t_0) dt_0 \int_S g(|\mathbf{r} - \mathbf{r}_0|, t)\, dS \\ &= v(t) * h(\mathbf{r}, t) \end{aligned} \tag{3.4}$$

donde $v(t)$ es la velocidad promedio en la superficie del pistón, la operación $*$ representa la convolución y $h(\mathbf{r}, t)$ se define como la respuesta al impulso.

$$h(\mathbf{r}, t) = \int_S g(|\mathbf{r} - \mathbf{r}_0|, t) dS). \tag{3.5}$$

La expresión que modela la respuesta al impulso (Eq. 3.5), es una integral de superficie, que depende de las características geométricas de la cara del pistón, es decir, si cambia la geometría del transductor, su respuesta cambiará.

Si se sustituye la Eq. 3.4 en la Eq. 3.2 la presión acústica se convierte en

$$p(\mathbf{r}, t) = \rho_0 v(t) * \frac{\partial h(\mathbf{r}, t)}{\partial t}. \tag{3.6}$$

Esta ecuación es un caso típico de modelado de un sistema digital (ver apéndice A).

3.2.2 Condiciones de Frontera.

Las condiciones de frontera más comunes para estos sistemas son [12]:

- Baffle Rígido: Si un pistón está empotrado en un baffle infinito y la única parte que puede vibrar es la cara del pistón, significa que la velocidad en la frontera es cero. La función de Green para está condición es [13]:

$$g(\mathbf{r}, t) = g(|\mathbf{r} - \mathbf{r}_0|, t) + g(|\mathbf{r} - \mathbf{r}_0 - 2\mathbf{z}|, t) \tag{3.7}$$

- Baffle suave: El potencial acústico o resistencia al medio, debe ser cero en la frontera, esto implica que la presión en z_0 = 0 es cero fuera de la superficie del transductor. La función de Green para esta condición es [13]:

$$g(\mathbf{r}, t) = g(|\mathbf{r} - \mathbf{r}_0|, t) - g(|\mathbf{r} - \mathbf{r}_0 - 2\mathbf{z}|, t) \tag{3.8}$$

3.2.3 Respuesta al Impulso para un Transductor Circular.

La respuesta al impulso para un transductor circular de radio a, localizado en $z_0 = 0$ y empotrado en un baffle rígido infinito, fue obtenida por *Obberhettinger* en 1961 [14], quien observó que la respuesta para una excitación arbitraria puede ser obtenida por la convolución. La respuesta al impulso, para un pistón empotrado en un baffle rígido infinito, se establece de la siguiente manera [10]:

$$h_{sb}(\mathbf{r}, t) = \begin{cases} 0 & t < t_1 \\ c & t_1 \leq t < t_2 \\ \dfrac{z}{t\pi} \cos^{-1}\left(\dfrac{r_{xy}^2 + (ct)^2 - z^2 - a^2}{2r_{xy}((ct)^2 - z^2)^{\frac{1}{2}}} \right) & t_2 \leq t \leq t_3 \\ 0 & t > t_3 \end{cases} \tag{3.9}$$

donde r_{xy} es la proyección de $\mathbf{r}$ dentro del plano del transductor. Los límites de tiempo se muestran en el Tabla 3.1, los cuales dependen de la posición de r_{xy}, como se muestra en la Figura 3.2.

La respuesta al impulso para un un transductor circular empotrado en un baffle suave como condición de frontera, fue propuesta por Rodríguez y

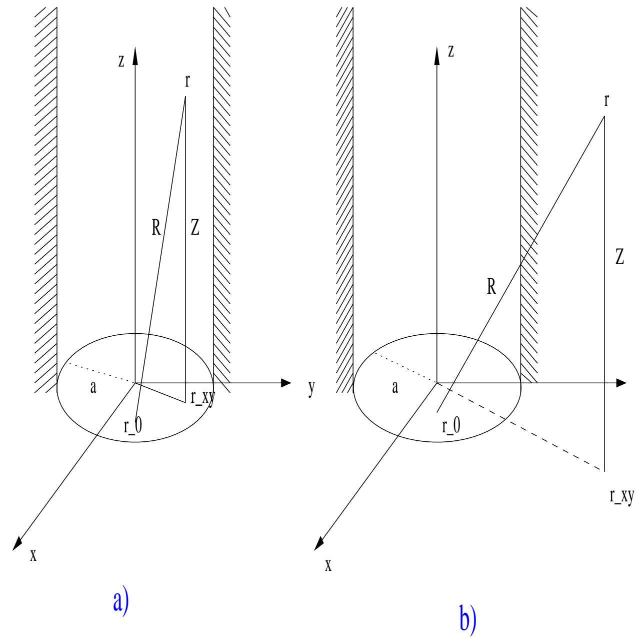


Figura 3.2: La proyección de **r**: a) $r_{xy} \leq a$ y b) $r_{xy} > a$

i) $r_{xy} \leq a$	ii) $r_{xy} > a$
$t_1 = \frac{z}{c}$	$t_1 = t_2$
$t_2 = \frac{(z^2+(a-r_{xy})^2)^{\frac{1}{2}}}{c}$ $t_3 = \frac{(z^2+(a+r_{xy})^2)^{\frac{1}{2}}}{c}$	

Tabla 3.1: Límites numéricos en la evaluación de $h(\mathbf{r}, t)$

colaboradores [15] de la siguiente manera:

$$h_{sb}(\mathbf{r},t) = \begin{cases} 0 & t < t_1 \\ \frac{z}{t} + \int\limits_{t_1}^{t} \frac{z}{t^2}dt & t_1 \leq t < t_2 \\ \frac{z}{t\pi}\cos^{-1}\left(\frac{r_{xy}^2 + (ct)^2 - z^2 - a^2}{2r_{xy}((ct)^2 - z^2)^{\frac{1}{2}}}\right) + \int\limits_{t_1}^{t_2} \frac{z}{t^2}dt \quad + & \\ \qquad \int\limits_{t_2}^{t} \frac{z}{t^2\pi}\cos^{-1}\left(\frac{r_{xy}^2 + (ct)^2 - z^2 - a^2}{2r_{xy}((ct)^2 - z^2)^{\frac{1}{2}}}\right)dt & t_2 \leq t \leq t_3 \\ \int\limits_{t_1}^{t_2} \frac{z}{t^2}dt + \int\limits_{t_2}^{t_3} \frac{z}{t^2\pi}\cos^{-1}\left(\frac{r_{xy}^2 + (ct)^2 - z^2 - a^2}{2r_{xy}((ct)^2 - z^2)^{\frac{1}{2}}}\right)dt & t > t_3 \end{cases} \tag{3.10}$$

donde el plano del transductor coincide con el plano xy y para $z_0 = 0$ la función de Green se convierte en cero.

3.3 Método Basado en el Principio de Huygens.

Supóngase que existe un punto que emite vibraciones (centro de perturbación), que se mueve en un medio continuo. El lugar geométrico de los puntos donde llegan las vibraciones en un momento determinado y a fase constante, se llama frente de onda. Si el medio es isótropo, la vibración del centro se propaga en todas direcciones, por lo que el frente de onda es una superficie esférica, cuyos centros coinciden con el centro de perturbación. El radio que existe entre el frente de onda y el origen de perturbación, se determina con $r = ct$.

Si un frente de onda F se hace pasar a través de un medio no uniforme, de tal forma que el frente de onda mismo es distorsionado. ¿Cómo se puede determinar la nueva forma del frente de onda F'? Dicho problema fue solucionado por el holandés *Christian Huygens* en 1690 en el trabajo titulado *Traité de la Lumiere*, donde se enunció lo que se conoce como el *principio de Huygens* [16]:

"Cada punto en un frente de onda primario sirve como fuente de ondas esféricas secundarias tales que el frente de onda primario un momento más tarde es la envolvente de estas ondas. Además, las ondas avanzan con una rapidez y frecuencia igual a la de la onda primaria en cada punto del espacio".

El la Figura 3.3 se observa en la región más cercana a la ranura o región Fresnel, las ondas son esféricas, mientras en la región Fraunhoffer el frente de onda es plano. El límite que divide las regiones se expresa [8]:

$$R_{fresnel} = \frac{B_{max}^2}{\lambda} = B_{max}^2\frac{f}{c} \tag{3.11}$$

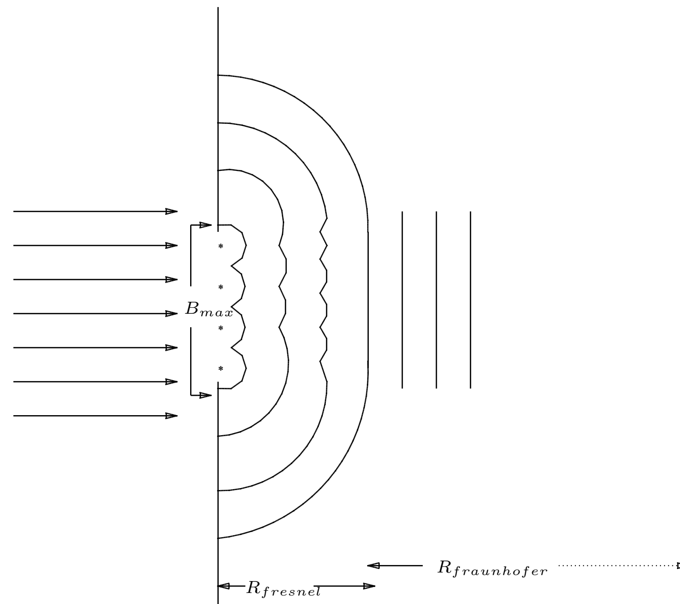


Figura 3.3: Principio de Huygens.

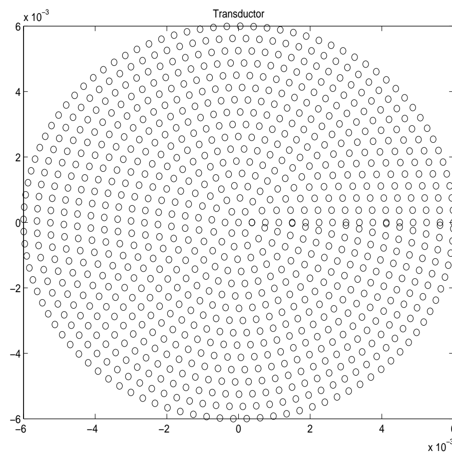


Figura 3.4: Arreglo de Transductores.

donde λ es la longitud de onda, B_{max} es la longitud máxima de la ranura y f es la frecuencia.

Para caracterizar la distribución de presión acústica utilizando el principio de Huygens, se parte de la región vibrante que se considera como un conjunto de puntos o emisores separados una longitud de $\frac{\lambda}{4}$, como se muestra en la Figura 3.4. La señal esférica generada por cada emisor viajará a través del espacio que se encuentra frente a la región. Dado algún punto sobre el espacio a caracterizar, se cuantifica la influencia de todas las señales emitidas por los transductores con respecto a dicho punto. Matemáticamente se expresa en la siguiente ecuación

$$S_r(\mathbf{r}, t) = \sum_{i=1}^{N} S_i(\mathbf{r}, t - \Delta t) \tag{3.12}$$

donde S_r es la señal total en $\mathbf{r}$ y S_i es la señal debida a cada punto i.

3.4 Medición de la Presión Acústica.

Existen parámetros importantes que permiten conocer el comportamiento de la distribución acústica. Estos parámetros identifican los puntos máximos y mínimos de presión acústica en cualquier punto de observación. Se derivan de la forma de onda de la presión y son:

- La presión pico (P_p), definida como el máximo positivo ($P+$) o el módulo del máximo negativo ($P-$) del valor de presión instantáneo.
- La presión pico a pico (P_{pp}) definida por el módulo de la diferencia entre el valor máximo y el mínimo de la presión acústica instantánea.
- La presión cuadrática media (P_{rms}) definida como:

$$\sqrt{\frac{1}{N} \sum_{n=1}^{N} (P_c[n])^2} \tag{3.13}$$

P_p y P_{pp} son parámetros que proporcionan una comparación entre puntos de intensidad, y una muy importante herramienta para imágenes ultrasónicas. Mientras la P_{rms} es una medida cuantitativa de la energía en la distribución de presión acústica.

Capítulo 4

De Secuencial a Paralelo.

Este capítulo trata la implementación de los métodos descritos en el capítulo 3 para obtener la distribución de presión acústica, tanto en su versión serial como paralela. Todos los programas se realizaron en el lenguaje de alto nivel orientado a objetos **PYTHON** versión 2.3 [17][18].

4.1 Simulación de la Respuesta al Impulso Secuencial.

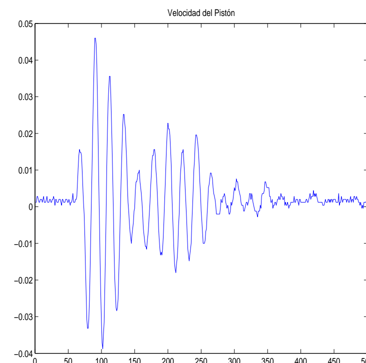
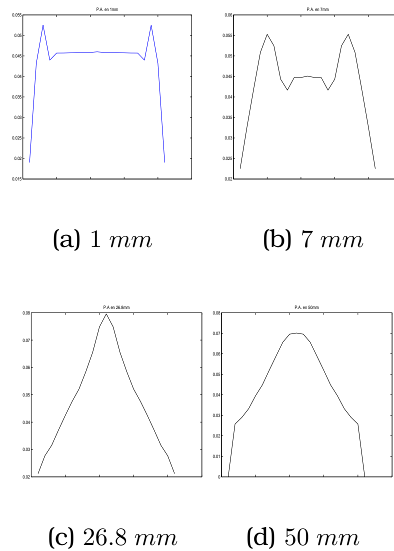


Figura 4.1: Velocidad del pistón obtenida experimentalmente.

Para conocer la distribución de presión acústica debida a la excitación de un transductor circular, es necesario estudiar su respuesta al impulso considerando la geometría del transductor y condiciones específicas de frontera. El cálculo de la respuesta al impulso para la condición de frontera baffle rígido, no requiere del uso de un sistema paralelo. Se puede observar en la Ec. 3.9 que no presenta una carga fuerte para una computadora serial. Sin embargo el método numérico para obtener la respuesta al impulso considerando la condición baffle suave, requiere de un alto poder computacional, debido al grado de complejidad temporal y de almacenamiento que requiere el algoritmo para su solución, por lo que esto justifica el uso de un sistema paralelo. Primero, se implementa el método numérico de forma serial y después en su versión paralela.

El programa serial se muestra en el apartado D.1. Una explicación por bloques facilitará al lector el entendimiento del programa. Para obtener la distribución de presión acústica, se debe contar con dos elementos como se ve en la Eq. 3.6. Primero $v(t)$ es la velocidad media de la cara del pistón obtenida experimentalmente [13] (ver Figura 4.1) y la respuesta al impulso $\mathbf{h}$ propuesta



(a) $1\ mm$ (b) $7\ mm$

(c) $26.8\ mm$ (d) $50\ mm$

Figura 4.2: Distribución de presión acústica

por Rodríguez y colaboradores [15] (ver 3.10).

El primer bloque *"carga velocidad"*, lee los valores de velocidad media del pistón, contenidos en el archivo *"waveform_cara.txt"*. El segundo, *"variables de entrada"*, define las variables de entrada, que cambian la respuesta al impulso y por tanto la presión acústica. Como se vio en el capítulo 3, los parámetros físicos del transductor y el medio donde interactúa intervienen en el comportamiento de la distribución de presión acústica. Estos parámetros son el radio del transductor y la velocidad de propagación del medio. Por último el tercer bloque "*ciclos*", es la parte interesante del programa para el cálculo computacional. Esta dividido en tres ciclos que forman *una matriz* que representa un plano. El ciclo más externo recorre el eje z (*profundidad frente a la cara del transductor*). El siguiente ciclo recorre el eje x horizontalmente a la cara del transductor y calcula la presión acústica instantánea para P_p, P_{pp} y P_{rms}, para cada punto en el espacio enfrente del transductor. El ciclo más interno obtiene la respuesta al impulso **h**.

4.1.1 Resultados de la Respuesta al Impulso Secuencial.

Con el fin de interpretar los resultados de la implementación del programa, se propusieron las entradas al sistema y se calculó el límite para la región *Fresnel* de la siguiente manera: frecuencia de resonancia $f = 1.0\ Mhz$, radio de transductor de $6.35\ \mu m$ y $c = 1500\ \frac{m}{s}$. Con la ecuación 3.11, se obtiene el límite de la región *Fresnel*, que resulta $R_{fresnel}$ de $26.8\ mm$.

A continuación se interpretan resultados: la Figura 4.2 (a) muestra la gráfica de un plano paralelo a la cara del transductor a una distancia de $1\ mm$. En las orillas del transductor se ven dos lóbulos provocados por el efecto de borde en la frontera del transductor y a la rápida variación de la pre-

sión acústica. A la distancia de $7\ mm$ Figura 4.2 (b), aparece un tercer lóbulo debido a la acumulación de presión sobre la línea central al transductor.

En la Figura 4.2 (c) para $26.80\ mm$, la onda mecánica tiene acumulada la máxima amplitud que puede alcanzar, y a partir de esta distancia la amplitud decrece parabólicamente, como se observa en la Figura 4.2 (d) que se encuentra a una distancia de $50\ mm$.

El plano central presenta la máxima concentración de presión acústica, debido a la geometría circular del transductor. La Figura 4.3, muestra la distribución de presión acústica para dicho plano para P_{rms}.

La simulación de la presión acústica para cualquier punto del espacio, proporciona información valiosa para localizar la máxima y mínima presión acústica producida por el transductor. Es importante notar que alrededor del límite del campo cercano se encuentran los puntos de mayor amplitud. Las matrices obtenidas por la simulación contienen las magnitudes de cada punto de presión sobre el plano.

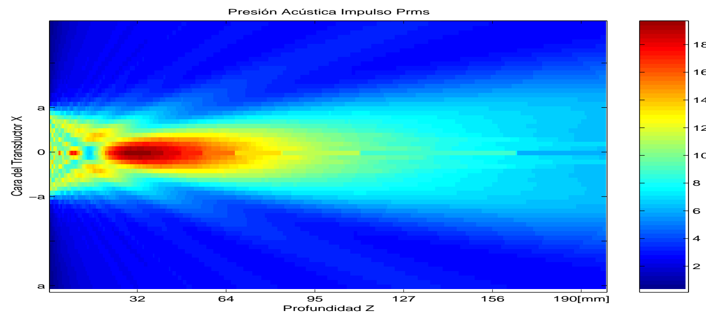


Figura 4.3: Plano central de la distribución de presión acústica

4.1.2 Paralelización de la Respuesta al Impulso.

Recordar que la idea para paralelizar, es hacer paquetes independientes donde éstos contengan la información necesaria para que cualquier *procesador* que reciba algún paquete pueda ejecutarlo sin ningún problema.

La implementación del método de la respuesta al impulso en su versión secuencial se localiza en el apéndice D.1. Obsérvese el bloque *"ciclos"*, el cual tiene tres proposiciones condicionales *for*, donde el más interno de ellos es el importante, debido a que en éste se calcula la respuesta al impulso **h** y aquí se encuentra el grado mayor de complejidad del algoritmo. Para crear los paquetes independientes, gracias a la mínima dependencia de datos, véase la última proposición bicondicional *for* como un conjunto de tareas autónomas que se deben realizar y por lo tanto factible de separarlas en varios lugares de procesamiento. Aquí se construye cada mensaje a enviar. Los mensajes deben tener los parámetros necesarios con el fin de que al enviar la información como un único paquete, el *nodo de procesamiento* que reciba dicho mensaje, realice el cálculo sin que le falte ninguna variable.

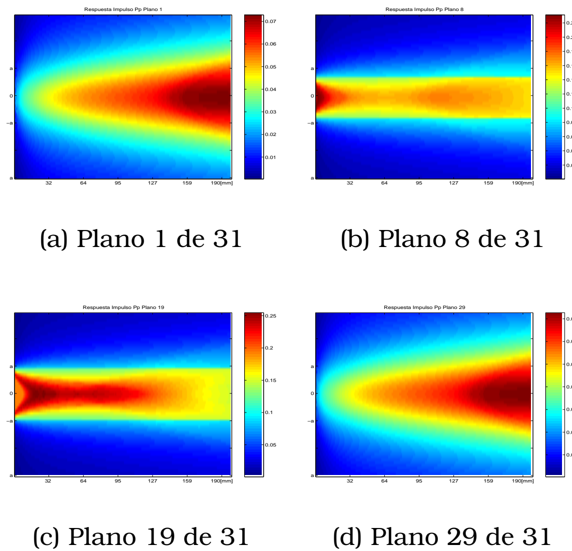
Para obtener la respuesta al impulso **h** sobre cualquier iteración de la última proposición condicional *for*, se necesita la posición en el plano, dada por los ciclos más externos (*"inx, inz"*) y los parámetro *"L, a, c, ts, z, sig"*. Con estos parámetros, la respuesta al impulso para una posición dada en el espacio puede ser calculada sin ningún error. Por lo que es fácil suponer que el cálculo de la respuesta al impulso se puede ver como una función independiente que tiene parámetros de entrada y deben ser pasados para su correcta ejecución. Véase el programa paralelizado en el apéndice D.2. El programa es idéntico al secuencial hasta el tercer *for*, que ya no existe y es una función llamada *baffpar* que requiere de parámetros de entrada para ser ejecutada (ver D.3). En el programa paralelo, existe la variable *parametros*, que no es más que el empaquetamiento de todas las variables necesarias con el fin de que la función *baffpar* pueda procesar. El paquete *parametros* se envía con la función *farm.send* vía el framework *pyFarm* a alguna máquina que componen la máquina virtual de *PVM*, para que ésta sea ejecutada. Después la función *farm.mastrab* generará los siguientes paquetes a enviar, hasta que el número de tareas a realizar se haya terminado. Cuando cada paquete enviado termine su proceso y tenga el resultado lo enviará al *catcher*, éste esperará hasta que todas las tareas estén completas por medio de la función *farm.wait*. Una vez entregadas todas las tareas, el programa principal o *farmer* ordena las tareas realizadas y presenta resultados.

4.1.2.1 Resultados al Paralelizar la Respuesta al Impulso.

Como resultado de paralelizar el programa, no sólo se puede caracterizar el plano central del transductor, ahora, se puede aumentar un ciclo más al programa para obtener un volumen; esto es, caracterizar la mayor cantidad de planos perpendiculares a la cara del transductor y observar el comportamiento de la distribución acústica conforme cambia de plano o altura. También se puede observar la simetría con respecto al plano central.

Se muestran diferentes planos de la distribución de presión acústica en la Figura 4.4. Si se comparan las Figuras (a) y (d) de la Figura 4.4, se observa la simetría de la distribución de presión acústica, es decir, se calcularon treinta y un planos y se muestra el plano número uno y el plano número treinta, que son simétricos con respecto al plano central mostrado en la Figura 4.3.

Al reducirse el tiempo de máquina, se pude obtener el volumen de la distribución de presión acústica. El comportamiento volumétrico de la presión acústica se muestra para diferentes cortes: la Figura 4.5 (a) muestran el plano central de la distribución para P_p, la Figura 4.5 (b) muestra un corte desde el plano central hacia planos inferiores, por último la Figura 4.5 (c) muestra el conjunto de planos calculados de la distribución acústica.



(a) Plano 1 de 31 (b) Plano 8 de 31

(c) Plano 19 de 31 (d) Plano 29 de 31

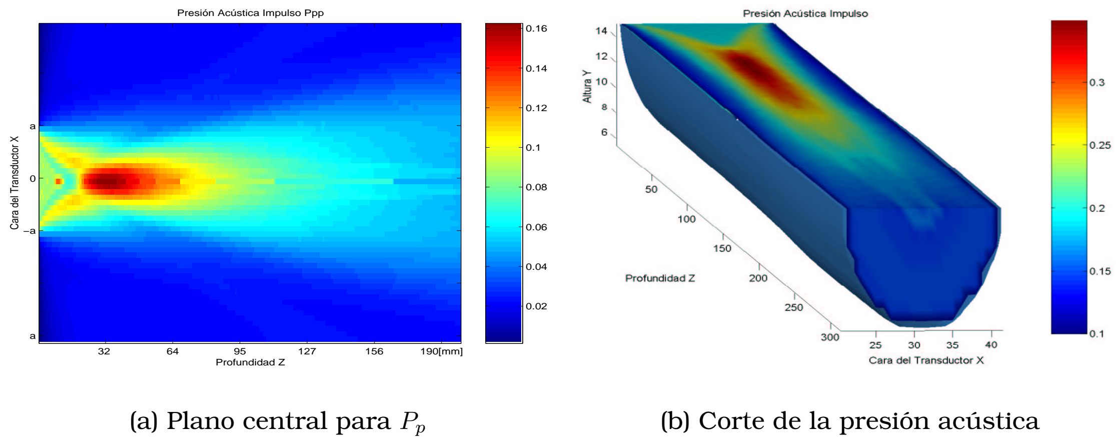
Figura 4.4: Presión acústica para diferentes planos

4.2 Simulación del Método de Huygens Secuencial.

En el apéndice D.4 se muestra el programa por bloques, al igual que en la respuesta al impulso, hay un primer bloque que define las variables de entrada *"entradas"* al sistema y éstas modifican la distribución de presión acústica. El segundo bloque (*transductor*) crea el arreglo de emisores o puntos vibrantes (ver Figura 3.4), donde cada emisor genera una señal esférica que se propaga por el espacio a caracterizar. El último bloque *"ciclos"* realiza la suma algebraica de las señales trasmitidas por los emisores e incidentes para cada punto de observación sobre el espacio.

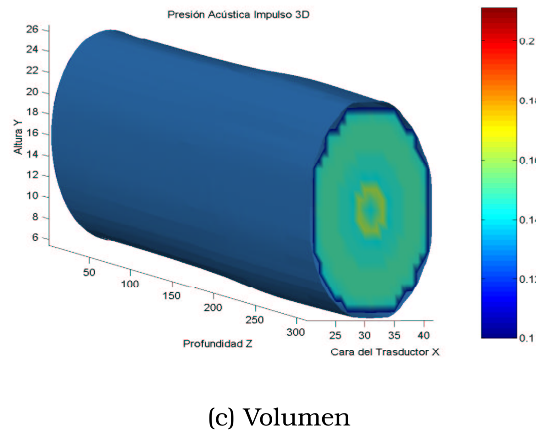
4.2.1 Resultados del Método de Huygens Secuencial.

En la Figura 4.6, se muestra el comportamiento de la distribución acústica del plano central al transductor para P_{pp}. El color rojo representa los puntos de mayor amplitud, mientras el color azul denota la menor amplitud sobre la distribución. La simulación permite conocer el comportamiento de la presión acústica en tiempo y espacio. Obsérvese el comportamiento de la distribución para el campo lejano, es muy similar al obtenido con la respuesta al impulso. También es importante decir que este comportamiento sólo se obtiene para las entradas específicas con las que se simuló, es decir, que el límite del campo cercano varía al cambiar el radio del transductor y el medio de propagación.



(a) Plano central para P_p

(b) Corte de la presión acústica



(c) Volumen

Figura 4.5: Distribución volumétrica de la respuesta al impulso

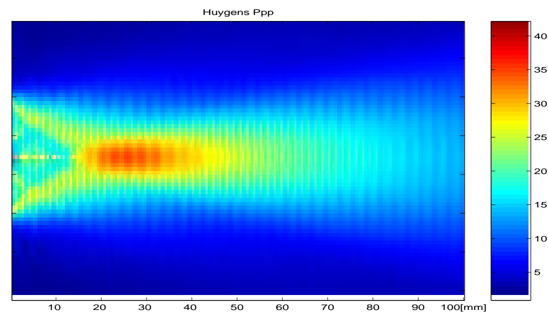
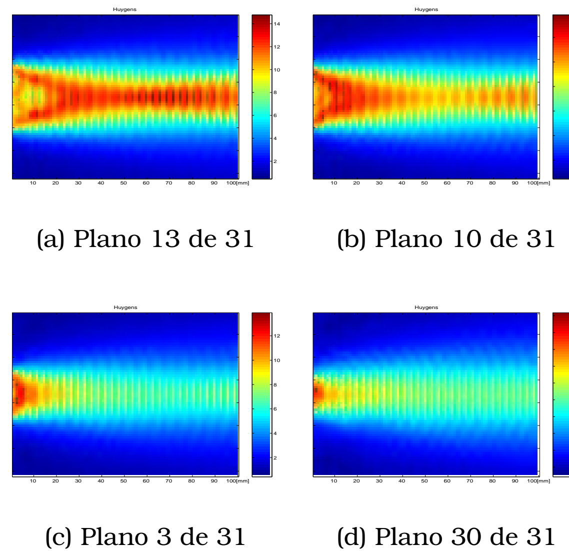


Figura 4.6: Plano central de la presión acústica en Huygens



(a) Plano 13 de 31 (b) Plano 10 de 31

(c) Plano 3 de 31 (d) Plano 30 de 31

Figura 4.7: Planos de la distribución volumétrica en Huygens

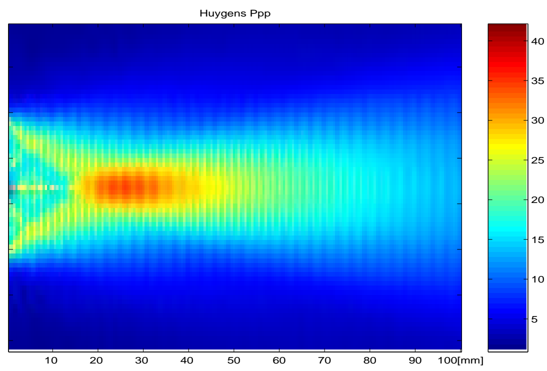
4.2.2 Paralelización del Método de Huygens.

El proceso para paralelizar este programa, es idéntico al visto en la sección 4.1.2, de la misma manera se empaquetan los parámetros del ciclo *for* más interno y se convierten en mensajes que serán enviados por *pyFarm* a cada nodo de procesamiento que componen la máquina virtual. El programa se muestra en el apéndice D.5 y el mensaje a enviar se presenta en D.6. Al igual que en el programa anterior se muestran diferentes planos a varias alturas con respecto a la cara del transductor (ver Figura 4.7).

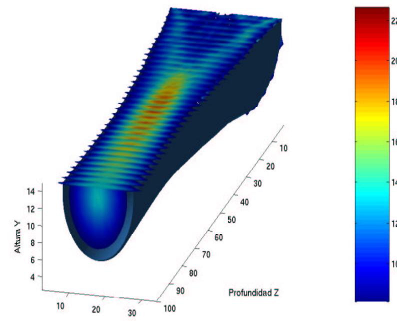
La Figura 4.8 (a) muestra el plano central de la distribución para P_{pp}. La Figura 4.8 (b) muestra un corte de la presión acústica y por último la Figura 4.8 (c) presenta el comportamiento volumétrico de la presión acústica.

Cabe mencionar que al reducir el tiempo de procesamiento, en ambos casos se puede caracterizar la mayor cantidad de planos y obtener una mayor aproximación en el comportamiento de la distribución de presión acústica.

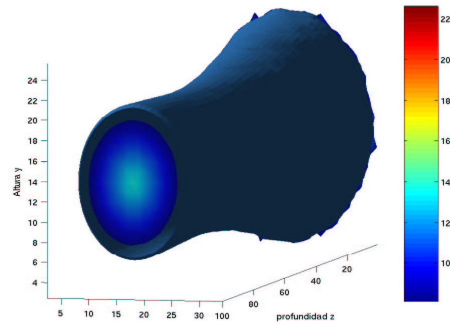
Con los elementos vistos hasta el momento, es posible plantearse la pregunta de que tan eficiente es un sistema multicomputadora y si tiene sentido su utilización o no. Para ello, en el siguiente capítulo, se realizan pruebas de desempeño, con el fin de obtener los parámetros más importantes para cuantificar su eficiencia.



(a) Plano central para P_{pp}



(b) Corte de la presión acústica



(c) Volumen

Figura 4.8: Distribución volumétrica en Huygens

Capítulo 5

Pruebas de Desempeño.

5.1 Automatización de Pruebas de Desempeño.

Con los programas en una versión paralela, el siguiente paso es encontrar el mejor desempeño al ejecutarlos sobre los *clusters Beowulf*. Para lograr esto, se realizan diferentes pruebas, básicamente variando el número de máquinas (procesadores) y el número de trabajadores (procesos).

Para realizar las pruebas, se cuenta con dos *Clusters Beowulf* de diferente configuración:

Para el primero de ellos, es la siguiente.

- 8 nodos diskless
- Sistema operativo: *LINUX*
- Distribución Red-hat 7.3
- Versión del kernel: 2.4.25
- Carga del kernel y sistema operativo vía *Network File System* (NFS).
- Procesador dual *INTEL pentium III* a 1.0 GHz con 512 $Mbytes$ en RAM
- Tecnología de red *ethernet*
- Topología de red: Estrella
- Nombre del *cluster*: Onomatopoeiia

El segundo *cluster* tiene la siguiente configuración:

- 4 nodos con disco duro
- Sistema operativo: *LINUX*
- Distribución Debian Woody
- Versión del kernel: 2.4.27

- Carga kernel y sistema operativo vía LILO
- Procesador *INTEL pentium IV* a 2.5 GHz con 512 $Mbytes$ en RAM
- Tecnologías de red *SCI y ethernet*
- Topología de red: Estrella para ethernet y malla 2D para SCI
- Nombre del *cluster*: Cuaco

Debido al volumen de pruebas que se deben realizar, se opta por utilizar la herramienta *make*, debido a que permite automatizar las pruebas sin la necesidad de supervisión (ver apéndice C.2).

5.2 Punto Óptimo.

Antes de comenzar las pruebas, se realiza un análisis de latencia para la tecnología SCI y ethernet con la ayuda del programa llamado *Netpipe* [1]; éste envía paquetes de tamaños variables entre dos nodos. El proceso se realiza a través de PVM. El tiempo de recorrido del paquete con respecto al tamaño del mismo se gráfica para cada tecnología de red con el fin de compararlas.

Los paquetes comienzan con un tamaño de 1 byte hasta 12582915 bytes, y sus tiempos varían desde 0.000067 segundos hasta 1.105405 segundos para ethernet y de 0.000047 hasta 1.03957 segundos para SCI. Para graficar esta relación se optó por usar una escala logarítmica base diez, debido a que la cantidad de datos es grande y en esta escala se aprecian con más detalle los resultados. En el apéndice E Tabla E.1, se muestran algunos de los valores de tiempo de latencia en segundos y tamaño de los paquetes en bytes. En la Figura 5.1 se muestra la comparación de las tecnologías de red.

Para comenzar las pruebas se parte al ejecutar los dos programas de forma secuencial (*Respuesta al impulso D.1*, *Huygens* D.4), con el fin de tener la referencia del tiempo que toma su ejecución. Las pruebas en paralelo se organizan en dos etapas de la siguiente manera:

- Determinación del número óptimo de trabajadores.
- Métricas de desempeño

5.2.1 Determinación del Número Óptimo de Trabajadores.

Para la primera etapa de pruebas, se usan todos los procesadores de cada uno de los *cluster*. En las pruebas se varía el número de trabajadores (procesos), procurando que éste sea el mismo para cada nodo, en caso de no ser posible, se asigna un trabajador por nodo hasta terminar con el residuo ("a la forma de repartir los trabajadores en los nodos se le llama mapeo"), esto se hace con

[1] http://www.scl.ameslab.gov/netpipe/

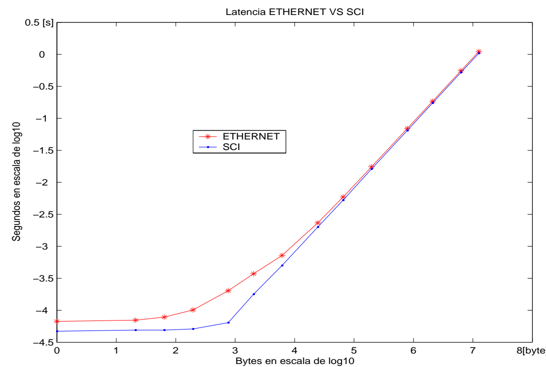


Figura 5.1: Latencias de la tecnología SCI y ethernet.

el fin de evitar en lo posible el desbalanceo de cargas en el *cluster*. Aquí el objetivo es determinar el número de trabajadores que concluyan la ejecución del programa en el menor tiempo.

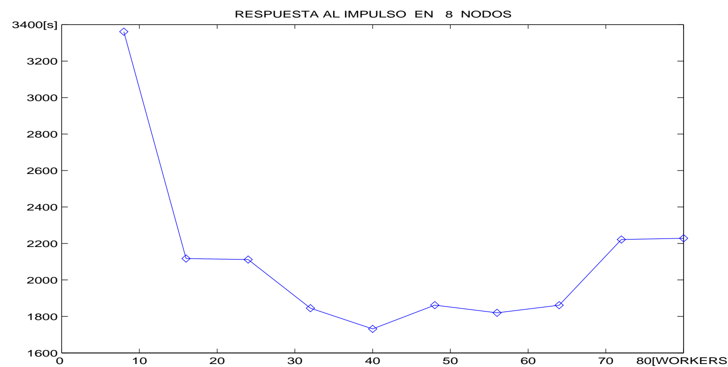
No hay que perder de vista, que se tienen dos *clusters* con diferente configuración y se desea evaluar sus desempeños, para lo cual hay que separar en dos grupos las pruebas. Además el *cluster Cuaco* tiene dos tipos de interfaz de comunicación de red (*SCI y ethernet*), por lo que para cada interfaz se deben repetir las pruebas. Por otro lado, se quiere comparar el desempeño de los dos *clusters* utilizando el mismo número de procesadores y tecnologías de red diferentes.

Cabe hacer notar que se observó que las pruebas tienen variaciones en el tiempo total de ejecución para un mismo caso. Como consecuencia se repitieron las pruebas 20 veces y se obtuvo los promedios y las desviaciones estándar de los tiempos totales de proceso en cada caso.

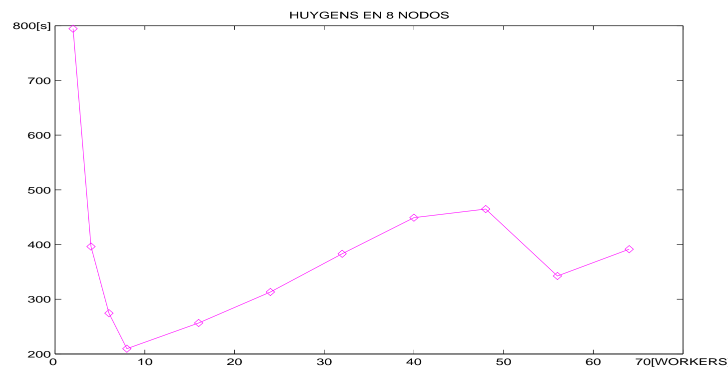
Las pruebas a realizar en esta etapa son las siguientes:

- Cluster Onomatopoeiia
  - Respuesta al Impulso paralelo
  - Huygens paralelo
- Cluster Cuaco
  - Respuesta al Impulso paralelo con ethernet
  - Huygens paralelo con ethernet
  - Respuesta al Impulso paralelo con SCI
  - Huygens paralelo con SCI
- Comparación de Clusters
  - Respuesta al Impulso con cuatro procesadores.
  - Huygens con cuatro procesadores.

De los vectores obtenidos de las pruebas anteriores, se elaboraron las siguientes gráficas:



(a) Trabajadores óptimos en respuesta al impulso



(b) Trabajadores óptimos en Huygens

Figura 5.2: Respuesta al impulso y Huygens en Onomatopoeiia

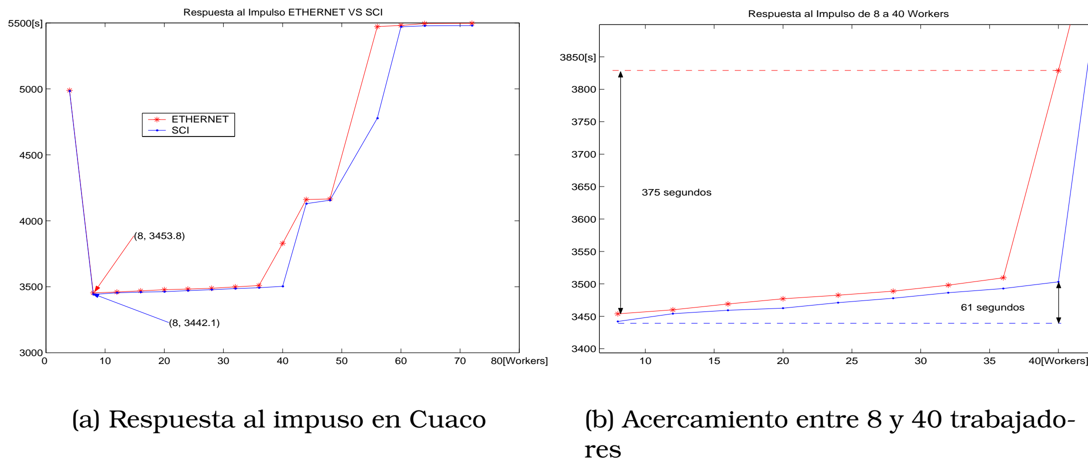
5.2.1.1 Respuesta al Impulso en Onomatopoeiia.

Para el caso del programa de la respuesta al impulso usando los 8 nodos del cluster (ver figura 5.2 (a)), el punto óptimo en tiempo se localiza en 40 trabajadores. Se observa que el comportamiento es similar a una ***U***. La prueba se realizó variando desde 8 hasta 80 trabajadores.

5.2.1.2 Huygens en Onomatopoeiia.

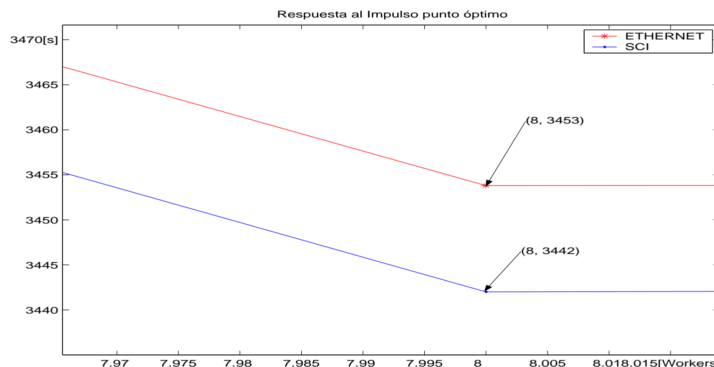
Para esta prueba se varió el número de trabajadores desde 2 hasta 64 trabajadores, usando los 8 nodos del cluster. La Figura 5.2 (b) muestra que 8 trabajadores logran el menor tiempo. Por un lado al variar entre 2 y 6 trabajadores, el número de trabajadores es menor que el número de nodos y como consecuencia dos o más nodos no son utilizados, debido a la forma de mapeo.

Las pruebas para Huygens como la respuesta al impulso en el cluster Cuaco se presentan comparando la tecnología de ethernet contra SCI.



(a) Respuesta al impuso en Cuaco

(b) Acercamiento entre 8 y 40 trabajadores



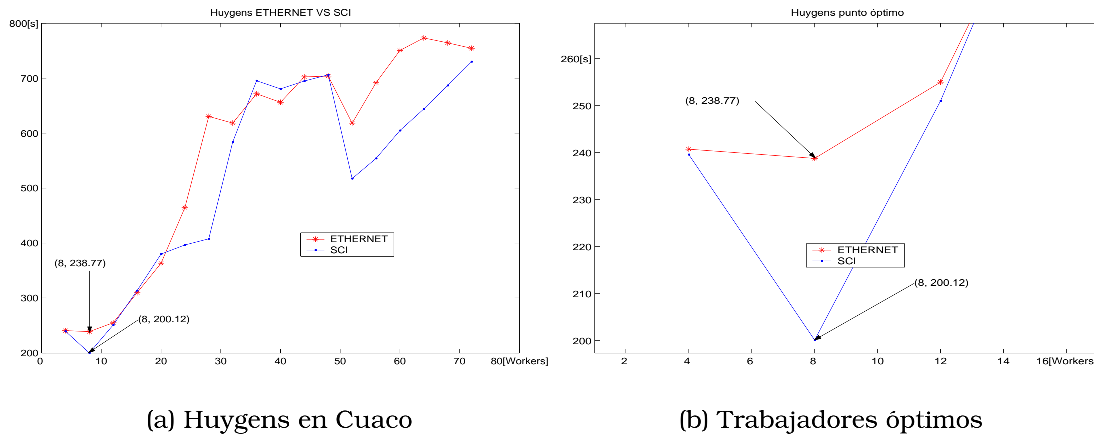
(c) Trabajadores óptimos en respuesta al impulso

Figura 5.3: Respuesta al impulso ethernet VS SCI en Cuaco

5.2.1.3 Ethernet VS SCI para la Respuesta al Impulso en Cuaco.

Al comparar el comportamiento del programa utilizando las tecnologías SCI y ethernet (ver Figura 5.3 (a)), se observa muy poca variación de tiempo entre ambas tecnologías para cada uno de los puntos de la gráfica. Ocho trabajadores logran el mejor tiempo. Hay que notar que el tiempo con SCI es menor en la mayoría de los casos, es decir, tiene una pequeña ganancia de tiempo debida a la *latencia* que presenta la tecnología, y como resultado obtiene una ganancia en tiempo total de proceso.

Un acercamiento a la región entre 8 y 40 trabajadores se muestra en la Figura 5.3 (b). Los tiempos obtenidos por SCI son mejores en todos los casos, además la diferencia de tiempo entre 8 y 40 trabajadores es de 61 segundos, sin embargo la misma medición para ethernet tiene una diferencia de 375 segundos. Se observa que para el caso de 40 trabajadores, la diferencia de tiempo entre ambas tecnologías es de 314 segundos y representa más del diez por ciento del tiempo promedio que toma la ejecución del programa.



(a) Huygens en Cuaco (b) Trabajadores óptimos

Figura 5.4: Trabajadores para Huygens en Cuaco

El punto óptimo son 8 trabajadores. La Figura 5.3 (c) presenta un acercamiento al punto óptimo de la Figura 5.3 (a) y se observa la ganancia de alrededor de 11 segundos al utilizar una tecnología de red u otra.

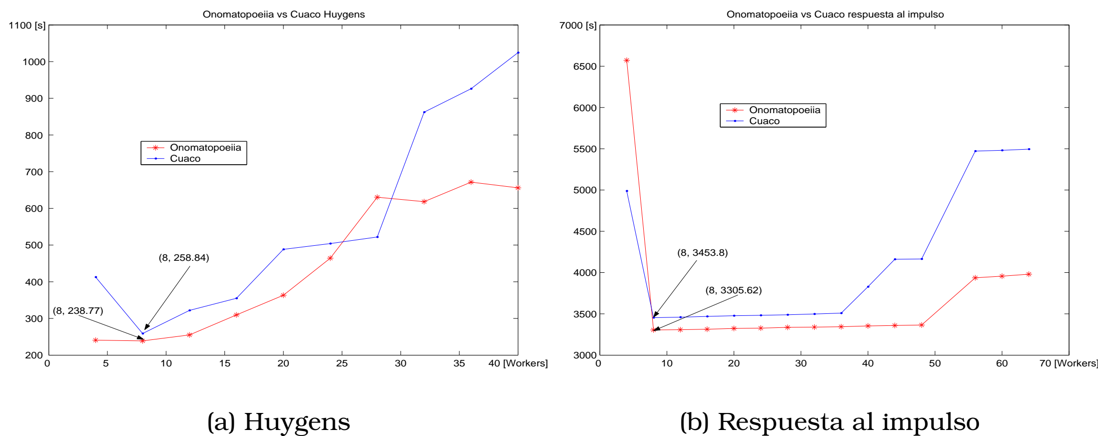
5.2.1.4 Ethernet VS SCI para Huygens en Cuaco.

Al igual que el *cluster* Onomatopoeiia, el punto óptimo está en 8 trabajadores, y el comportamiento de la gráfica es muy similar (ver Figura 5.4 (a)). Este programa no es la excepción, ya que aquí también la tecnología SCI gana tiempo a ethernet, un acercamiento al punto óptimo mostrado en la Figura 5.4 (b), se observa que la tecnología SCI reduce 40 segundos el tiempo total de proceso.

En el apéndice E Tablas E.2 y E.3 se presentan los datos numéricos de las pruebas para el método basado en el principio de Huygens y respuesta al impulso respectivamente, con el fin de comparar los valores de tiempo de ejecución total al usar las tecnologías SCI y ethernet y su desviaciones estándar.

5.2.2 Comparación de Clusters.

Para esta prueba es necesario tomar solamente 4 procesadores del cluster de Onomatopoeiia y utilizar la tecnología SCI para Cuaco y ethernet en Onomatopoeiia y comparar sus tiempos de procesos. Las Figuras 5.5 (a) y 5.5 (b) muestra la comparación de los dos *clusters*, para el método de Huygens y la respuesta al impulso respectivamente. En el apéndice E Tablas E.4 y E.5 se muestran los tiempos totales de proceso al variar el número de trabajadores. Para ambos casos el número de trabajadores que permitió el menor tiempo de ejecución es 8 trabajadores. A pesar del uso de la tecnología SCI en Cuaco, Onomatopoeiia obtiene mejores tiempos de ejecución y es debido a que tiene dos procesadores por nodo o tecnología Symmetrical Multiprocessor (SMP),



(a) Huygens

(b) Respuesta al impulso

Figura 5.5: Huygens y respuesta al impulso para ambos clusters

que logran un mejor desempeño en la ejecución de los procesos. El mismo comportamiento se observa para Huygens.

Con esta parte se concluye la primera etapa de pruebas, que tuvo el fin de determinar el número óptimo de trabajadores que logren el mejor tiempo en ambos programas. La siguiente etapa de pruebas se explica a continuación.

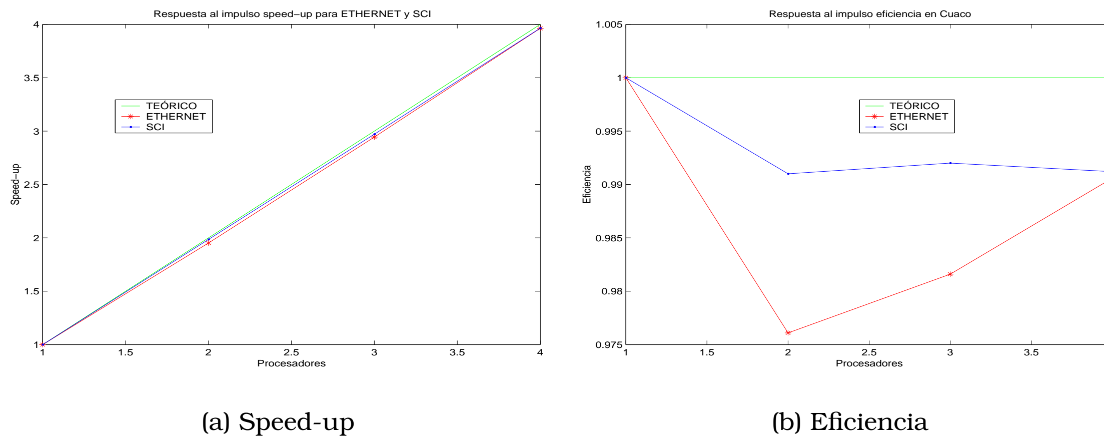
5.3 Métricas.

Una vez encontrado el número óptimo de trabajadores para cada programa y tipo de cluster, se toma como variable el número de procesadores, se hacen mediciones de tiempo y con ello se calculan métricas que ayudan a interpretar los resultados.

Las métricas para un sistema paralelo, se explicaron en el capítulo 2. Estas métricas permiten obtener información valiosa acerca de la ejecución de un programa en paralelo.

Las pruebas que se realizan en esta etapa son las siguientes:

- Cluster Cuaco
  - Huygens con ethernet variando de 1 a 4 procesadores
  - Huygens con SCI variando de 1 a 4 procesadores
  - Respuesta al Impulso con ethernet variando de 1 a 4 procesadores
  - Respuesta al Impulso con SCI variando de 1 a 4 procesadores
- Cluster Onomatopoeiia
  - Huygens variando de 1 a 8 procesadores
  - Respuesta al Impulso variando de 1 a 8 procesadores



(a) Speed-up

(b) Eficiencia

Figura 5.6: Respuesta al Impulso para eficiencia y speed-up en Cuaco

- Comparación de clusters
  - Respuesta al Impulso variando de 1 a 4 procesadores para ambos clusters

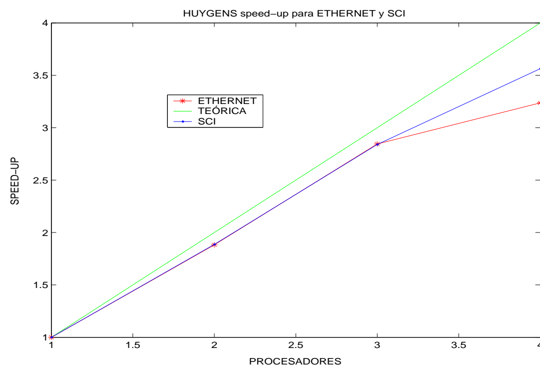
Para realizar estas pruebas se debe tomar la base de las pruebas anteriores, esto es, el número de trabajadores que optimizó cada programa, con el fin de dejar la variable de *número de trabajadores* fija en cada una de las pruebas.

La Figura 5.6 (a) y (b) muestran los resultados obtenidos al calcular las métricas *spped-up* y *eficiencia* para el método de la respuesta al impulso en el cluster Cuaco, al variar el número de procesadores y para ambas tecnologías de red. Las Tablas E.8 y E.9 contienen los valores de dichas métricas. En la Figura 5.7 (a) y (b) se grafican los valores obtenidos por las métricas *spped-up* y *eficiencia* para el método basado en el principio de Huygens sobre el cluster Cuaco y en ambas tecnologías de red, las Tablas E.6 y E.7 presentan los valores prácticos obtenidos para las diferentes métricas.

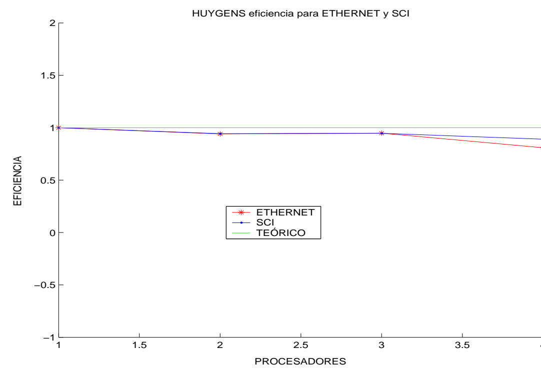
Para el cluster Onomatopoeiia, las Figuras 5.8 (a) y (b) y 5.9 (a) y (b) muestran los valores obtenidos para *speed-up* y *eficiencia* en los métodos respuesta al impulso y Huygens respectivamente. Para cada una de las Figuras anteriores se compara el valor teórico contra los valores prácticos. En el apéndice E Tablas E.10 y E.11 se presentan los valores numéricos de las métricas mencionadas.

5.4 Análisis de Resultados.

Lo ideal al paralelizar un programa es que no exista una parte serial y que todo el algoritmo sea paralelizable, esto es imposible y por ello ninguna métrica alcanza el valor teórico (ver Figura 5.11). El caso de la respuesta al impulso para el *cluster* Onomatopoeiia, es un buen ejemplo de lo dicho anteriormente. Si se observa la Figura 5.8 (b), todos los valores de eficiencia están por encima

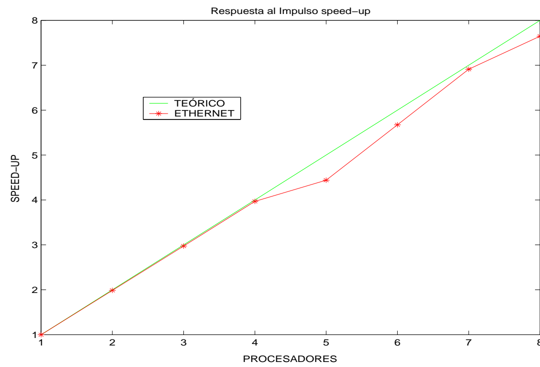


(a) Speed-up

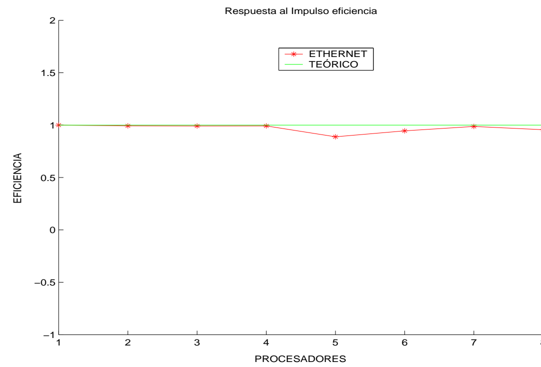


(b) Eficiencia

Figura 5.7: Huygens para eficiencia y speed-up en Cuaco

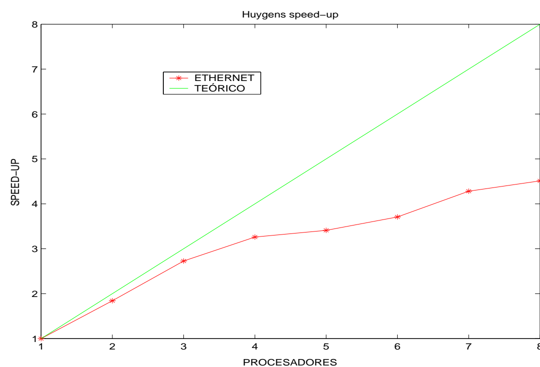


(a) Speed-up

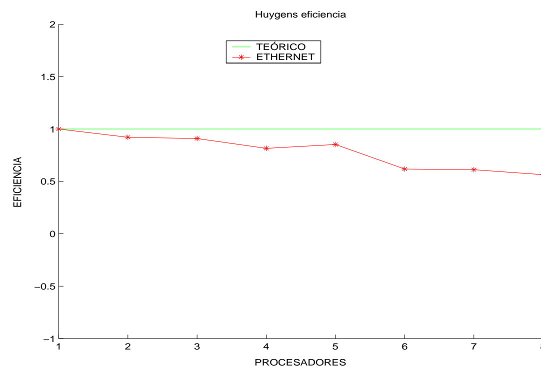


(b) Eficiencia

Figura 5.8: Respuesta al Impulso para eficiencia y speed-up en Onomatopoeiia

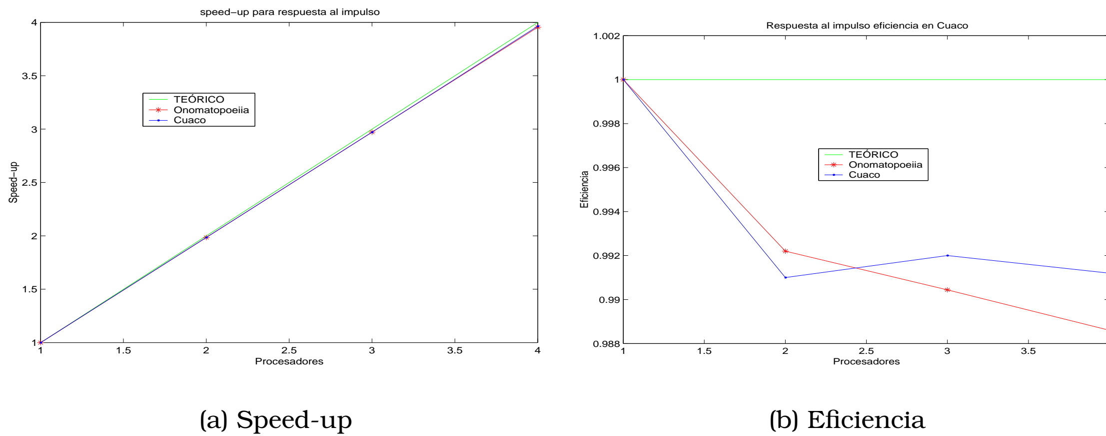


(a) Speed-up



(b) Eficiencia

Figura 5.9: Huygens para eficiencia y speed-up en Onomatopoeiia



(a) Speed-up (b) Eficiencia

Figura 5.10: Respuesta al impulso para eficiencia y speed-up en Onomatopoeiia y Cuaco

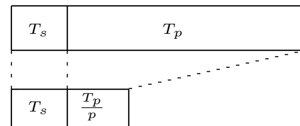


Figura 5.11: Tiempo serial y paralelo en un algoritmo.

del 90%, pero ninguno de ellos logra el valor teórico, y esto se debe a que existe comunicación entre los diferentes procesos, que aunque es muy pequeño comparado con el tiempo de proceso de cálculo, tiene un peso que le impide al algoritmo lograr el valor teórico.

Para la métrica *speed-up*, se espera una recta de $45°$, como valor ideal. Cuando dicha métrica se acerca al valor teórico, quiere decir que el tiempo de cálculo es mucho mayor que el tiempo de comunicación, y por lo tanto los recursos del *cluster* son aprovechados de forma correcta. Por otro lado, disminuciones significativas de esta métrica implican un tiempo grande de comunicación con respecto al tiempo de proceso de cálculo. El ejemplo más claro es el método de Huygens en el cluster Onomatopoeiia, la Figura 5.9 (a), muestra la caída del *seepd-up* a partir del cuarto procesador. Lo que quiere decir, que para este programa, más de cinco procesadores no logran una reducción de tiempo importante, debido a que la comunicación entre más procesadores es mayor que el tiempo de cálculo.

La variaciones de la *eficiencia*, proporcionan información del aprovechamiento de los recursos del sistema paralelo. Valores bajos de eficiencia se pueden deber a un algoritmo ineficiente que no va de acuerdo con la arquitectura del sistema, o a que la comunicación se incrementa para un número grande de procesadores o trabajadores impidiendo el uso correcto de los recursos. El mejor ejemplo es Huygens en Onomatopoeiia mostrado en la Figura 5.9 (b).

La comparación entre el *cluster* Onomatopoeiia y Cuaco (ver Figura 5.10)

para el método de la respuesta al impulso, se observa que Onomatopoeiia obtiene el menor tiempo en ejecución, además aprovecha de mejor manera los recursos del *cluster*. Esto se puede observar en las Tablas E.8 y E.12. Onomatopoeiia obtiene los valores más cercanos a los teóricos excepto para el cuarto procesador, donde Cuaco tiene un mejor desempeño, sin embargo Onomatopoeiia logra el menor tiempo de ejecución. Se concluye que Onomatopoeiia aprovecha de mejor manera los recursos de hardware del sistema, y se asume al hecho de que cuenta con dos procesadores por nodo, y estos logran un mejor control sobre los procesos.

Capítulo 6

Conclusiones.

El desarrollo de dos métodos que caracterizan la distribución de presión acústica, permite observar la similitud del comportamiento de las ondas ultrasónicas en el campo lejano a través del medio de propagación. Sin embargo existen considerables diferencias en el campo cercano, debidas a la rápidas variaciones del campo sonoro. Ambos métodos son aproximaciones, pero el método de la respuesta al impulso obtiene una mayor resolución del comportamiento sobre el campo cercano. La utilización de un método u otro, depende de la aplicación específica, ya que el método de Huygens computacionalmente requiere de un menor número de cálculos y conlleva a la obtención de la presión acústica de una forma más rápida. Sin embargo el método de la respuesta al impulso debe realizar un gran número de cálculos para obtener el comportamiento sonoro, pero se acerca más al sistema real. Además, si se caracterizan la mayor cantidad de planos para obtener un volumen de la distribución de presión acústica se mejora la resolución y permite distinguir con mayor precisión y de mejor forma el diseño y la aplicación de los transductores ultrasónicos para un campo específico.

Cuando un programa paralelo implementado bajo un lenguaje como C o Fortran se desea ejecutar en otro sistema paralelo, se requiere compilar el programa para el tipo de arquitectura y sistema operativo deseado. Al utilizar el lenguaje Python que es un intérprete y genera un “bytecode” logra un código “portable”, es decir, se puede ejecutar prácticamente en cualquier tipo de sistema paralelo en el que esté instalado. Además, utilizando la implementación del paradigma *Farming* hecho en Python y PVM (“framework”), se garantizó heterogeneidad entre arquitecturas y sistemas operativos.

Las diferentes tecnologías de red y la topología influyen en el desempeño de los sistemas paralelos, ya que de ellas depende el tiempo que utiliza el sistema para comunicarse o pasar mensajes. Este tiempo afecta las métricas de desempeño del sistema y ocasiona que no alcancen sus valores teóricos, debido al aumento del número de procesos de comunicación. La tecnología SCI comparada con ethernet, tiene menor latencia, pero ello implica una mayor inversión monetaria. La tecnología SCI reduce el tiempo total de comunicación; sin embargo, dicho tiempo no es significativo, debido a que en nuestro caso el tiempo de proceso de cálculo es mayor. Es importante no olvidar esto, ya que

si se tuviera un algoritmo donde la comunicación entre procesos es grande, la tecnología SCI permitirá el mejor desempeño sobre el algoritmo.

Las métricas de desempeño facilitan la observación del comportamiento del algoritmo paralelo y a su vez permiten ver el desempeño del sistema paralelo. La métrica de *speed-up* permitió observar las variaciones de tiempo con respecto al aumento de procesadores, por lo que es una buena métrica que indica cuántos procesadores aprovechan de mejor forma el sistema paralelo. Se concluye que valores por debajo del teórico se deben a que el tiempo de comunicación entre procesos comienza a crecer.

La *fracción serial* sirve como una métrica que a groso modo interpreta el grado de paralelismo del algoritmo, conforme se aumentan procesadores, esta métrica sólo es útil cuando se complementa su análisis con las demás técnicas de evaluación de desempeños.

Ninguna de las métricas usadas toman en cuenta detalles importantes como: la frecuencia de la máquina, el ancho de banda de la red, sobrecarga de comunicación y demás, por lo que sólo proporcionan un punto de partida para una análisis mas completo. Debido a lo dicho anteriormente, la única métrica clave en el paralelismo es el tiempo total de ejecución.

Cada programa tiene un punto óptimo que logra el mejor aprovechamiento de los recursos del *cluster*, esto no quiere decir que el menor tiempo en ejecución sea el punto óptimo, ya que los recursos del sistema no necesariamente son aprovechados al máximo para dicho tiempo a esta última parte se denomina precio desempeño.

Comparando el desempeño de ambos clusters, las métricas indican que el aprovechamiento de los recursos de los clusters es muy alto, debido a la naturaleza del problema, al tipo de paralelismo usado y a la eficiencia del algoritmo. Las diferencias entre dichas métricas son muy pequeñas, aunque el tiempo total de procesamiento para Onomatopoeiia es menor en todos los casos y se asume que se debe a la tecnología SMP empleada en este cluster.

6.1 Trabajos Futuros.

- La implementación del “framework” en un lenguaje como Java o jython, que garantice un menor tiempo en la ejecución de los algoritmos. Además se propone una modificación sobre la estructura del paradigma de *Farming*, donde el “farmer” no debe enviarle los trabajos al primer trabajador, si no que éste debe ver de frente a todos los trabajadores y cada uno de ellos debe ser capaz de comunicarse directamente con él y viceversa.

- Trabajar con el kernel versión 2.6.x, ya que éste tiene un control más eficiente en la ejecución de procesos para sistemas con dos o más procesadores en el mismo nodo. Esto es llamado afinidad del CPU.

- Medir de forma independiente los tiempos de proceso y de comunicación, para conocer más acerca del balanceo de cargas.

Apéndice A

Sistemas Digitales

Un sistema en tiempo discreto se define matemáticamente como una transformada u operador que transforma una secuencia de entrada con valores $x[n]$ en una secuencia de salida con valores $y[n]$ esto se puede expresar así:

$$y[n] = Tx[n] \tag{A.1}$$

y se muestra gráficamente en la figura A.1. La ecuación A.1 representa una regla para calcular los valores de la secuencia de salida a partir de los valores de la secuencia de entrada.

A.1 Sistemas lineales invariantes con el tiempo

La combinación de estas dos propiedades conduce a representaciones especialmente convenientes de los sistemas que las cumplen. La clase de los sistemas lineales está definido por el principio de superposición mostrada en B.1. Si la propiedad de linealidad se combina con la representación general de una secuencia como una combinación lineal de impulsos retrasados como se ve en la ecuación A.2, se puede concluir que un sistema queda completamente caracterizado por su respuesta al impulso. Concretamente sea $h_k[n]$ la respuesta del sistema a la entrada $d[n-k]$ un impulso en $n = k$. Entonces, aplicando la ecuación A.2

$$y[n] = T\sum_{k=-\infty}^{\infty} x[k]d[n-k] \tag{A.2}$$

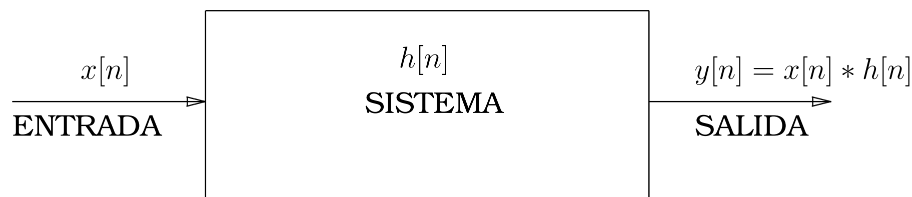


Figura A.1: Sistema digital.

cualquier secuencia se puede representar.

$$x[n] = \sum_{k=-\infty}^{\infty} x[k]d[n-k] \tag{A.3}$$

Aplicando el principio de superposición de la ecuación B.1

$$y[n] = T\sum_{k=-\infty}^{\infty} x[k]d[n-k] = \sum_{k=-\infty}^{\infty} x[k]h_k[n] \tag{A.4}$$

Al aplicar la propiedad de invariante en el tiempo implica que si $h[n]$ es la respuesta a $\delta[n]$ entonces $\delta[n-k]$ es $h[n-k]$, con esto la ecuación se convierte en

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{A.5}$$

Esto dice que dado $h[n]$ es posible utilizar A.5 para calcular $y[n]$ debida a cualquier entrada $x[n]$.

Esta operación se le denomina la convolución.

$$y[n] = x[n] * h[n]. \tag{A.6}$$

Apéndice B

Ondas Mecánicas

Un medio elástico se comporta como si estuviese constituido por una serie de partículas adyacentes cuyas fuerzas de atracción y repulsión mutua estuviesen completamente equilibradas. Si una de las partículas comienza a vibrar, las inmediatas a ella también lo harán, pero atrasadas en tiempo, con respecto a la que comenzó a vibrar. El movimiento vibratorio del medio que de esta forma se origina, se designa con el nombre de *movimiento ondulatorio*, y la forma que en un instante dado toma la totalidad de la perturbación recibe el nombre de *onda*.

El mundo está lleno de ondas, y los dos tipos principales son las ondas mecánicas y electromagnéticas. En el caso de las ondas mecánicas se necesita un medio que sea perturbado para su propagación, mientras que las ondas electromagnéticas no necesitan un medio para propagarse, y algunos ejemplos son la luz, las ondas de radio, las señales de televisión, etc. En el caso que implican ondas mecánicas, lo que se interpreta como una onda corresponde a la perturbación de un cuerpo o medio, por lo tanto se puede considerar a una onda como el movimiento de una perturbación.

Las ondas mecánicas requieren de una fuente de perturbación y un medio que pueda perturbarse. Tres características son importantes en la descripción de las ondas: la longitud de onda, la frecuencia, y la velocidad de onda. Una *longitud de onda* es la *distancia mínima entre dos puntos cualesquiera sobre una onda que se comportan idénticamente*, es decir, que se encuentran en la misma fase de su vibración. La *Frecuencia* de una onda es la *tasa de tiempo en la cual la perturbación se repite a sí misma*. La *velocidad* es la razón de cambio de la distancia con respecto al tiempo; ésta depende de las propiedades del medio, como la elasticidad y la densidad y puede calcularse por el producto de la *longitud de onda* con la *frecuencia* $v = \lambda * f$.

B.1 Superposición

Si dos o más ondas viajeras se mueven a través de un medio, la función de onda resultante en cualquier punto es la suma algebraica de las funciones de

onda de las ondas individuales.

$$T[a_1 x_1(n) + b_2 x_2(n)] = a_1 T[x_1(n)] + b_2 T[x_2(n)] \tag{B.1}$$

B.2 Interferencia

Este fenómeno se produce cuando dos movimientos ondulatorios que actúan sobre un punto dado se refuerzan o se reducen mutuamente, según coincidan en él las vibraciones de idéntica o opuesta fase, llamándose interferencia constructiva o destructiva, respectivamente.

B.3 Difracción

Fenómeno en el cual se observa que la onda se desvía al bordear el obstáculo, y produce patrones de interferencia.

Apéndice C

Cluster Beowulf.

El proyecto Beowulf nació en 1994 por *Tomas Sterling y Don Becker*, trabajando para CESDIS (*Center of Excellence in Space Data and Information Sciencies*) de la NASA, bajo el patrocinio del proyecto de la tierra y ciencias del espacio (ESS), construyeron un sistema paralelo, que consistía de *16 procesadores DX4*. Llamaron a su máquina ***Cluster Beowulf***. La máquina fue un éxito inmediato para satisfacer requisitos de cómputo específicos y a bajo precio.

C.1 Socket.

El término de socket se usa para vincular la comunicación entre dos procesos. Usa una quíntupla para establecer dicha comunicación:

- Protocolo
- Dirección local
- Proceso local
- Dirección remota
- Proceso remoto

C.2 Makefile

Make es una herramienta que controla los procesos de creación de ejecutables y revisión de archivos a partir de los archivos fuente. Esta utilería obtiene conocimiento de cómo construir programas o aplicaciones mediante un script llamado *makefile*, que lista cada operación necesaria para obtener los resultados a partir de sus dependencias. Entre las capacidades de *make* se cuentan:

- Hace posible que el usuario final compile, instale o actualice archivos sin conocer en detalle cómo es tal proceso.

- Decide automáticamente que archivos deben actualizarse, basándose en los archivos fuentes modificados.
- Determina automáticamente el orden en que la actualización debe hacerse, como el resultado de cambiar algunas líneas de código y ejecutar nuevamente *make*. Éste no necesita compilar todo el proyecto, sólo actualiza aquellos objetos que sean necesarios.

El *make* no está limitado a ningún lenguaje en particular, ni a compilar sólo aplicaciones. Se puede usar para instalar o desinstalar programas, actualizar archivos con dependencias o cualquier otra cosa que se requiera automatizar.

Para estas pruebas, *make* permite tener control sobre los archivos de resultados y realización de las pruebas de forma automatizada con la utilización de un sólo comando. En caso de que alguna prueba fallara, solamente es necesario volver a ejecutar el comando *make* para que las pruebas se reanuden.

El cuerpo de un archivo *makefile* tiene el siguiente formato:

```
objetivo: dependencia 1 ... dependencia n
          comando
```

Para este caso cada línea del *makefile* es una prueba a realizar con características específicas. El `objetivo` es el archivo de resultados que generará cada prueba. La `dependencia` es el programa que se ejecuta para llegar al resultado y el `comando` es una llamada al lenguaje Python. Como se muestra en el archivo D.8, se observa que es necesario crear las `dependencias` de acuerdo a las pruebas a realizarse. Para lograr esto se usa un *shell script* (ver apéndice D.9) que toma información del archivo llamado *parametros* (ver apéndice D.10). Dicho archivo contiene las variables para realizar las diferentes pruebas, esto es, el número de trabajadores (procesos), archivo de salida y máquinas donde se distribuyen las tareas.

Finalmente, a partir de los archivos de resultados que genera el *makefile* se obtiene un par de vectores:

- Tiempo de prueba y
- Número de trabajadores utilizados

Los cuales sirven para generar gráficas, tablas y cálculos necesarios para la interpretación y análisis.

C.3 PyFarm.

El framework pyfarming es la implementación del paradigma de farming en el lenguaje Python e interactúa con PVM a través de un módulo de Python llamado pypvm. La clase farmer inicializa los objetos kernel, switch y catcher, como se explica a continuación:

El constructor de farmer (__init__) requiere de 5 argumentos:

- NumberOfWorkers: Es el número de trabajadores que se instancian sobre la máquina virtual.
- NBloques: Es el número de tareas a realizar.
- host: Es una cadena, que corresponde a los nombres o direcciones IP de las máquinas que componen la máquina virtual.
- path: Es la ruta donde se encuentra el programa a realizar
- debug: Es una bandera de PVM, que le dice si inicia en modo de corrección o no.

Al instanciarse el objeto farmer, se inicializan “NumberOfWorkers” procesos kernel y switch como se ilustra en la Figura 2.8.

Las funciones miembro utilizadas en el presente trabajo son:

```
def send(self,i,data):
```

El argumento `i`, contiene el i-ésimo bloque a procesar, `data` contiene el paquete o mensaje. La función `send`, recibirá los parámetros `i, data`, con el fn de que sean enviados al switch del primer trabajador. Esto lo logra por medio de la funcion `pypvm.send`.

```
def wait(self):
```

Recibe las tareas enviadas por el catcher, a través de la función `recv` de PVM hasta que la bandera WorkDone sea enviada. Una vez recibida, la función `upk` desempaqueta las tareas con la finalidad de entregárselas al farmer.

```
def mastrab(self):
```

Recibe una petición para enviar más tareas. Con esta petición se debe llamar a la función send para enviar más tareas. Ésta función tiene el propósito de sincronizar el envío de paquetes.

Apéndice D

Programas.

D.1 Respuesta al Impulso Secuencial.

```
from scipy import *
from time import time
import pickle
from baffles import *
from string import split,atof

quad = integrate.quad
########################### carga velocidad ##################################
arch="waveform_cara.txt" # valores de la volocidad el pistón
ar = open(arch,"r")

lins = ar.readlines()
tmp = []
for i in lins:
    cad = split(i)
    tmp.append(atof(cad[1]))
sig = array(tmp,Float)


########################## variables de entrada ##############################
t11 = time() #inicio del contador de tiempo
f  = 1.0e6   # frec. de resonancia [Hz]
c  = 1500.0   # Vel. del sonido en agua [m/s]
fs = 25.0e6  # Frec. de muestreo [Hz]
ts = 1/fs
lam= c/f    # longitud de onda [m]
a  = .5*25.4e-3/2  # Radio del transductor [m]


dz   = .636e-3 #paso sobre z
x    = arange(-a,a+dz,dz)  # diámetro del transductor
y    = 0
z1   = arange(dz,1e-3,dz) # longitud sobre el eje z
# matrices para almacenar la presión
Pp   = zeros((len(x),len(z1)),Float)
Ppp  = zeros((len(x),len(z1)),Float)
Prms = zeros((len(x),len(z1)),Float)

################################# ciclos ####################################
# método propuesto por Rodríguez y colaboradores para el cálculo de la
# respuesta al impulso
for inx in range(0,len(x)):
    print "inx -> " + `inx`
    L  = sqrt(x[inx]**2 + y**2)
```

```
for inz in range(0,len(z1)):
    print "inz -> " + `inz` + " inx -> " + `inx`
    z  = z1[inz]
    t1 = sqrt(z**2 + (a-L)**2) / c
    t2 = sqrt(z**2 + (a+L)**2) / c
    if(L > a):
        tmin = t1
    else:
        tmin = z / c
    t0=arange(tmin-20*ts,t2+20*ts,ts)
    H=zeros(len(t0),Float)
    i = 0
    for t in arange(t0[0],max(t0),ts):
        if t < tmin:
            print 1
            H[i] = 0.0

        elif t >= tmin and t < t1:

            func1 = quad(lambda v: z / v**2 , tmin, t, epsabs=1.48e-3, epsrel=1.48e-3)
            print 2
            H[i] = (z / t) + func1[0]




        elif t >= t1 and t <= t2:

            func3 = quad(lambda v: z / v**2 , tmin , t1, epsabs=1.48e-5, epsrel=1.48e-5)
            func3 = func3[0]


            func2 = quad(baffle,t1,t,args=(z,a,c,L), epsabs=1.48e-5, epsrel=1.48e-5)
            func2 = func2[0]



            if ((L**2 + c**2 * t**2 - z**2 - a**2)/(2*L*sqrt(c**2*t**2-z**2)))> 1:
                  func4=(z/(t*pi))*arccos(1)
            elif ((L**2 + c**2 * t**2 - z**2 - a**2)/(2*L*sqrt(c**2*t**2-z**2)))< -1:
                  func4=(z/(t*pi))*arccos(-1)
            else :
                  func4 = (z/(t*pi))*arccos((L**2 + c**2 * t**2 - z**2 - a**2) /
                          (2 * L *sqrt(c**2*t**2 - z**2)))

            print 3
            H[i] = func2 + func3 + func4

        else:

            func5 = quad(lambda v: z/v**2,tmin,t1, epsabs=1.48e-3, epsrel=1.48e-3)
            func5 = func5[0]
            func6 = quad(baffle,t1,t2,args=(z,a,c,L), epsabs=1.48e-3, epsrel=1.48e-3)
            func6 = func6[0]
            print 4
            H[i] = func5 + func6

        i = i + 1

    # cálculo de la presión puntual.
    # derivada de la respuesta al impulso
    dH = diff( abs(H) / c)
    # convolución de la derivada de la respuesta al impulso
    pressure = convolve(sig,dH)
    Vp   = max(abs(max(pressure)), abs(min(pressure)))
    Vpp  = max(pressure) - min(pressure)
    Vrms = sqrt(mean(pressure**2))
    Ppp[inx,inz] = Vpp
    Pp[inx,inz] = Vp
    Prms[inx, inz] = Vrms
```

```
t22 = time()  #cálculo del tiempo que le toma al programa ejecutarse

################################ salvar datos ################################
pickle.dump(Pp, open("Pp.pic","wb"))
pickle.dump(Ppp, open("Ppp.pic","wb"))
pickle.dump(Prms, open("Prms.pic", "wb"))
print " El tiempo de es -> " + `t22-t11` + " segundos"
```

D.2 Respuesta al Impulso Paralelo.

```
from farmer import Farmer
import zlib
from scipy import *
from time import time
import pickle
from baffles import *
from string import split,atof

quad = integrate.quad

####################################  carga velocidad ######################
arch="waveform_cara.txt"
ar = open(arch,"r")

lins = ar.readlines()
tmp = []
for i in lins:
    cad = split(i)
    tmp.append(atof(cad[1]))
sig = array(tmp,Float)


#########################  variables de entrada #########################
t11 = time()
f  = 1.0e6   # frec de resonancia [Hz]
c  = 1500.0   # Vel. del sonido en agua [m/s]
fs = 25.0e6  # Frec. de muestreo [Hz]
ts = 1.0/fs   # Periódo
lam= c/f    # longitud de onda [m]
a  = .5*25.4e-3/2  # Radio del transductor [m]

dz   = .636e-3 #paso sobre z
x    = arange(-a-20*dz,a+20*dz,dz) # diámetro del transductor
y    = 0
z1   = arange(dz,200e-3,dz)
Pp   = zeros((len(x),len(z1)),Float)
Ppp  = zeros((len(x),len(z1)),Float)
Prms = zeros((len(x),len(z1)),Float)

Ntrab = 3   # número de trabajadores a instanciar
# directorio en el cual se localizan los programas
path = "/mnt/home/frank/pry_imp/"
farm = Farmer(Ntrab,len(z1)*len(x),None,path,1) # instancia del farmer
raw_input("Se inicializo el Farmer")
inicio = 1
t11 = time() # inicio del contador de tiempo

for inx in range(0,len(x)):
    print "inx -> " + `inx`
    L  = sqrt(x[inx]**2 + y**2)
    for inz in range(0,len(z1)):
        print "inz -> " + `inz` + " inx -> " + `inx`
        z  = z1[inz]
        # sincronización de envio de mensajes
```

```
        if inz > 2*Ntrab-1:
              inicio = 0
        if inicio:
             # paquete o mensaje a enviar
             parametros = {"L":L,"z":z1[inz],"a":a,"c":c,"ts":ts,"func":"baffpar",
                               "mod":"baffles","sig":zlib.compress(sig.tostring()) }
             # función para enviar mensajes
             farm.send((inx,inz),parametros)
             print "Enviando los primeros trabajos -> " + `inz`
        else:
             farm.mastrab() # función para pedir más trabajo a realizar
             parametros = {"L":L,"z":z1[inz],"a":a,"c":c,"ts":ts,"func":"baffpar",
                                "mod":"baffles","sig":zlib.compress(sig.tostring()) }
             farm.send((inx,inz),parametros) # función para enviar mensajes


Res=farm.wait()  # función que espera todos las tareas terminadas.

t22 = time() # cálculo del tiempo del programa
# ciclo para ordenar las tareas entregadas.
for i in Res.keys():
    Pp[i[0],i[1]] = Res[i]["Vp"]
    Ppp[i[0],i[1]]  = Res[i]["Vpp"]
    Prms[i[0],i[1]] = Res[i]["Vrms"]

pickle.dump(Pp, open("Pp.pic","wb"))
pickle.dump(Ppp, open("Ppp.pic","wb"))
pickle.dump(Prms, open("Prms.pic", "wb"))

print " El tiempo de es -> " + `t22-t11` + " segundos"
```

D.3 Función baffpar

```
from scipy import *
from farming import Work

# función llamada en baffpar

def baffle(v,z,a,c,L): #parámetros de entrada
     try:
         if (( L**2+c**2*v**2-z**2-a**2)/(2*L*sqrt(c**2*v**2-z**2))) > 1:
              return  (z/(pi*v**2))*arccos(1)
         elif ((L**2+c**2*v**2-z**2-a**2)/(2*L*sqrt(c**2*v**2-z**2))) < -1:
              return (z/(pi*v**2))*arccos(-1)
         else:
              return  (z/(pi*v**2))*arccos((L**2+c**2*v**2-z**2-a**2)/
                      (2*L*sqrt(c**2*v**2 - z**2)))

     except:
              return 0.0


# función baffpar
def baffpar(msg):
    import zlib
    #parámetros pasados en el mensaje
    nb = msg[0] # número de tareas
    L  = msg[1]["L"]
    z  = msg[1]["z"]
    a  = msg[1]["a"]
    c  = msg[1]["c"]
    ts = msg[1]["ts"]
    sig = fromstring(zlib.decompress(msg[1]["sig"]),Float)
    # implementación de la respuesta al impulso propuesta por
    # Rodríguez y colabodores
    quad = integrate.quad
```

```
t1 = sqrt(z**2 + (a-L)**2) / c
t2 = sqrt(z**2 + (a+L)**2) / c
if(L > a):
    tmin = t1
else:
    tmin = z / c
t0=arange(tmin-20*ts,t2+20*ts,ts)
H=zeros(len(t0),Float)
i = 0
for t in arange(t0[0],max(t0),ts):
      if t < tmin:

         H[i] = 0.0

      elif t >= tmin and t < t1:

         func1 = quad(lambda v: z / v**2 , tmin, t,
                 epsabs=1.48e-3, epsrel=1.48e-3)

         H[i] = (z / t) + func1[0]

      elif t >= t1 and t <= t2:

         func3 = quad(lambda v: z / v**2,tmin,t1, epsabs=1.48e-5, epsrel=1.48e-5)
         func3 = func3[0]


         func2 = quad(baffle,t1,t,args=(z,a,c,L), epsabs=1.48e-5, epsrel=1.48e-5)
         func2 = func2[0]


         try:
          if((L**2 + c**2 * t**2 - z**2 - a**2) / (2 * L *sqrt(c**2*t**2 - z**2)))> 1:
                  func4=(z/(t*pi))*arccos(1)
          elif((L**2 + c**2 * t**2 - z**2 - a**2) / (2 * L *sqrt(c**2*t**2 - z**2)))< -1:
                  func4=(z/(t*pi))*arccos(-1)
          else :
                func4 = (z/(t*pi))*arccos((L**2 + c**2 * t**2 - z**2 - a**2) /
                          (2 * L *sqrt(c**2*t**2 - z**2)))

         except:
                 func4=0.0

         H[i] = func2 + func3 + func4

      else:


         func5 = quad(lambda v: z/v**2,tmin,t1, epsabs=1.48e-3, epsrel=1.48e-3)
         func5 = func5[0]
         func6 = quad(baffle,t1,t2,args=(z,a,c,L), epsabs=1.48e-3, epsrel=1.48e-3)
         func6 = func6[0]

         H[i] = func5 + func6

      i = i + 1
# deriva la respuesta al impulso
dH = diff((H)/c)
# convoluciona la derivada de la respuesta al impulso
pressure = convolve(sig,dH)
Vp   = max(abs(max(pressure)), abs(min(pressure)))
Vpp  = max(pressure) - min(pressure)
Vrms = sqrt(mean(pressure**2))
return (nb,{"Vp":Vp,"Vpp":Vpp,"Vrms":Vrms})
```

D.4 Huygens Secuencial.

```
#!/usr/bin/env python2.3


from math import *
from scipy import *
import pickle

########################## entradas #######################################
a=6e-3    # radio del transductor
f=1e6     # frecuencia de resonancia
c=1500    # velocidad del sonido en agua
lam=c/f   # longitud de onda
dy=lam/4  # separación de los emisores
fs=5e6    # frecuencia de muestreo
ts=1/fs
t=arange(0,1000*ts,ts)  # tiempo experimento
lent=len(t)
t1=3/f
x=[]
y=[]
env=signal.signaltools.hanning(floor(t1/ts))

############################# transductor ####################################
# crea los emisores o puntos de vibración
for r in arange(a, dy,-dy):
      theta1=acos(1-dy**2/(2*r**2))
      for thet in arange(0,2*pi,theta1):
           x.append(r*cos(thet))
           y.append(r*sin(thet))

trans=array([x,y])
trans=transpose(trans)

sztot, col =shape(trans)

A=1e-3
ty=arange(0,t1-ts,ts)
pulse=A*env*cos(2*pi*f*ty)
lenp=len(pulse)

yT=0
xT=arange(-a-20*dy, a+20*dy, dy) # recorrido en el eje x
zT=arange(.5, 100.5, .5)*1e-3  # recorrido en el eje z
lenx=len(xT)
lenz=len(zT)
# inicialización de las matrices de presión
Pp=zeros((lenx, lenz))*0.0
Ppp=zeros((lenx, lenz))*0.0
Prms=zeros((lenx, lenz))*0.0

#############################  ciclos ####################################
# suma algebraica de las señales emitidas por los emisores con
# respecto a un punto de observación
for inx in range(0,lenx):
        press=zeros((lenz,lent))*0.0
        for inz in range(0,lenz,):
                sig=zeros((sztot, lent))*0.0
                for insig in range(0,sztot):
                          sens=trans[insig,:]
                          xi=sens[0]
                          yi=sens[1]
                          ri=sqrt((xT[inx]-xi)**2+(yT-yi)**2+zT[inz]**2)
                          ti=floor(ri/(c*ts))
                          on1=arange(0,ti)*0.0
                          on2=arange(0,lent-lenp-len(on1))*0.0
                          delpul=concatenate([on1, pulse/float(ri), on2])
```

```
                          sig[insig,:]=delpul[:]


                press[inz,:]=sum(sig)
                Vp=max(abs(max(press[inz,:])),abs(min(press[inz,:])))
                Vpp=max(press[inz,:])-min(press[inz,:])
                Vrms=sqrt(mean(press[inz,:]**2))
                Ppp[inx,inz]=Vpp
                Pp[inx,inz]=Vp
                Prms[inx,inz]=Vrms

        inx

######################### salva datos ####################################

pickle.dump(Pp ,open("Pp.pik","wb"))
pickle.dump(Ppp, open("Ppp.pik","wb"))
pickle.dump(Prms,open("Prms.pik","wb"))

#if __name__ == "__main__":
#
```

D.5 Huygens Paralelo

```
#!/usr/bin/env python2.3

from farmer import Farmer
from time import time
from scipy import *
import pickle
from interno import *

######################## entradas ########################################
a=6e-3    # radio del transductor
f=1e6     # frecuencia de resonancia
c=1500    # velocidad del sonido en agua
lam=c/f   # longitud de onda
dy=lam/4  # separación de los emisores
fs=5e6    # frecuencia de muestreo
ts=1/fs
t=arange(0,1000*ts,ts)
lent=len(t)
t1=3/f
x=[]
y=[]
env=signal.signaltools.hanning(floor(t1/ts))

############################# transductor ###################################

# crea los emisores o puntos de vibración
for r in arange(a, dy,-dy):
      theta1=arccos(1-dy**2/(2*r**2))
      for thet in arange(0,2*pi,theta1):
           x.append(r*cos(thet))
           y.append(r*sin(thet))

trans=array([x,y])
trans=transpose(trans)

sztot, col =shape(trans)

A=1e-3
ty=arange(0,t1-ts,ts)
pulse=A*env*cos(2*pi*f*ty)
lenp=len(pulse)
```

```
yT=0
xT=arange(-a-dy, a+dy, dy)
zT=arange(.5, 50.5, .5)*1e-3
lenx=len(xT)
lenz=len(zT)

Pp=zeros((lenx, lenz),Float)
Ppp=zeros((lenx, lenz),Float)
Prms=zeros((lenx, lenz),Float)

Ntrab = 2 # trabajadores a instanciar
# directorio donde se encuentra los programas
path = "/mnt/home/frank/python/tesis/cir/"
farm = Farmer(Ntrab,len(zT)*len(xT),None,path,1) #instancia del farmer
raw_input("Se inicializo el Farmer")
inicio = 1
t11 = time() # inicia contador de tiempo

for inx in range(0,lenx):

        for inz in range(0,lenz,):
                print "inz -> " + `inz` + " inx -> " + `inx`
                # sincronización de los mensajes a enviar
                if inz > 2*Ntrab-1:
                    inicio = 0
                if inicio:
                     # mensaje a enviar
                     parametros = {"xT":xT[inx],"zT":zT[inz],"trans":trans,"lent":lent,
                     "sztot":sztot,"c":c,"ts":ts,"pulse":pulse,"func":"ciclo","mod":"interno"}
                     # función para enviar el mensaje en la máquina virtual
                     farm.send((inx,inz),parametros)
                     print "Enviando los primeros 2*Ntrabajadores -> "
                else:
                     # función para pedir más trabajo
                     farm.mastrab()
                     # mensaje a enviar
                     parametros = {"xT":xT[inx],"zT":zT[inz],"trans":trans,"lent":lent,
                     "sztot":sztot,"c":c,"ts":ts,"pulse":pulse,"func":"ciclo","mod":"interno"}
                     # envia mensaje a la máquina virtual
                     farm.send((inx,inz),parametros)

# función que espera que el catcher le entrege todas las tareas
Res=farm.wait()

t22 = time() # almacena tiempo del programa
# función para acomodar los mensajes
for i in Res.keys():
    Pp[i[0],i[1]] = Res[i]["Vp"]
    Ppp[i[0],i[1]]  = Res[i]["Vpp"]
    Prms[i[0],i[1]] = Res[i]["Vrms"]

pickle.dump(Pp , open("Pp.pik","wb"))
pickle.dump(Ppp, open("Ppp.pik","wb"))
pickle.dump(Prms,open("Prms.pik","wb"))

print " El tiempo de es -> " + `t22-t11` + " segundos"

#if __name__ == "__main__":
#
```

D.6 Mensaje en Huygens

```
from scipy import *
from farming import *
```

```
# función independiente que se ejecuta en cada procesador
# de la máquina virtual.

def ciclo(msg):
    #parámetros de entrada pasados en el mensaje
    nb = msg[0]
    xT  = msg[1]["xT"]
    zT  = msg[1]["zT"]
    trans  = msg[1]["trans"]
    lent  = msg[1]["lent"]
    sztot = msg[1]["sztot"]
    c = msg[1]["c"]
    ts = msg[1]["ts"]
    pulse = msg[1]["pulse"]
    yT = 0
    lenp = len(pulse)
    sig=zeros((sztot, lent),Float)
    # suma algebraica de las señales enviadas por los emisores con
    # respecto a cada punto de observación
    for insig in range(0,sztot):
           sens=trans[insig,:]
           xi=sens[0]
           yi=sens[1]
           ri=sqrt((xT-xi)**2+(yT-yi)**2+zT**2)
           ti=floor(ri/(c*ts))
           on1=arange(0,ti)*0.0
           on2=arange(0,lent-lenp-len(on1))*0.0
           delpul=concatenate([on1, pulse/float(ri), on2])
           sig[insig,:]=delpul[:]


    press = sum(sig)
    Vp=max(abs(max(press)),abs(min(press)))
    Vpp=max(press)-min(press)
    Vrms=sqrt(mean(press**2))

    return (nb,{"Vp":Vp,"Vpp":Vpp,"Vrms":Vrms})
```

D.7 Clase Farmer

```
#!/usr/bin/env python2.3

from sys import stdout
from farming import *
import pypvm
from Numeric import concatenate

kernel  = "kernel.py"
switch  = "switch.py"
catcher = "catcher.py"

class Farmer:
    def __init__(self,NumberOfWorkers = 3, NBloques = 1, host = [''], path = "", debug = 0):
        self.NBloques = NBloques
        self.mytid = pypvm.mytid()
        self.NumberOfWorkers = NumberOfWorkers
        if debug :
            pypvm.catchout(stdout)

        #inicializando los componentes: switches y kernels

        if len(host) == 1 and host[0] == 'None':

           self.tidsSw     = pypvm.spawn(path + switch, [], pypvm.spawnOpts['TaskDefault'],"",
                              NumberOfWorkers)
           self.tidsK      = pypvm.spawn(path + kernel, [], pypvm.spawnOpts['TaskDefault'],"",
```

```
                          NumberOfWorkers)
         self.tidCatcher = pypvm.spawn(path + catcher,[], pypvm.spawnOpts['TaskDefault'],
                                     "",1)

      else: ## Se crean en el mismo host del parent

          Nxn = NumberOfWorkers/len(host)
          Nresto = NumberOfWorkers%len(host)
   self.tidsSw = []
   self.tidsK = []

          for nodo in host:
self.tidsSw.append(pypvm.spawn(path + switch, [], pypvm.spawnOpts['TaskHost'],
                                nodo, Nxn))
              self.tidsK.append(pypvm.spawn(path + kernel, [], pypvm.spawnOpts['TaskHost'],
                                nodo, Nxn))
          #ordena los kernels y switches que no son divisibles
          if Nresto > 0:
            for i in range(Nresto):
self.tidsSw.append(pypvm.spawn(path + switch, [], pypvm.spawnOpts['TaskHost'],
                                    host[i],Nresto))
              self.tidsK.append(pypvm.spawn(path + kernel, [], pypvm.spawnOpts['TaskHost'],
                                    host[i],Nresto))
          self.tidCatcher = pypvm.spawn(path + catcher,[], pypvm.spawnOpts['TaskHost'],
                                     host[0], 1)

          #Concatenando la lista de tids
   self.tidsSw = concatenate(self.tidsSw)
   self.tidsK = concatenate(self.tidsK)

      # Se inicialisa el catcher
      pypvm.initsend(pypvm.data['default'])
      pypvm.pk({"NBloques":self.NBloques})
      pypvm.send(self.tidCatcher[0],SC)



      # Inicializamos el pipe menos el último miembro
      for  i in xrange(NumberOfWorkers-1):
          pypvm.initsend(pypvm.data['default'])
          pypvm.pk({"ToNext_ID":self.tidsSw[i+1],"Catcher_ID":self.tidCatcher[0],
                  "Kernel_ID":self.tidsK[i],"Switch_ID":self.tidsSw[i]})
          pypvm.send(self.tidsSw[i],SK)
          pypvm.send(self.tidsK [i],SK)

      # Inicializamos el ultimo miembro del pipe
      pypvm.initsend(pypvm.data['default'])
      pypvm.pk ({"ToNext_ID":self.tidsSw[0],"Catcher_ID":self.tidCatcher[0],
                 "Kernel_ID":self.tidsK[NumberOfWorkers-1],
                 "Switch_ID":self.tidsSw[NumberOfWorkers-1]})
      pypvm.send(self.tidsSw[NumberOfWorkers-1],SK)
      pypvm.send(self.tidsK [NumberOfWorkers-1],SK)


  def mastrab(self):
      bufid = pypvm.recv(-1,MoreWork)
      if bufid:
          return 1
      else:
          return 0


  def wait(self):
      bufid = pypvm.recv(self.tidCatcher[0],WorkDone)
      msg = pypvm.upk()
      pypvm.freebuf(bufid)
      return msg


  def send(self,i,data):
```

```
        pypvm.initsend(pypvm.data['default'])
        pypvm.pk({ToNext:(i,data)})
        pypvm.send(self.tidsSw[0],ToNext)
```

D.8 Makefile

```
RESULTADOS = res1 res2 res3 res4 res5 res6 res7 res8 res9 res10
PYTHON = /usr/local/bin/python2.3
ALL : ${RESULTADOS}
res1 :
 ${PYTHON} hpres1.py
res2 :
 ${PYTHON} hpres2.py
res3 :
 ${PYTHON} hpres3.py
res4 :
 ${PYTHON} hpres4.py
res5 :
 ${PYTHON} hpres5.py
res6 :
 ${PYTHON} hpres6.py
res7 :
 ${PYTHON} hpres7.py
res8 :
 ${PYTHON} hpres8.py
res9 :
 ${PYTHON} hpres9.py
res10 :
 ${PYTHON} hpres10.py$
```

D.9 Hace Makefile

```
#Arma los archivos para pruebas que ejecuta python
while read N res host
do
  archivo='hp'$res'.py'
  echo 'import pickle' > $archivo
  echo 'import i_r' >> $archivo
  echo 'p = {"N":'$N', "host": '$host'}' >> $archivo
  echo 'f = open("parametros.'$res'.est", "wb")' >> $archivo
  echo 'pickle.dump(p,f)' >> $archivo
  echo 'f.close()' >> $archivo
  echo 'N = p["N"]'  >> $archivo
  echo 'host = p["host"]'  >> $archivo
  echo 'i_r.miFarmer(N, "'$res'", host)'  >> $archivo
done < parametros

#Arma el Makefile
resultados='RESULTADOS = '
while read N res host
do
  resultados=$resultados' '$res' '
done < parametros
echo $resultados > Makefile
echo 'PYTHON = /usr/local/bin/python2.3' >> Makefile
echo 'ALL : ${RESULTADOS}' >> Makefile
while read N res host
do
```

```
  echo $res' :' >> Makefile
  echo ' ${PYTHON} hp'$res'.py' >> Makefile
done < parametros$
```

D.10 Parámetros

```
# variables de entrada para cada prueba.
# Ejemplo para el cluster de 8 nodos variando el número de
# trabajadores a instanciar en la máquina virtual

8 res1 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
16 res2 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
24 res3 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
32 res4 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
40 res5 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
48 res6 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
56 res7 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
64 res8 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
72 res9 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
80 res10 "['n1', 'n2', 'n3', 'n4', 'n5', 'n6', 'n7', 'n8']"
```

Apéndice E

Tablas de las Pruebas.

BYTES	TIEMPO ETHERNET [s]	TIEMPO SCI [s]
1	0.000067	0.000047
19	0.000070	0.000048
61	0.000082	0.000049
192	0.000100	0.000051
515	0.000157	0.000056
3069	0.000458	0.000273
12285	0.001255	0.001019
65533	0.005878	0.005278
262141	0.023205	0.021522
786432	0.069447	0.064991
12582915	1.105405	1.039574

Tabla E.1: Latencias para ethernet y SCI

TRABAJADORES	ETHERNET [s]	SCI [s]	STD de ETHERNET [s]	STD de SCI [s]
4	240.726	239.587	2.408	1.585
8	238.775	200.125	0.904	1.093
12	254.985	250.996	2.971	0.851
16	309.655	313.519	4.094	2.102
20	363.304	379.977	4.208	2.904
24	464.315	396.457	2.364	4.515
28	630.423	407.674	2.071	4.281
32	618.192	583.644	2.570	2.301
36	671.504	695.226	1.932	3.387
40	655.958	680.370	3.183	2.963
44	702.252	694.722	2.536	2.639
48	703.663	706.055	4.085	6.569
52	617.964	517.154	6.426	3.122
56	691.745	553.948	2.006	2.460
60	750.358	604.637	1.669	2.381
64	773.055	643.873	2.735	2.791
68	764.079	686.640	3.962	3.609
72	754.211	729.876	1.215	1.578

Tabla E.2: Huygens ethernet VS SCI con desviación estándar para Cuaco

TRABAJADORES	ETHERNET 1e3[s]	SCI 1e3[s]	STD de ETHERNET 1e3[s]	STD de SCI 1e3[s]
4	4.988	4.983	0.0016	0.0018
8	3.453	3.442	0.0059	0.0011
12	3.460	3.454	0.0016	0.0060
16	3.469	3.459	0.0021	0.0027
20	3.477	3.462	0.0046	0.0008
24	3.482	3.471	0.0072	0.0013
28	3.488	3.477	0.0070	0.0009
32	3.498	3.486	0.0010	0.0017
36	3.509	3.492	0.0049	0.0003
40	3.828	3.503	0.0020	0.0037
44	4.161	4.129	0.0081	0.0009
48	4.165	4.156	0.0029	0.0030
56	5.471	4.777	0.0051	0.0090
60	5.480	5.471	0.0017	0.0025
64	5.494	5.478	0.0015	0.0040
72	5.496	5.479	0.0034	0.0013
76	7.198	6.942	0.0016	0.0020
80	7.178	8.809	0.0022	0.0030

Tabla E.3: Impulso ethernet VS SCI con desviación estándar para Cuaco

TRABAJADORES	Onomatopoeiia [s]	Cuaco [s]
4	412.655	240.726
8	258.847	238.775
12	321.932	254.989
16	354.989	309.655
20	488.241	363.304
24	504.036	464.315
28	521.882	630.423
32	862.145	618.192
36	926.093	671.504
40	1024.486	655.958

Tabla E.4: Huygens comparación de clusters en tiempo de proceso

TRABAJADORES	Onomatopoeiia[s]	Cuaco [s]
4	6571.938	4988.756
8	3305.623	3453.834
12	3308.524	3460.023
16	3313.493	3469.067
20	3323.459	3477.698
24	3327.084	3482.534
28	3336.833	3488.712
32	3339.788	3498.945
36	3344.198	3509.378
40	3353.202	3828.798
44	3359.604	4161.408
48	3364.921	4165.155
56	3937.020	5471.834
60	3956.657	5480.921
64	3980.046	5494.818

Tabla E.5: Respuesta al impulso comparación clusters en tiempo de proceso

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	707.555	1	1	
2	375.933	1.882	0.941	0.0626
3	248.688	2.845	0.948	0.0272
4	218.585	3.236	0.809	0.0785

Tabla E.6: Huygens para ethernet en Cuaco

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	703.860	1	1	
2	373.064	1.886	0.943	0.0600
3	247.855	2.839	0.946	0.0282
4	197.523	3.563	0.890	0.0598

Tabla E.7: Huygens para SCI en Cuaco

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	14252.410	1	1	
2	7300.325	1.952	0.976	0.0241
3	4839.6603	2.944	0.981	0.0093
4	3596.8145	3.962	0.990	0.0031

Tabla E.8: Respuesta al Impulso para ethernet en Cuaco

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	13297.823	1	1	
2	6699.502	1.985	0.991	0.0103
3	4465.324	2.971	0.992	0.0050
4	3353.832	3.964	0.991	0.0030

Tabla E.9: Respuesta al Impulso para SCI en Cuaco

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	13238.972	1	1	
2	6665.518	1.986	0.993	0.0069
3	4450.710	2.974	0.991	0.0042
4	3336.860	3.967	0.991	0.0027
5	2981.449	4.440	0.888	0.0315
6	2333.934	5.672	0.945	0.0115
7	1915.716	6.910	0.987	0.0021
8	1731.473	7.646	0.955	0.0066

Tabla E.10: Respuesta al Impulso en Onomatopoeiia

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	933.229	1	1	
2	506.698	1.841	0.920	0.0859
3	342.175	2.727	0.909	0.0499
4	286.258	3.260	0.815	0.0756
5	273.697	3.409	0.852	0.1166
6	251.656	3.708	0.618	0.1235
7	218.004	4.286	0.611	0.1058
8	206.884	4.510	0.563	0.1104

Tabla E.11: Huygens en Onomatopoeii

PROCESADORES	TIEMPO [s]	SPEED-UP	EFICIENCIA	FRACCIÓN SERIAL
1	13037.843	1	1	
2	6569.860	1.984	0.992	0.0078
3	4387.876	2.971	0.990	0.0048
4	3297.041	3.954	0.988	0.0038

Tabla E.12: Respuesta al impulso en Onomatopoeiia para 4 procesadores

Bibliografía

[1] B. P. Lester, *The Art of Parallel Programming.* 1993.

[2] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE TRANSACTION ON COMPUTERS*, vol. C-21, Septiembre 1972.

[3] V. Kumar, *Introduction to Parallel Computing.* 1994.

[4] *Parallel Programming with MPI.* 1997.

[5] H. Dietz, "Linux parallel processing howto," vol. 980105, pp. 4–71, Enero 1998.

[6] A. S. Tanenbaum, *Sistemas Operativos Distribuidos.* 1996.

[7] A. H. Karp and H. P., "Measuring parallel processor performance," *Communication of the ACM*, vol. 33, pp. 539–542, Mayo 1990.

[8] G. Kinno, *Acoustic waves: devices, imaging and analog signal processing.* 1987.

[9] L. Beranek, *Acoustic.* 1954.

[10] P. Stepanishen and G. Fisher, "J. acoust. soc.," no. 69, 1981.

[11] P. Morse and K. Ingard, *Theoretical acoustics.* 1968.

[12] D. Guyomar and J. Powers, "J. acoust. soc.," vol. 77, 1985.

[13] L. Medina, E. Moreno, and L. González, G.and Leija, "Circular ultrasonic transucer characterization theoretical and experimental results," *Revista mexicana de física*, vol. 6, pp. 511–518, Diciembre 2003.

[14] F. Oberhettinger, "J. res. natl. bur. stand.," vol. B65, 1961.

[15] F. Rodríguez, A. R. Fernández, J. S. E. Prieto, and E. R. F. de Sarabia, "An. fís.," vol. 89, p. 25, 1993.

[16] A. Timoreva and F. S., *Curso de física General.* 1981.

[17] M. Lutz, *Programming Python.* 1996.

[18] F. Lundh, *Python Standar Library.* 2001.