BENEMERITA UNIVERSIDAD AUTONOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACION

“DISEÑO E IMPLEMENTACION DE UN SISTEMA DE PROCESAMIENTO MIMD CON MEMORIA COMPARTIDA”

TESIS

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

LILIANA ELENA OLGUIN GIL

ASESOR:

M.C. GRACIANO CRUZ ALMANZA

PUEBLA, PUE., FEBRERO 2004

Agradecimientos

A Dios por haberme dado la fortaleza y la capacidad para la realización y conclusión de este importante proyecto para mi vida profesional.

A mis padres Ma. Elena y Armando, que siempre han estado a mi lado apoyándome en cada nuevo reto emprendido.

A Irene, Francisco y Ramón, por su apoyo moral en todo momento, los quiero mucho, gracias por su amistad incondicional.

A la Dra. Lourdes Sandoval Solís y al M.C. Graciano Cruz Almanza por su inmenso apoyo para la realización de esta tesis, sin Uds. No hubiera sido posible este proyecto.

A las autoridades del Instituto Tecnológico de Tehuacán por las facilidades brindadas para la realización de este proyecto.

A COSNET, por el apoyo económico brindado para la realización de esta tesis.

A todos los que directa o indirectamente contribuyeron para la realización y culminación de este proyecto.

A ti, que estas siempre en mi corazón por tu paciencia y comprensión en todo momento.

Indice

Introducción

Capítulo 1. Supercómputo **1**

1.1 Introducción 1
1.2 Supercomputadoras 3
1.3 Cluster 7

Capítulo 2. El Modelo Linda **12**

2.1 Introducción 12
2.2 Modelo Linda 13
2.3 Espacio de tuplas 14
2.4 Sistema Linda 17
2.5 Instrucciones 18
2.6 Preprocesasdor 19

Capitulo 3. Diseño Linda - BUAP **20**

3.1 Introducción 20
3.2 Descripción General 21
3.2.1 Comunicación de datos y código entre Trabajador y Maestro 22
3.2.2 Comunicación de datos entre Trabajadores y aplicación del usuario 23
3.2.3 Comunicación de datos y código entre Desarrollador y Maestro 24
3.3 Unidad de procesamiento (worker) 25
3.3.1 Función que recibe código ejecutable del master 27
3.4 Biblioteca de funciones de coordinación 28
3.4.1 Descripción del diagrama de la funciòn Init_linda. 28
3.4.2 Descripción del diagrama de la funciòn End_linda. 29
3.4.3 Descripción del diagrama de la funciòn Out. 29

3.4.4 Descripción del diagrama de las funciones In y Rd. 31
3.4.5 Descripción del diagrama de las funciones Inp y Rdp. 34
3.4.6 Descripción del diagrama de la función Eval 35
3.4.7 Descripción del diagrama de las funciones Init_app y fin_app 36

Capitulo 4. Implementación de Linda - BUAP 38

4.1 Introducción 38
4.2 Estructura de datos y tipos 39
4.3 Diseño Final 42
4.3.1 Unidad de procesamiento (worker) 42
4.3.2 Función recv_file 48
4.3.3 Implementación de las funciones de coordinación 49
4.4 Pruebas realizadas sobre el sistema Linda – BUAP 57
4.4.1 Proyecto Nº 1: Hola mundo 58
4.4.2 Proyecto Nº 2: Sumatoria de los n primeros números naturales 59
4.4.3 Proyecto Nº 3: Suma de vectores 61
4.4.4 Proyecto Nº 4: Suma de matrices 62
4.4.5 Proyecto Nº 5: Multiplicación de matrices 63
4.4.6 Proyecto Nº 6: Inversa de una matriz 66
4.4.7 Proyecto Nº 7: Integral de una función 70
4.4.8 Proyecto No 8: Multiplicación de matrices por el algoritmo de Canon 72
4.4.9 Proyecto No 9: Multiplicación de matrices por el algoritmo de Fox 74

Conclusiones 76

Bibliografía 80

Anexo A 83

Introducción

La investigación y el desarrollo de aplicaciones en áreas donde se requieren grandes cantidades de cálculos aritméticos y lógicos ha incrementado la demanda de velocidad de procesamiento en las computadoras. Esto nos ha llevado a pensar en máquinas que permitieran obtener resultados en un menor tiempo, todo esto dio lugar al *procesamiento paralelo.*

El procesamiento paralelo se encarga de ejecutar una aplicación, dividiéndola en subtareas, ejecutando cada una de estas subtareas en dos o más procesadores en forma concurrente y en donde estos procesadores resuelven esta tarea completamente en forma cooperativa.

Las ventajas principales del procesamiento paralelo es la reducción significativa en tiempo de procesamiento para obtener resultados, tal es el caso del procesamiento de imágenes, investigaciones sobre genética, etc.

En la actualidad, este procesamiento se realiza en supercomputadoras de alto costo, y por lo tanto inaccesibles para ciertos sectores como el educativo.

Analizando lo anterior, se observó la necesidad de implementar una computadora paralela, basada en un clusters de computadoras personales, con una arquitectura MIMD (Múltiples Instrucciones, Múltiples Datos), con una memoria virtual compartida y un software de comunicación y control que permita desarrollar aplicaciones paralelas a un bajo costo.

Si bien es cierto que el costo de las computadoras PC's ha bajado significativamente, también es cierto que en las instituciones educativas se desecha equipo considerado obsoleto, pensando en esto se propone reutilizar este equipo para formar un cluster y además desarrollar un software que permita ejecutar programas sobre esta máquina utilizando software libre como los es el sistema operativo Linux, reduciendo con esto significativamente el costo de la implementación del sistema.

En el diseño y la implementación de este sistema, se utiliza el lenguaje ANSI C, que se proporciona con todas las distribuciones de Linux y el uso de los Sockets como herramienta de comunicación, ya que este sistema esta pensado en un modelo Cliente – Servidor.

El producto obtenido en este trabajo es el sistema Linda-BUAP V.1.0, del cual se ha implementado una aplicación llamada worker (trabajador), que es un sistema que simula a una unidad de procesamiento y es donde se ejecutan las tareas que son distribuidas por los programa de usuario. Este trabajador se conecta a una memoria virtual llamada master, en donde se encontrarán los datos necesarios para la ejecución de las aplicaciones.

Linda – BUAP, está basado en el modelo de los lenguajes de coordinación, los cuales se fundamentan en la separación de los aspectos de computación y de la interacción de los componentes que integran un sistema. En Linda –BUAP, el modelo de computación está representado por el lenguaje C y el modelo de coordinación consta de seis instrucciones que permiten sincronización y comunicación de datos y procesos.

Este sistema, Linda BUAP, está enfocado a la solución de problemas que requieran una gran cantidad de cálculos y poca acceso a la memoria.

El sistema Linda – BUAP, presenta aportaciones importantes tales como un monitoreo que permite al usuario determinar que esta sucediendo durante la ejecución de su aplicación, la reasignación de trabajo a los workers, esto es que se encuentren siempre listos para poder recibir trabajo, si aún existieran tareas pendientes, el no utilizar preprocesadores para traducir instrucciones Linda – BUAP a código C, la finalización normal de los trabajadores, aún cuando la memoria (master) salga de ejecución y el uso de funciones de biblioteca de C en las aplicaciones que se ejecutan en los workers.

En el Capítulo I de este documento se presenta una semblanza del Supercómputo en el mundo y en México, que son los sistemas medulares en esta tesis y una descripción de los Clusters de computadoras.

En el Capítulo II se presenta el Modelo Linda, del cual se ha basado el diseño de Linda – BUAP, presenta lo que es un espacio de tuplas, las instrucciones en Linda y el preprocesador utilizando en Linda.

En el Capítulo III se presenta de forma detallada el diseño del sistema Linda – BUAP con una amplia explicación de los algoritmos que se plantearon para la implementación del sistema.

El Capítulo IV se presenta la implementación detallada del sistema Linda – BUAP, una serie de ejemplos que comprueban el funcionamiento eficiente del sistema.

Capítulo 1

Supercómputo

1.1 Introducción

La tecnología de cómputo hoy en día se utiliza para una gran variedad de aplicaciones. Para las necesidades de la mayoría de los usuarios de cómputo, estas aplicaciones se limitan a procesadores de palabras, hojas de cálculo y multimedia, estas necesidades pueden ser cubiertas por una computadora personal, e incluso el porcentaje de utilización del procesador, generalmente no excede del 15% de su capacidad total. Por otra parte, existen aplicaciones que requieren de realizar millones de operaciones por segundo. Para satisfacer este tipo de necesidades, una computadora de escritorio no es suficiente, y se deben de utilizar computadoras que ofrezcan una capacidad de cómputo mucho mayor para poder responder a estas necesidades. Las supercomputadoras son utilizadas en la investigación, para resolver problemas y realizar cálculos numéricos que resultarían imposibles sin el uso de este tipo de máquinas. La función principal de las supercomputadoras es la de realizar cálculo numérico intensivo. Este tipo de cálculo es principalmente realizado en física, astronomía, química, y biología para simular condiciones que son imposibles de simular en un laboratorio. También son ampliamente utilizadas en la industria, especialmente en empresas internacionales que requieren manejar volúmenes gigantescos de información sobre sus clientes, lo cual resultaría imposible utilizando computadoras de escritorio.

En la década de los 60´s se utilizaban supercomputadoras para calcular la trayectoria de un misil, o el lanzamiento de un cohete al espacio. Hoy en día las supercomputadoras se utilizan para realizar simulaciones de colisiones de estrellas, simulaciones climatológicas, simulaciones nucleares, e incluso han sido una herramienta esencial para el desciframiento del genoma humano, así como en la industria del entretenimiento.

El supercómputo es la tecnología más avanzada de cálculo numérico que existe actualmente para desarrollar investigaciones complejas de alto nivel de especialización; es una herramienta que le permite al investigador llevar a cabo con certeza y velocidad, billones de cálculos matemáticos para estudiar problemas de gran magnitud; su altísima capacidad para procesar simultáneamente grandes volúmenes de información facilita el estudio de fenómenos y condiciones que tan solo hace menos de 30 años era imposible; sus aplicaciones abrieron en todo el mundo, nuevas líneas de investigación científica.

El supercómputo es un término genérico que engloba equipos para cálculo numérico a grandes velocidades, compiladores altamente desarrollados,

bibliotecas numéricas, herramientas auxiliares para optimizar códigos y sistemas para balance y calendarización de los trabajos enviados para su procesamiento. La finalidad de esta tecnología es servir como una herramienta de propósito general para apoyar la investigación científica básica y aplicada.

En México el supercómputo tiene ya su historia [1], en 1991 se instaló en la Universidad Autónoma de México la primera supercomputadora de América Latina, la Cray Y-MP[2], que operó de 1991 a 2001(*ver fig.1.1*), actualmente en esta institución cuentan con supercomputadoras tales como una SGI Cray Origin 2000 con 40 procesadores que inició operaciones en 1997 y una HP Alpha modelo SC-45 que inició operaciones en marzo de 2003, esta última es una máquina con 32 procesadores, 15 veces más rápida que la Origin y 60 veces mas rápida que la Cray.

Fig. 1.1
Cray Y-MP fue la primer supercomputadora instalada en la UNAM y en Latinoamérica.

Aun y con todos los beneficios que se pudieran mencionar del supercómputo no se puede pasar por alto mencionar el alto costo que implica adquirir una supercomputadora como las anteriormente descritas, solo para darse una idea mencionaremos que en 1991 el costo del equipo adquirido por la UNAM fue de Ocho Millones de Dólares, el que adquirieron en 1997 costó Tres millones y el adquirido en 2003 costó Quinientos Mil Dólares en números redondos, como se puede observar es verdaderamente difícil por no mencionar imposible que instituciones o empresas cuenten con sendos presupuestos para adquirir equipos como estos.

1.2 Supercomputadoras

Las supercomputadoras son máquinas de alto rendimiento muy poderosas que se usan sobre todo para cálculos científicos. Para acelerar la operación, los componentes se juntan para minimizar la distancia que tienen que recorrer las señales electrónicas. Las supercomputadoras también utilizan técnicas especiales para evitar el calor en los circuitos y prevenir que se quemen debido a su proximidad (ver fig. 1.2).

Fig. 1.2a
Origin 2000 instalada en 1997 por SGI.

Fig. 1.2b
Alpha Server SC 45 es la más reciente adquisición de la UNAM.

Una supercomputadora es un sistema computacional que se reconoce por su alta velocidad de cálculo, sus sistemas de memoria grandes y rápidos y un uso amplio de procesamiento paralelo. Está equipada con unidades funcionales múltiples y cada unidad tiene su propia configuración de arquitectura paralela. Aunque la supercomputadora maneja aplicaciones de propósito general que se encuentran en todas las otras computadoras, está optimizada específicamente para el tipo de cálculos numéricos que involucran vectores y matrices de números de punto flotante.

Las supercomputadoras no son convenientes para procesamiento cotidiano normal de una instalación de computadora típica.

La velocidad de una supercomputadora se mide en base a la cantidad de operaciones de punto flotante que hace por segundo. El término técnico para esta velocidad es FLOPS. Si se realiza una operación en la calculadora en un segundo, diríamos que la velocidad sería de 1 FLOPS.[3].

Lo que hace que una supercomputadora sea "SUPER" es su capacidad para ejecutar al menos mil millones de operaciones de punto flotante por segundo. Es una medida de velocidad sorprendente conocida como "*gigaflop*", actualmente las computadoras más potentes son capaces de realizar varios billones de operaciones por segundo, es decir, *“teraflop”.*

El primer gran éxito en el diseño de un supercomputador moderno fue el Cray-1, el cual se presentó en 1976. Una de las razones por las que el Cray-1 tuvo un gran éxito fue porque podía realizar más de cien millones de operaciones aritméticas por segundo (100 Mflops). Fue diseñado por Seymour Cray (Cray Research Inc.), actualmente las supercomputadoras son diseñadas y producidas por compañías tradicionales como IBM o HP[4].

Hoy en día el diseño de supercomputadoras se sustenta en 3 importantes tecnologías:

- *Las de registros vectoriales***,** las cuales contienen en su interior muchos elementos de un vector al mismo tiempo, lo que les permite la ejecución de innumerables operaciones aritméticas en paralelo. Los procesadores vectoriales son únicos, Cray tiene el suyo, Nec tiene el suyo, Fujitsu tenía el suyo, son procesadores físicamente mas grandes que los ordinarios, que tienen por ejemplo 64, 128 o 256 circuitos en paralelo de procesamiento. El procesamiento vectorial requiere específicamente instrucciones con vectores en lenguaje de máquina. Una instrucción vectorial indica la operación que se va a realizar y especifica la lista de operandos (denominada *vector*) sobre los que operará.

- *El sistema conocido como M.P.P (Massively Parallel Processor) o Procesadores Masivamente Paralelos*, que consiste en la utilización de cientos y a veces miles de microprocesadores estrechamente coordinados que operan simultáneamente sobre una tearea específica. Por ejemplo, la CDC 6600 y algunas de sus sucesoras tienen unidades funcionales múltiples (ALU especializadas), cada una de las cuales puede realizar una operación sencilla a alta velocidad. En este ejemplo, se tienen cinco unidades funcionales, dos de ellas para la suma y las tres restantes para las operaciones de resta, multiplicación y división.

  La idea que sustenta este diseño es que la unidad de control extrae un instrucción y la dirige a una de las unidades funcionales para su ejecución; mientras tanto, la unidad de control extrae la siguiente instrucción y la envía a otra unidad funcional. Este proceso continúa hasta que no se puede avanzar más, ya sea porque todas las unidades del tipo requerido están ocupadas, o bien porque se necesita un operando que todavía se está procesado.

- *Clusters*, los cuales consisten en interconectar varias computadoras de uso general (PC), por medio de una red local de baja latencia y un gran ancho de banda, elaborar programas para hacerlos trabajar en paralelo. El resultado ha sido una solución que rivaliza con las mejores supercomputadoras a un precio muy inferior. La idea viene desde la década de los 50's, sin embargo, han sido los factores técnico – económicos que hicieron que hasta los 90's fuera viable la idea por el bajo costo de las PC's,

la facilidad para conectar en red las computadoras y la aparición de sistemas operativos abiertos como Linux.

El primer cluster fue creado en el Centro Goddard de la NASA por los investigadores Becker y Sterling, quienes conectaron entre si 16 PC's con procesadores Intel 486 a través de una red Ethernet y utilizando Linux, alcanzando los 70 megaflops de velocidad.

Las supercomputadoras se conocen tanto por sus aplicaciones como por su velocidad o capacidad de computación, que puede ser 10 veces mayor que la de una macrocomputadora. Las siguientes aplicaciones son representativas de las supercomputadoras:

- Las supercomputadoras ordenan y analizan grandes cantidades de datos sísmicos que se recopilan durante las exploraciones en busca de yacimientos petrolíferos.
- Las supercomputadoras permiten la simulación de una corriente de aire alrededor de un avión a diferentes velocidades y altitudes.
- Los fabricantes de automóviles usan supercomputadoras para simular accidentes automovilísticos en pantallas de video. (Es menos costoso, permite conocer más detalles y es más seguro que desarrollar un choque real).
- Los físicos usan supercomputadoras para estudiar los resultados de explosiones de armas nucleares.
- Los meteorólogos usan supercomputadoras para estudiar la formación de tornados.
- Los estudios de producción de Hollywood usan gráficos avanzados para crear efectos especiales para películas y comerciales de televisión.
- Analizar el comportamiento de fluidos, diseñar piezas aerodinámicas

Sin la rapidez y la capacidad de cálculo de las supercomputadoras, algunas disciplinas se habrían "ahogado" en sus planteamientos teóricos. Tal es el caso de la física de alta energía. Hay experimentos en el CERN (Centro Europeo para la Investigación Nuclear) que hacen colisionar electrones y positrones y que producen tal cantidad de información que, sin la ayuda de una supercomputadora que sepa discriminar entre todos los sucesos, no se habría podido comprobar experimentalmente las ideas teóricas.[6]

En la investigación espacial, la utilización de computadoras se convirtió en esencial. La nave Voyager 2, que fue lanzada el 20 de agosto de 1977 con la misión de explorar los planetas exteriores al sistema solar, iba equipada con seis computadoras diferentes, con capacidad de 540 Megas, algo portentoso para la época.

Hoy en día, la existencia de supercomputadoras que, naturalmente, trabajen en tiempo real, se ha convertido en una necesidad.

Por ejemplo, son imprescindibles en las industrias del automóvil y la aeronáutica. En este caso los estudios de aerodinámica son una pieza fundamental para optimizar la forma del fuselaje o de las alas. También se emplea en simulación de vuelos para el entrenamiento de los pilotos, etc. El análisis de la estructura del avión Boeing 777 se realizó completamente por una supercomputadora y también el diseño del avión invisible F-117.

Otras aplicaciones son el diseño de nuevos productos farmacéuticos, componentes electrónicos, simulación de terremotos, estudio de la evolución de la contaminación en áreas extensas, predicción meteorológica y estudios del cambio climático o simulación de órganos corporales con el objetivo de reproducir su funcionamiento con representaciones en 3D de alta precisión a partir de métodos de resonancia magnética.

El tiempo transcurrido desde la concepción de un nuevo producto hasta su introducción en el mercado y el costo de su proceso de diseño pueden reducirse drásticamente con la ayuda de la simulación por computadora. Todo esto, combinado con la exactitud alcanzada en reproducir la realidad, tiene el efecto de aumentar substancialmente la competitividad de la industria al reducir costos y mejorar la calidad. Al mismo tiempo hay áreas donde las supercomputadoras deben utilizarse para establecer los métodos y modelos de simulación física más eficientes.

Otro ejemplo típico es el diseño de nuevas moléculas o materiales y es básico para las industrias química y farmacéutica. Combinado con desarrollos en biotecnología, se pueden diseñar y producir nuevas proteínas, obteniéndose nuevos productos que tendrán impacto en la vida cotidiana. Las estrategias tradicionales para el diseño de nuevos componentes involucran métodos inteligentes de prueba y error. Los modelos computacionales están tan solo comenzando a hacer impacto en el proceso de diseño. Sin embargo, este diseño aún consume mucho tiempo y dinero. Un aumento considerable en potencia computacional es necesario para acelerar el diseño de nuevos productos, lo que permitirá un mayor nivel de competitividad en nuestra industria química.

En México se hace cómputo numérico de alto rendimiento en instituciones tales como la UNAM, la UAM , el CINVESTAV, el Instituto Mexicano del Petroleo, PEMEX, el CICESE en Ensenada y la Facultadad de Ciencias de la Universidad Autónoma del Estado de Morelos, actualmente la Benemérita Universidad Autónoma de Puebla acaba de adquirir un CLUSTER con 32 Procesadores Opteron .

El mercado de las supercomputadoras ha crecido de manera que cada vez son más los fabricantes de computadoras que se dedican al diseño de supercomputadoras, algunos de ellos son:

- CRAY INCORPORATION
- IBM
- INTEL
- HITACHI
- COMPAQ
- FUJITSU
- SUN
- HEWLETT PACKARD

1.3 Clusters

Ya se ha comentado la necesidad de “supercómputo” que existe en muchas ramas de la ciencia y en el ámbito comercial y del alto costo que implica adquirir una supercomputadora como la Cray. En los últimos años, el personal académico de diversas universidades y centros de investigación se ha dado a la tarea de construir sus propias supercomputadoras conectando computadoras personales y desarrollando software para enfrentar problemas complejos y abatir costos.

Una alternativa importante para atender soluciones a ciertas demandas de cómputo científico se deriva del hardware de mercado masivo y de componentes de software libre, como el sistema operativo Linux[5], el cual por ser de acceso gratuito y con gran robustez permite que los clusters sean muy eficientes

"*Un cluster es una clase de arquitectura de computador paralelo que se basa en unir máquinas independientes integradas por medio de redes de interconexión, para proveer un sistema coordinado, capaz de procesar una carga" (Dr. Thomas Sterling).*

El nacimiento de los clústers se remonta a los años 80’s pero fue hasta los años 90’s cuando empezaron a cobrar fuerza.

En el verano de 1994 Thomas Sterling y Don Becker, trabajando en CESDIS (Center of Excellence in Space Data and Information Sciencies) bajo el patrocinio del Proyecto de la Tierra y Ciencias del Espacio (ESS), construyeron un Cluster de Computadoras, que consistía en 16 procesadores DX4, conectadas por un canal Ethernet a 10 Mbps. Ellos llamaron a su máquina *Beowulf.*[6]

Esta máquina nació de la idea del reciclaje, al ver que las viejas computadoras eran arriconadas para dar paso a las maquinas modernas, así pues tomaron maquinas de diferentes tipos y velocidades de procesador, para el año 2001 la máquina Beowulf [7] constaba de 133 nodos: 75 procesadores Intel, 53 Intel – Pentium y 5 Alphas.

Los clusters surgieron para dar alternativas de economía, eficacia y confiabilidad, la creación de los clusters ofrece a los investigadores otras opciones de trabajo

para la obtención de resultados óptimos en tiempos razonables y representa menores costos de adquisición, operación y mantenimiento.

Un servicio basado en clusters es tan confiable como un equipo comercial. Es importante comentar que una parte importante de los servicios de supercómputo en los próximos años, será sustentada por súper clusters para cómputo paralelo escalable o masivo.

Por otro lado la evolución y la estabilidad que ha alcanzado el sistema operativo Linux ha contribuido importantemente al desarrollo de muchas tecnologías nuevas entre ellas la de los clusters. En el mundo de GNU/Linux hay ya software para clusters tal como Mosix [8](permite realizar balance de carga para procesos particulares en clusters) y su contraparte OpenMosix[9], estos sistemas proveen migración automática de procesos en clusters homogéneos de máquinas GNU/Linux.

Otra tecnología de clusters es Piranha[10], que permite a servidores Linux proveer alta disponibilidad sin la necesidad de invertir cantidades mayores en hardware, puede configurar un servidor de respaldo en caso de fallo de la contraparte, también puede hacer que el cluster aparezca como un servidor virtual.

Existen varias clasificaciones de los tipos de clusters, así en función de su disponibilidad y desempeño tendríamos:

- Un cluster de alta disponibilidad (High Availability) consisten en varias máquinas que comparten los discos duros y se monitorizan entre sí constantemente. Su utilidad deriva de la necesidad de tener un sistema capaz de reponerse de un fallo de hardware y de la necesidad de abaratar los costos de los sistemas redundantes y tolerantes a fallos. Cuando uno de los equipos cae, los demás migran sus procesos hacia ellos mismos para que no haya pérdidas de datos y de disponibilidad del conjunto. A su vez, se encargan de reiniciar a la computadora caída y, tras lograrlo, le devuelven su carga correspondiente para volver a la normalidad.
  La aplicación más común para este tipo de diseño son servidores de páginas web o de bases de datos tolerantes a fallos, que necesiten estar operativos las 24 horas del día todos los días del año.

- Los clusters de alto rendimiento, se basan en un conjunto de máquinas diseñadas para lograr una capacidad de cálculo máxima al repartirse la carga de los procesos entre los nodos de acuerdo a las reglas escogidas. Con esto se consigue mejorar el rendimiento en la obtención de la solución de un problema. Normalmente se utilizan en la resolución de algoritmos científicos, en las granjas de compilación (conjunto de computadoras destinadas a compilar una misma cosa entre todos), granjas de procesamiento de imágenes (grupo de máquinas que se dedican a la visualización, modificación y reproducción de imágenes en 3D) y en la compresión o cifrado de códigos.

- Los clusters de alta confiabilidad, que son una mezcla de los de alta disponibilidad y los de alto rendimiento, tienen como objetivo evitar que las aplicaciones se caigan. No existe otra manera de conservar el estado de las aplicaciones que mediante la inclusión de puntos de parada (checkpoints), pero en conexiones de tiempo real no son suficientes, ya que no se trata únicamente de utilizar el último checkpoint del sistema y relanzar el servicio. Estos clusters necesitan tener un único reloj de sistema conjunto. No pueden ser implementados de manera eficiente, en tiempo real, únicamente mediante software, debido a problemas de latencia de red. Este tipo de clusters no son fáciles de implementar en la actualidad.

Todos los clusters mencionados anteriormente hacen uso de un sistema de balanceo de carga que reparte los procesos entre todos los nodos del cluster de forma que no haya ningún nodo saturado.

Si hablamos de la transparencia de los clusters, nos topamos con 2 tipos:

Por una parte, los clusters no transparentes requieren una programación paralela explícita y el conocimiento de la topología de la red del cluster por adelantado(Beowulf) y/o la utilización de librerías para el paso de mensajes entre tareas, que pueden estar (PVM MPI[11]), dentro de los clusters no transparentes destaca el proyecto Beowulf.

Por otra parte, en los clusters transparentes el programador no necesita saber la topología de la red, ni necesita utilizar técnicas de programación paralela explícita, ya que ni siquiera necesita saber que su programa va a funcionar en un cluster. Simplemente con adoptar la metodología de programación en multiprocesador es suficiente.

Con respecto al software con el que se cuenta para la administración del cluster tendríamos:

- *Clusters escasamente acoplados*: si no se posee con un sistema de instalación y gestión integrado que posibilite una recuperación rápida antes fallos y una administración centralizada que ahorre tiempo al administrador.
- *Clusters medianamente acoplados*: representa una mejora al modelo anterior, incorporando un sistema centralizado de instalación y administración del cluster.
- *Clusters altamente acoplados*: estos sistemas tiene como objetivo eliminar las unidades de disco de los nodos y así utilizar unicamente el procesador y la memoria. Estos clusters estarian clasificados dentro de los de balance de carga.

Entre las aplicaciones más comunes de los clusters se encuentra cálculo numérico, astronomía, procesamiento de imágenes, investigación de criptografía, etc.

Un ejemplo muy palpable de lo que es la eficiencia de los clusters de alta disponibilidad es el cluster basado en Linux de el motor de búsqueda GOOGLE [12]. Este buscador debe atender, en promedio, mas de 200 millones de consultas por día, debe almacenar la información indexada de mas de 3 mil millones de páginas web en 36 idiomas, etc. Google opera con un cluster de más de 10,000 nodos Linux.

En 1996, hubo también otros dos sucesores del proyecto Beowulf de la NASA. Uno de ellos es el proyecto *Hyglac* desarrollado por investigadores del Instituto Tecnológico de California (CalTech) y el Laboratorio de Propulsión Jet (JPL), y el otro, el proyecto *Loki* construido en el Laboratorio Nacional de Los Alamos, en Nuevo México. Cada cluster se integró con 16 microprocesadores Intel Pentium Pro y tuvieron un rendimiento sostenido de más de un gigaflop con un costo menor a $50,000 dólares.

En 1996, en el Laboratorio Nacional de Oak Ridge en Tennessee, se enfrentaban al problema de elaboración de un mapa de las condiciones ambientales del territorio de Estados Unidos. El territorio fue dividido en 7.8 millones de celdas de 1Km. Cada celda contenía la información de 25 variables, desde la precipitación promedio mensual hasta el contenido de Nitrógeno del suelo. Ninguna estación de trabajo o PC podría con esta tarea. Se requería una supercomputadora de procesamiento paralelo. En la actualidad cuentan con un cluster de 130 PCs para trabajar en la elaboración del mapa de eco regiones.

Una de las grandes ventajas de los clusters es que pueden construirse prácticamente con cualquier tipo de procesador. Incluso se han construido clusters que en vez de utilizar computadoras personales, utilizan Play Stations. Este cluster utiliza Linux como su sistema operativo. A pesar de no ser uno de los clusters más poderosos del mundo, es un proyecto interesante, ya que utilizó un hardware que nunca se había utilizado para la construcción de un cluster.

En México, se utilizan clusters para cómputo numérico de alto rendimiento en el Instituto Mexicano del Petróleo (IMP), en PEMEX, en el ámbito educativo con fines de investigación se tienen proyectos en la Universidad de Sonora (Cluster Cientifico Alfa), en la Universidad Autónoma de Baja California en el Instituto de Investigaciones Oceanológicas (Cluster UABC), en la UNAM en el Instituto de Astronomia (Cluster EMONG ver Fig.1.3) [13], y actualmente la Universidad

Fig.1.3
Este cluster es el primero para cómputo de alto rendimiento en la UNAM.

Autónoma de Puebla en la Facultad de Ciencias de la Computación acaba de adquirir un cluster con 32 procesadores Opteron, el cual estará dedicado a la investigación.

Capítulo 2
El Modelo Linda

2.1 Introducción

En la actualidad se han desarrollado un sinfín de herramientas que permiten el desarrollo de programas paralelos. Esto ha surgido como una necesidad a la creciente demanda de velocidad de cómputo para el desarrollo de aplicaciones que demandan un alto costo en tiempo de procesamiento.

Existen bibliotecas basadas en el modelo de paso de mensajes tales como PVM y MPI [14], que han sido ampliamente aceptados para el desarrollo de programas paralelos. En este modelo los programadores son los encargados de administrar la carga de trabajo entre los procesadores del sistema, indicando en forma explícita en que procesador se ejecutará cada tarea.

Los lenguajes de coordinación [15] son una nueva clase de lenguajes de programación paralela, los cuales se fundamentan en la separación de los aspectos de computación y de interacción de los componentes que integran un sistema.

Bajo este esquema un lenguaje de coordinación se forma de un lenguaje secuencial (tal como Pascal o C) el cual se encarga de los aspectos de cómputo aritmético y lógico e instrucciones adicionales que se encargan de los aspectos de comunicación de datos y código entre los procesadores del sistema.

Linda [16] es históricamente el primer miembro genuino de la familia de los lenguajes de coordinación, fue desarrollado a mediados de los 80´s por David Gelernter en la Universidad de Yale.

En este capítulo se da la descripción del modelo Linda propuesto por Gelernter, en el cual se basa el desarrollo de LINDA-BUAP, un lenguaje de coordinación desarrollado con fines didácticos para la investigación, el cual da como resultado este trabajo de tesis.

2.2 Modelo Linda

El paradigma de coordinación ofrece una forma prometedora y sencilla de programar aplicaciones paralelas y distribuidas.

Un modelo de coordinación [17] se basa en separar los aspectos de cómputo de los aspectos de comunicación e intercambio de datos entre los procesos que se encuentran involucrados en la solución de un problema.

Un lenguaje de coordinación [17] esta formado por dos partes un modelo de computación y un modelo de coordinación. El modelo de computación esta dado por un lenguaje de programación secuencial tradicional, el cual se encarga de ejecutar los cálculos y operaciones que requiere el problema y el modelo de coordinación utiliza estructura de datos para la comunicación y sincronización de datos y procesos. La comunicación entre procesos se basa en operaciones de lectura y escritura dentro de la estructura de datos.

Linda es un lenguaje de coordinación [18], este lenguaje de coordinación permite desarrollar programas paralelos que se ejecutan sobre una gran variedad de arquitecturas de computadoras ya que el conjunto de operaciones de lectura y escritura puede ser integrado en diferentes lenguajes de programación de una forma simple y efectiva, facilitando claramente la separación entre los aspectos de cómputo y de coordinación.

Dado que el modelo Linda no separa automáticamente el código en subtareas, el programador deberá paralelizar sus aplicaciones por sí mismo. Esto se realiza insertando instrucciones Linda dentro del código fuente de la aplicación, las cuales se convierten en llamadas a funciones de biblioteca por medio de un preprocesador, mismas que controlan la distribución de las tareas a través de los diferentes procesadores en el sistema.

Linda y el uso de un espacio de tuplas proporcionan un nuevo paradigma de coordinación y comunicación llamado *Comunicación Generativa[19]*, el cual presenta dos características que lo hacen mas potente que el modelo de paso de mensajes, lo primero es que el que recibe un mensaje no conoce que proceso crea el mensaje y el que envía un mensaje no conoce quien recibirá el mensaje, esto significa además, que dos procesos se pueden comunicar aún y cuando no existan al mismo tiempo y sin saber con quien se están comunicando; la segunda característica consiste en que cualquier campo con un mensaje (tupla) puede ser utilizada para recuperar un mensaje.

En el paso de mensajes los datos no son compartidos, en algunas aplicaciones los procesos tienen que esperar a que otros procesos liberen los datos que ellos necesitan, y esto puede generan cuellos de botella o hasta abrazos mortales, aunado a esto el proceso que genera un mensaje debe conocer la identidad del receptor.

2.3 Espacio de Tuplas

El modelo Linda esta basado en una memoria asociativa compartida llamada Espacio de Tuplas[20]. Este es el medio de coordinación, es una estructura de datos compartida la cual contiene Tuplas (Registros).

Una *Tupla* es una secuencia finita y ordenada de campos, cada campo tiene un valor y un tipo soportado por el lenguaje host. Un ejemplo de una tupla en Linda seria:

(‘ejemplo’, 8, 12.6)

Esta tupla esta formada por 3 elementos:

‘ejemplo’	que es el nombre de la tupla
8	es un número entero
12.6	es un número flotante

Las tuplas pueden contener desde datos simples hasta estructuras de datos más complejas como arreglos o registros.

Las *tuplas* se clasifican en dos tipos: *tuplas activas* y *pasivas*. Las *tuplas pasivas* se utilizan para el intercambio de información bidireccional entre procesos a través del espacio de tuplas, por lo que representarán datos. Las *tuplas activas* son usadas para el envío de trabajo del desarrollador hacia el espacio de tuplas –trabajo que será adquirido por los procesadores para su ejecución– y por lo tanto representan procesos.

Dentro de una tupla podemos encontrar además:

- *Elementos actuales:* es un valor actual, puede ser una representación numérica o puede estar contenido en una variable. Cuando se utilice un número como elemento, simplemente se coloca el número explicito y cuando se utilicen variables se deberá preceder a las variables con el signo & para poder visualizar el contenido de las mismas, los valores contenidos en las variables se integrarán a la tupla en tiempo de ejecución, ejemplo:

  (‘ejemplo1’,5,6)
  (‘ejemplo2’,&a,&b)

- *Elementos formales:* son utilizados para recoger un valor desde una tupla y se distinguen de los actuales por no necesitar el & . Los valores de las variables que aparecen en la tupla serán modificados por valores retornados después de que se haya encontrado una tupla que coincida con el prototipo, ejemplo:

  (‘ejemplo3, a, b, c)

La diferencia entre tupla y prototipo reside en el hecho de que una tupla será usada para enviar datos (o código para su ejecución por parte de los procesadores) al espacio de tuplas, mientras que el prototipo se usará para requerir datos del espacio de tuplas.

Un espacio de tuplas es un tipo de memoria compartida, pero a diferencia de ella, los objetos en un espacio de tuplas no tienen direcciones. Cuando un objeto tiene necesidad de comunicarse, genera una tupla y la inserta en el espacio de tuplas, otros procesos pueden recuperar tuplas del espacio de tuplas, mediante una búsqueda. Los procesos emisores y receptores están desacoplados en este modelo de programación, es decir, no necesitan existir al mismo tiempo y unos no conocen la identidad de los otros; solamente se comunican a través de información compartida (ver Fig. 2.1).

Un espacio de tuplas tiene cuatro propiedades importantes:

1. Es compartido. Todos los procesos trabajan con el mismo espacio de tuplas y el espacio puede ser utilizado por cualquier proceso.
2. Es como una bolsa, no como un conjunto. El espacio puede contener muchas tuplas idénticas.
3. Es asociativo. Las tuplas son removidas del espacio usando reglas de comparación de patrones, más que el ser direccionadas.
4. Es anónimo. Una vez que una tupla ha sido colocada en el espacio, el sistema no mantiene la pista de quién la creó o cuando. Un programa puede incluir información en tuplas para mantener el seguimiento de donde viene o dónde serán usadas, pero este manejo es responsabilidad del programador.

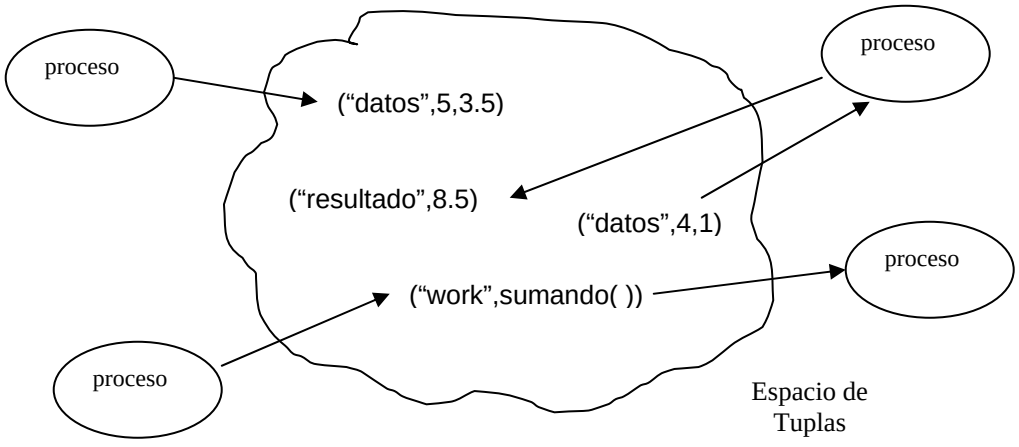


Fig. 2.1 Espacio de Tuplas

Cuando se desarrolla un programa utilizando un espacio de tuplas, se debe crear un proceso principal, el cual va a ser el encargado de repartir el trabajo en subprocesos, los cuales serán enviados al espacio de tuplas en forma de tuplas activas. Estas tuplas serán obtenidas por los procesadores y ejecutarán el código en ellas contenido, si necesitan algún dato para llevar a cabo su tarea, podrán solicitarlo a través de tuplas pasivas al espacio de tuplas y una vez finalizada su ejecución, depositarán resultados en el espacio de tuplas en forma de tuplas pasivas, para que el proceso principal las obtenga y las procese como mejor convenga al programador.

Las tuplas pueden ser modificadas únicamente después de ser recuperadas del espacio de tuplas, esto da como resultado un mecanismo implícito de sincronización.

Toda esta comunicación se lleva a cabo utilizando cuatro operaciones [21] sobre el espacio de tuplas. Los procesos individuales no se interesan por lo que los otros hacen o como lo hacen. Esta característica hace que para los programadores sea fácil escribir programas paralelos porque no tienen que preocuparse por detalles de bajo nivel como el destino que los mensajes y la sincronización explícita.

2.4 Sistema Linda

Es una combinación de hardware (cluster de PC's conectadas a través de una red LAN) y software (Lenguaje de Coordinación) que al estar integradas bajo el modelo Linda conforman una máquina paralela escalable, ya que se pueden incorporar tantas computadoras como sea necesario. Esta máquina paralela estará integrada por una Unidad de Control a la cual llamaremos Desarrollador, una unidad de Memoria a la cual llamaremos Maestro y Unidades Centrales de Procesamiento a las cuales llamaremos Trabajadores, Ver Fig. 2.2.

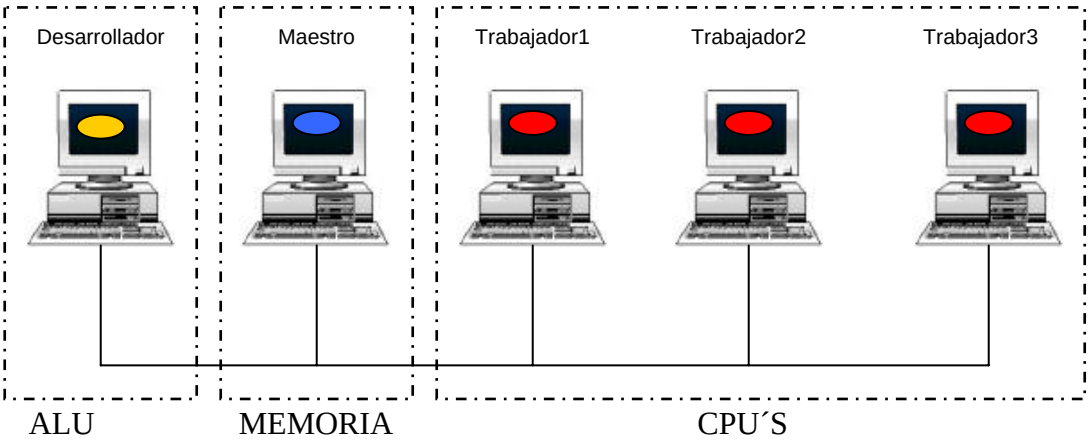


Fig. 2.2 Sistema Linda

Desarrollador:
En esta computadora se realizarán las aplicaciones. En ella deberá de estar instalado el entorno de programación Turbo Pascal y el preprocesador de Linda (Linda.exe). El equipo que tenga este papel tendrá la responsabilidad de repartir el trabajo mediante la creación de tuplas 'work' para los trabajadores, coordinar dichos procesos y finalmente recuperar los resultados producidos para su debida manipulación final.

Maestro:
Es el encargado de administrar el espacio de tuplas. Su función es la de mantener las tuplas que son usadas como un medio de comunicación entre los trabajadores y el desarrollador, así como recibir las peticiones por algunas de ellas y darles seguimiento hasta que son encontradas y devueltas. En caso de no encontrar ninguna igual, entonces tiene que mantener la pista de que procesador es el que hizo la petición y por lo tanto es el que se encuentra bloqueado.

Trabajador:
Está máquina se usa solamente para realizar cálculos. En ella se obtiene el trabajo a ejecutar mediante tuplas activas que empiecen con el nombre 'work'. El intercambio de información con el desarrollador se hace por medio de tuplas pasivas; en cualquiera de los dos casos, usa al maestro como puente de comunicación.

2.5 Instrucciones Linda

El lenguaje Linda cuenta con seis primitivas de coordinación que permiten el envío y recuperación de tuplas pasivas del espacio de tuplas. Dichas operaciones están representadas por las siguientes instrucciones:

Out(t). Coloca una tupla pasiva *t* en el espacio de tuplas y el proceso en ejecución continua inmediatamente, por ejemplo: out("datos",x,10), coloca la tupla llamada "datos" con el valor que contenga la variable x y el valor entero 10.

In(s). Busca una tupla *t* en el espacio de tuplas que coincida con el prototipo *s* , si existe la recupera del espacio de tuplas y la borra, los valores de los campos actuales en *t* son asignados a los campos formales en *s* y el proceso en ejecución continua inmediatamente, si no se encentra una tupla *t* disponible para el prototipo *s* el proceso se bloquea hasta que sea depositada la tupla *t* esperada. Si existen varias tuplas *t* que coinciden con el prototipo *s*, se escoge una arbitrariamente. Por ejemplo: in("datos",&x,y);

Rd(s). Funciona igual que **in** sólo que este unicamente toma una copia de *t* y no la elimina del espacio de tuplas. Al igual que **in** es una instrucción bloqueante.

Inp(s). Funciona igual que **in** sólo que esta instrucción es no bloqueante, es decir que si no existe una tupla *t* que coincida con el prototipo *s* el proceso no se bloquea y retorna un valor booleano.

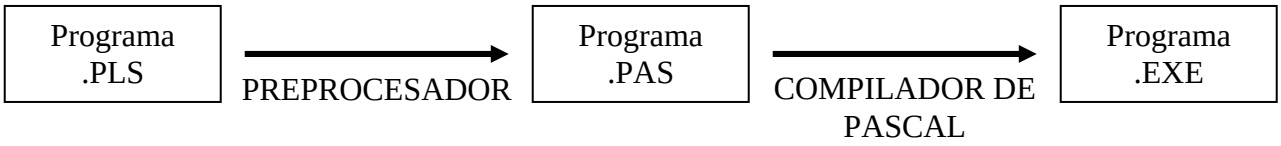
Rdp(s). Funciona igual que **rd** sólo que en una forma no bloqueante, es decir devuelve un valor booleano y no bloquea al proceso en caso de que no exista la tupla *t* que coincida con el prototipo *s*.

Eval. Coloca una tupla activa en el espacio de tuplas. Cada tupla colocada por esta instrucción es una función, para la cual se crea un nuevo proceso para evaluar la función. Una vez que todas las funciones han sido evaluadas, eval coloca la tupla resultado (tupla pasiva) en el espacio de tuplas.

2.6 Preprocesador.

El modelo Linda utiliza un preprocesador llamado Linda.exe, el cuál se encarga de traducir instrucciones linda a su correspondiente lenguaje huésped, para que en el momento de ser compilado el programa reconozca las instrucciones Linda.

Un programa desarrollado en Linda consta de las siguientes fases:



Primeramente se tiene que crear un programa con la extension PLS, el cual contendrá instrucciones Linda mezcladas con la sintaxis del lenguaje host (pascal para en este caso). Una vez creado este programa, se tiene uno que salir del compilador de pascal y ejecutar el programa Linda.exe que es el preprocesador que convertira las llamadas a las funciones in, out, rd, inp, rdp, eval a instrucciones válidas en el lenguaje, sustituye los argumentos no utilizados en la llamada a las instrucciones con espacios en blanco.

Un inconveniente presentado en este modelo es que hay que tener cuidado con los espacios en blanco que se dejan en el momento de la edición del programa PLS, ya que en caso de omitir o insertar un espacio, la instrucción no puede ser traducida y por lo tanto no podremos obtener el programa ejecutable final.

Una vez que se preprocesa el archivo PLS, se obtiene un archivo con la extensión .PAS, el cual al ser compilado con el compilador del lenguaje (Turbo Pascal), todas las instrucciones ahí incluidas son ya reconocidas. En caso de no tener ningun error se genera un archivo ejecutable listo para correr en el desarrollador.

Para este modelo el uso del preprocesador significa una herramienta importante para poder incluir instrucciones Linda dentro de los programas desarrollados en lenguajes huesped tales como Pascal.

Capítulo 3
Diseño de Linda-BUAP

3.1 Introducción

La investigación en el ramo del paralelismo a dado como resultado un gran número de proyectos, cuyo objetivo primordial es crear sistemas de procesamiento paralelo eficientes que permita desarrollar aplicaciones paralelas.

Como ya se ha mencionado, el adquirir una supercomputadora comercial puede convertirse en algo casi imposible para algunas instituciones educativas, ya que los presupuestos asignados muchas veces no son suficientes para poder cubrir la demanda de estas necesidades.

En este capítulo se describe el diseño del sistema de procesamiento paralelo Linda-BUAP, el cual permita a las instituciones educativas contar con una maquina paralela para desarrollar aplicaciones que demanden una gran cantidad de cálculos aritméticos y lógicos y poca comunicación con la memoria.

El sistema Linda-BUAP es un lenguaje de coordinación desarrollado con fines didácticos para la investigación, el cual, será el precedente para el desarrollo de futuros sistemas de memoria compartida, sistemas operativos paralelos, etc.

Durante el desarrollo de este sistema se ha buscado realizar mejoras y aportaciones con respecto a su antecesor Linda, con el fin de presentar un sistema estable y con un rendimiento eficiente.

Basándose en las características de sistemas operativos *libres* como lo es Linux, este sistema (fuentes y ejecutables) estará disponible para los alumnos de licenciatura y postgrado de la facultad de ciencias de la computación para futuras mejoras y aportaciones.

3.2 Descripción General

El sistema Linda–BUAP se basa en un esquema basado en la combinación de hardware y software, donde el hardware lo representa un Cluster de computadoras personales conectadas bajo una red LAN con arquitectura MIMD (Multiple Instructions, Multiple Data) con memoria virtual compartida y la parte de software esta representado por dos programas que trabajan bajo el modelo de Cliente – Servidor.

Este sistema en su arquitectura se puede observar como una máquina, basada en el modelo de Von Neumann, la cual cuenta con una unidad de control(UC), una unidad de memoria y varias unidades de procesamiento(CPU's) ver Fig. 3.1.

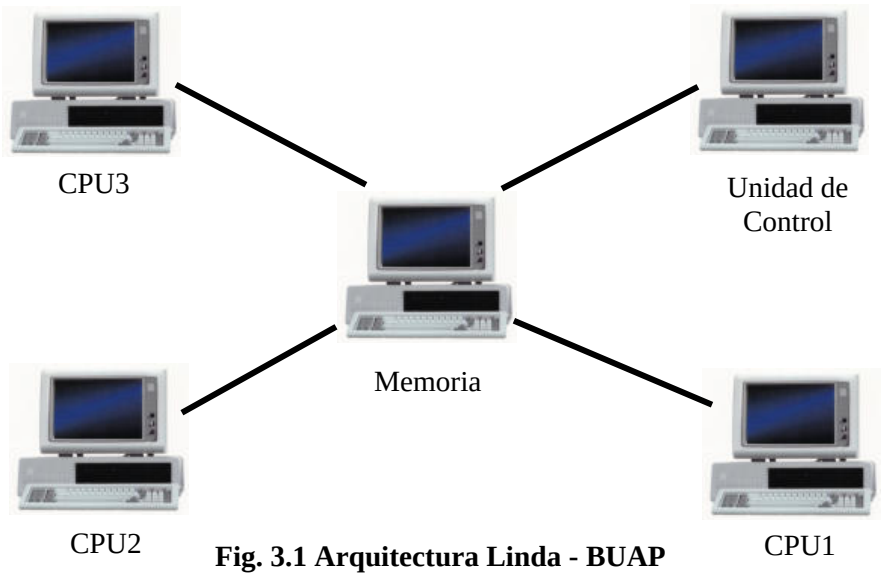


Fig. 3.1 Arquitectura Linda - BUAP

Este sistema utiliza una memoria virtual compartida representada por un programa llamado ***Master***, el cual trabaja bajo el esquema de Espacio de Tuplas. Aquí es donde se almacenarán los datos y programas que serán ejecutados por los procesadores. Este Maestro, será un servidor que recibirá peticiones de servicios(conexiones, datos, código) , estando siempre activo para poder atender a dichas peticiones, las cuales pueden provenir de los procesadores o de la unidad de control. A través de esta memoria es como se van a comunicar los diferentes procesos que se encuentren ejecutándose en el sistema.

Las unidades de procesamiento estarán representadas por un programa llamado ***Worker*** el cual se encargará de ejecutar las aplicaciones desarrolladas por el usuario, estas aplicaciones demandarán datos para poder procesarlos y generar resultados, estas demandas de datos se harán a través de tuplas enviadas de la aplicación al Trabajador para que este a su vez las envíe al Maestro, que es el encargado de proveer datos, este responderá al trabajador con el dato solicitado y

éste a su vez se lo enviará a la aplicación. Al finalizar la ejecución de la aplicación del usuario, ésta enviará los resultados al trabajador para que éste a su vez los envíe al master.

La comunicación de datos y código entre el maestro, los trabajadores y la unidad de control se hace a través de 6 instrucciones de coordinación: In, Out, Rd, Inp, Rdp y Eval, las cuales se encargarán de recuperar, depositar datos y depositar código en el espacio de tuplas en el master. Ver Fig. 3.2

Esta sistema se encuentra funcionando sobre el sistema operativo Linux, ya que por ser un sistema operativo libre no representa ningún tipo de inversión para obtener una distribución.

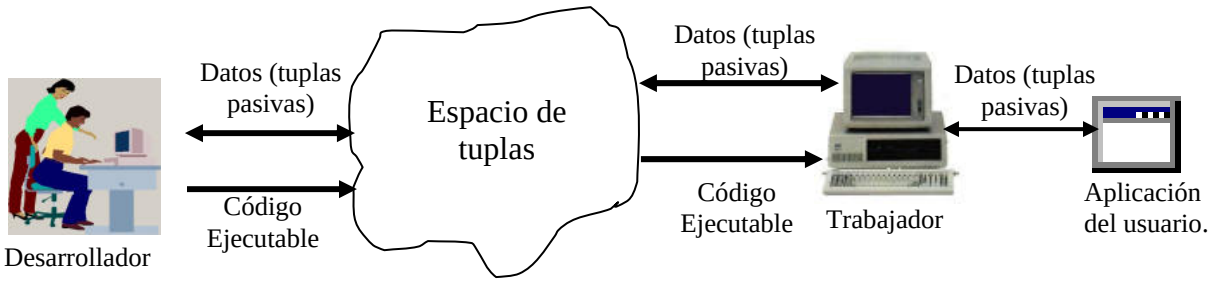


Fig. 3.2
Funcionamiento del
Sistema Linda -BUAP

3.2.1 Comunicación de datos y código entre Trabajadores y Maestro.

La comunicación de datos y código entre los trabajadores y el maestro se realiza a través de sockets TCP. Esta comunicación se realiza utilizando 2 sockets TCP. El Socket Principal será por el cual se estará enviando y recibiendo datos a través de tuplas del y hacia el Maestro, éste socket permanecerá activo siempre hasta que finalice la ejecución del Trabajador, el Socket Secundario, el cual se creará solo para recibir el código ejecutable del master, éste código ejecutable es la aplicación que será ejecutada por el trabajador después de terminar la recepción, este socket será cerrado una vez que el archivo ha sido recibido en el trabajador. Ver Fig. 3.3

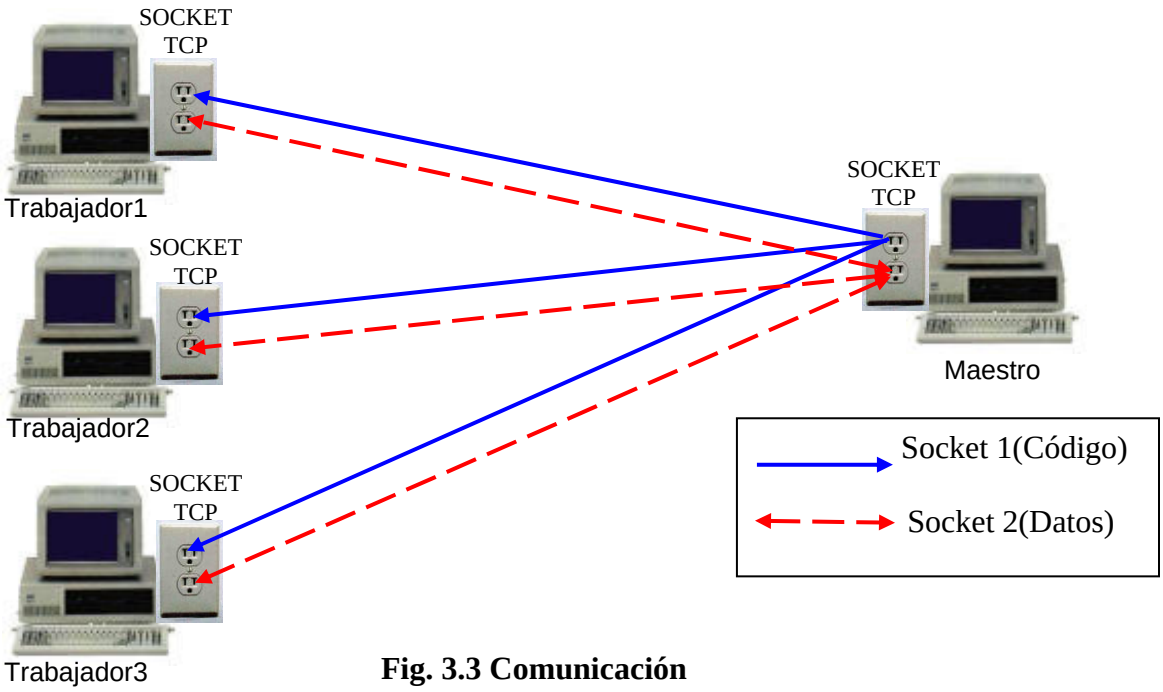


Fig. 3.3 Comunicación Trabajador - Maestro

3.2.2 Comunicación de datos entre Trabajadores y la aplicación del usuario.

Una vez que el código ejecutable ha sido recibido por el trabajador, éste lanza un proceso para empezar su ejecución. Durante la ejecución de esta aplicación habrá demanda de datos para ser procesados y una vez procesados se tendrán que devolver resultados al trabajador.

Todo lo anterior se hace a través de un Socket de Unix, el cual, nos permite comunicar datos entre aplicaciones que se encuentran ejecutándose en la misma computadora. El socket se crea al iniciar la ejecución de la aplicación, y permanece activo para que la aplicación puede enviar y recibir datos a través de tuplas a y desde el trabajador. Este socket se destruye al finalizar la aplicación en ejecución. Ver fig. 3.4.

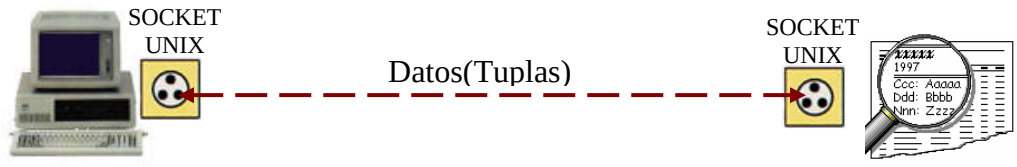


Fig. 3.4 Comunicación Trabajador-Aplicación vía Socket Unix

3.2.3 Comunicación de datos entre Desarrollador y el Maestro

El desarrollador será la máquina que represente la unidad de control. En esta máquina se desarrollarán las aplicaciones del usuario. Al ejecutar estas aplicaciones, se necesitarán enviar datos a la memoria (Maestro) y también enviar el código ejecutable de las aplicaciones que serán ejecutadas por cada uno de los trabajadores.

Esto se realiza utilizando 2 Sockets TCP, El primer socket se crea desde el momento en que se conecta con el maestro para enviar y recibir datos, el segundo socket se crea para enviar el código ejecutable al maestro , este socket se destruye una vez enviado el archivo, el primero permanece activo hasta que termina la ejecución de la aplicación. Ver Fig. 3.5.

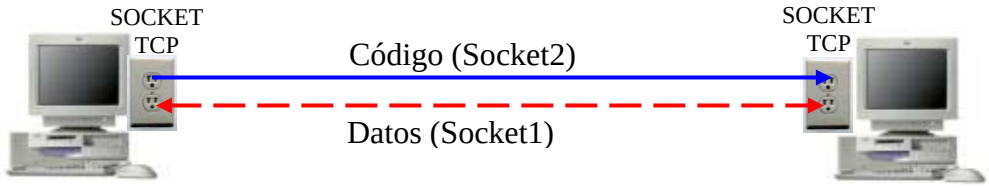
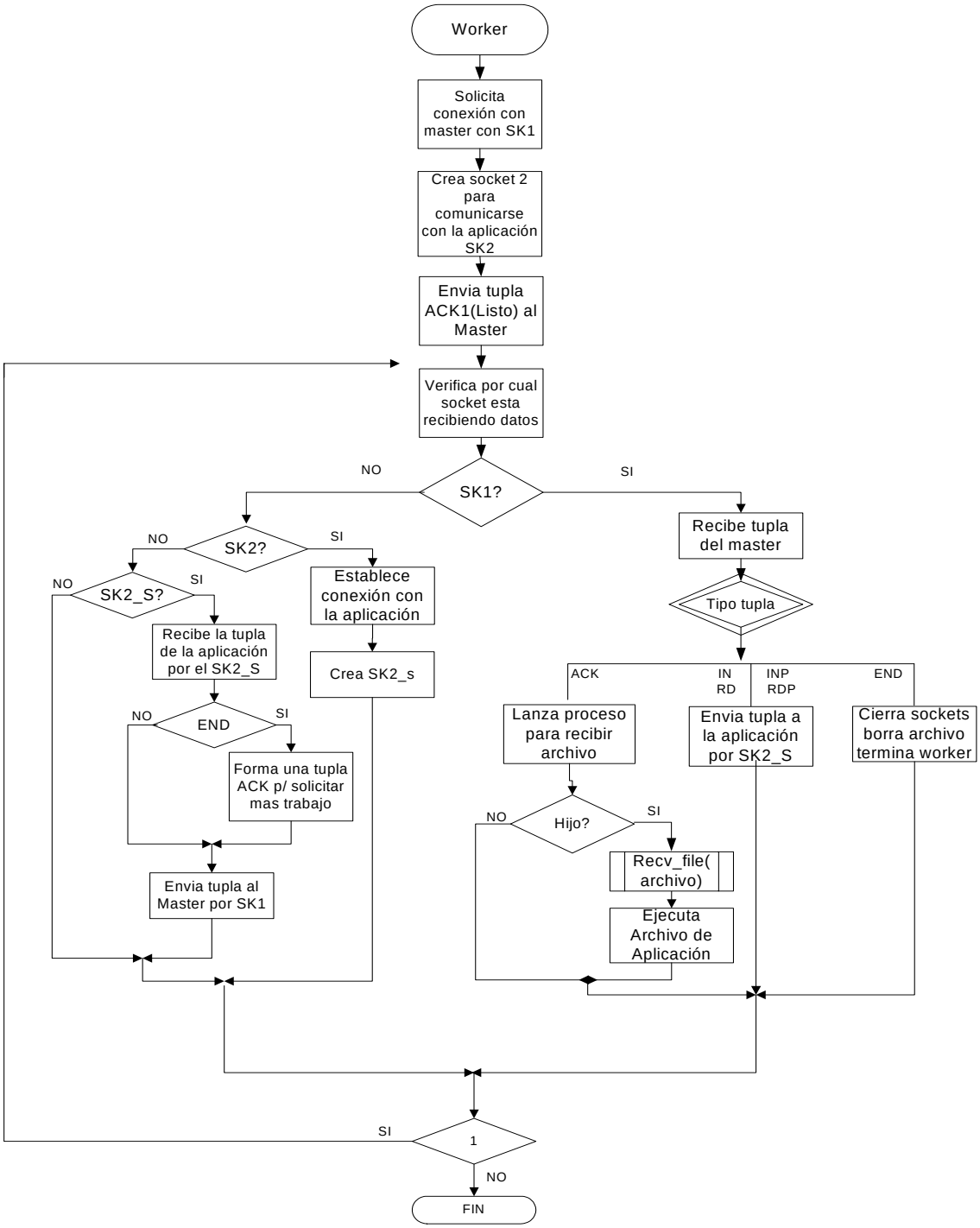


Fig. 3.5 Comunicación Desarrollador-Maestro

3.3 Unidad de procesamiento(worker).

Los workers o trabajadores son los encargados de ejecutar las aplicaciones del usuario y de recibir las peticiones de datos de ésta aplicación en ejecución. El worker a su vez, hace la petición al master(memoria) y espera la respuesta de éste, con el dato requerido para posteriormente enviárselo a la aplicación en ejecución. Ver fig. 3.6



3.6 Diagrama Worker (trabajador)

Envía solicitud de conexión con el maestro, una vez conectado y creado el socket principal (TCP) envía una tupla ACK1, la cual le indica al maestro que el trabajador esta listo para recibir trabajo.

Crea el socket secundario (UNIX) con el cual establece comunicación con la aplicación una vez que se este ejecutando.
Inicia un ciclo infinito esperando ya sea tuplas de datos, tuplas de código o tuplas de control.

Verifica por cual socket se esta recibiendo información:

- Si es por el socket principal se esta recibiendo una tupla del master.
  - Si es una tupla ACK: lanza un proceso hijo para recibir el archivo (código) ejecutable del maestro y una vez recibido inicia la ejecución del archivo ejecutable en el mismo proceso hijo.
  - Si es una tupla In, Inp, Rd, Rdp , envía esa tupla a la aplicación en ejecución a través del socket secundario.
  - Si es una tupla End , cierra todos los sockets creados, borra el archivo ejecutable que se recibió del disco duro local de la máquina trabajador para no dejar basura y termina la ejecución del programa worker.
- Si es por el socket secundario(de la aplicación):
  - Se esta recibiendo una tupla de la aplicación que puede ser un in,inp,rd o rdp, lo cual indica que está solicitado datos para ser procesados y la aplicación esperará a recibir la respuesta del trabajador con su dato solicitado, o bien, puede ser una tupla out para depositar algún resultado.
  - Si la tupla recibida es un END, indica que la aplicación a finalizado, por lo tanto, forma una tupla ACK y la envía al maestro para indicar que el trabajador esta disponible para recibir mas trabajo en caso de exista trabajo pendiente por ejecutar y espera la respuesta del maestro.

3.3.1 Función que recibe el código ejecutable del master.

Esta función, que es ejecutada dentro de un proceso hijo, se ejecuta cada vez que el trabajador recibe una tupla ACK, la cual es un indicativo de que hay trabajo pendiente por ejecutar en el maestro y le solicita al trabajador que este listo para recibir un archivo ejecutable. Ver Fig.3.7.

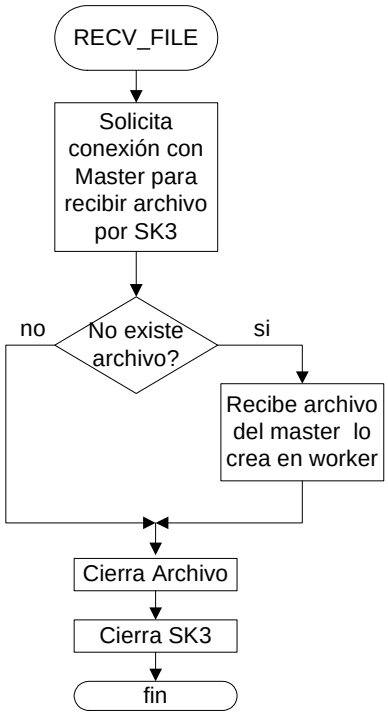


Fig. 3.7 Diagrama de la función Recv_File

Solicita conexión con el maestro, y crea socket 3 (TCP) para recibir el archivo (código) ejecutable. Verifica en el disco duro local de la máquina trabajador si el archivo ya existe para no volverlo a recibir e invertir tiempo innecesario de transmisión que puede repercutir en tiempo total de procesamiento, si no existe, entonces comienza la recepción del archivo a través del socket 3 y lo crea en el disco duro local de la maquina trabajador. Al finalizar la recepción cierra el archivo y cierra el socket3.

3.4 Biblioteca de funciones de coordinación.

Las funciones de coordinación serán las encargadas de enviar y recuperar tuplas en el espacio de tuplas del maestro. Estas instrucciones son Out, In, Rd, Inp, Rdp y Eval. Se diseñaron 2 versiones para las primeras 5 instrucciones, unas para ser utilizadas en la aplicación principal del usuario, las cuales, utilizan el socket TCP para comunicarse con el maestro y las segundas con el mismo nombre, pero utilizan el socket Unix para comunicarse con el trabajador.

3.4.1 Descripción del diagrama de la función init_linda.

Función que se encargará de establecer la conexión, vía socket TCP, con el maestro para poder intercambiar datos a través de tuplas. Ver Fig. 3.8.

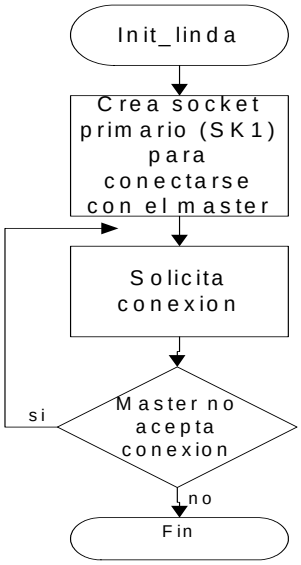


Fig.3.8 Diagrama de la Función Init_Linda

Crea el socket primario para conectarse con el maestro. Solicita la conexión, si el master esta disponible se establece la comunicación y queda en espera de recibir o listo para enviar datos(tuplas), si no estuviera listo el maestro, queda en un ciclo infinito hasta que el maestro se encuentre disponible para atender su petición.

Esta función deberá ser incluida al inicio de los programas principales que el usuario desarrolle.

3.4.2 Descripción del diagrama de la función end_linda.

Esta función se utiliza para finalizar la conexión de la aplicación con el maestro de forma correcta. Para esto se utiliza una tupla de control llamada ENDA la cual le indica al maestro que no necesita más de sus servicios, además se encargará de cerrar el socket principal. Ver fig. 3.9

Fig.3.9 Diagrama de la Función End_Linda

3.4.3 Descripción del diagrama de la función Out.

La función Out se encarga de depositar una tupla en el espacio de tuplas. Esta función forma la tupla out que se enviará al maestro, analizando una cadena de entrada que indicará el tipo de dato (entero, flotante, cadena, carácter) y el tipo de acceso (“%”= acceso por valor) de cada variable o valor que vaya a ser enviado. Una vez formada la tupla, la envía por el socket principal al maestro. Ver fig. 3.10.

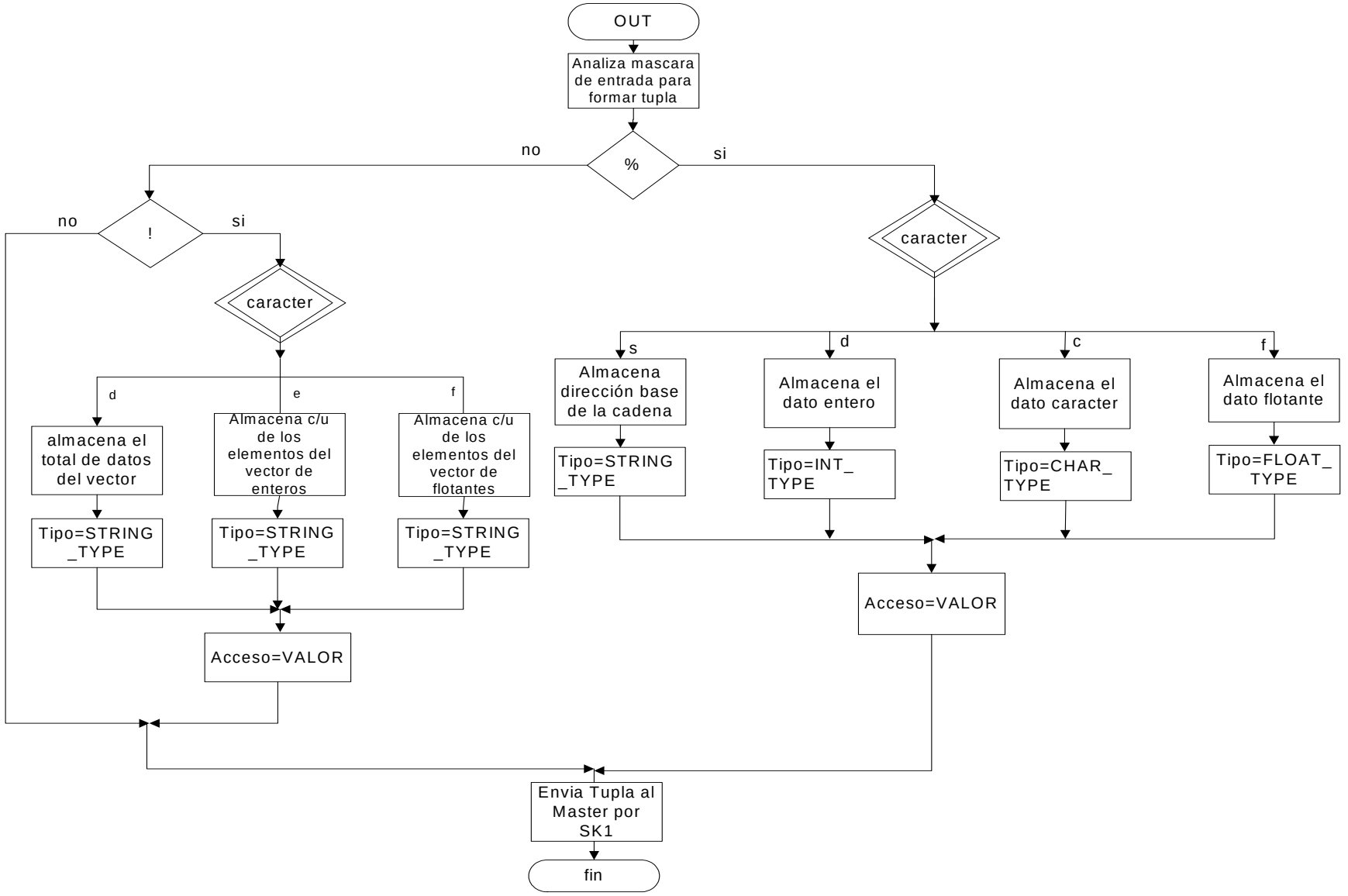


Fig.3.10 Diagrama de la Función Out

Se analiza carácter por carácter la cadena de entrada proporcionado en la tupla, hasta terminar con todos los caracteres.

Si el carácter analizado es un **"%"**, hay que verificar el carácter posterior a éste, el cual indicará el tipo de dato del valor o la variable correspondiente en los siguientes casos:

- Si el carácter es **"s"**: almacena la dirección base de la cadena de caracteres y establece el tipo de dato como **STRING_TYPE** en la tupla.
- Si el carácter es **"d"**: almacena el dato entero y establece el tipo de dato como **INT_TYPE** en la tupla.
- Si el carácter es **"c"**: almacena el dato carácter y establece el tipo de dato como **CHAR_TYPE** en la tupla.
- Si el carácter es **"f"**: almacena el dato de tipo flotante y establece el tipo de dato como **FLOAT_TYPE** en la tupla.

En todos los casos anteriores el tipo de acceso será por **VALOR**, ya que se almacena el valor del dato explícito, y no direcciones. Para esta función no existe el tipo de acceso **REFERENCIA**.

Si el carácter analizado es un **"!"** , indica que los datos a almacenar se encuentran dentro de un arreglo unidimensional(vector). En este sistema solo se pueden enviar vectores. Si el carácter que posterior a **"!"** es:

- **"d"** : almacena el dato numérico que indica el total de elementos del vector en la tupla.
- **"e"** : almacena cada uno de los elementos numéricos de tipo entero del vector en la tupla.
- **"f"** : almacena cada uno de los elementos numéricos de tipo flotante del vector en la tupla.

Para todos los casos anteriores el tipo de acceso será por **VALOR**.

Una vez formada la tupla con todos los datos necesarios se envía por el socket primario al maestro para ser almacenada en el espacio de tuplas.

3.4.4 Descripción del Diagrama de las funciones In y Rd.

La función In y Rd trabajan de forma muy similar. Ambas envían una solicitud de búsqueda de una tupla en particular en el espacio de tuplas del maestro, si la tupla no existe, se bloquea el proceso que la solicitó hasta que se deposite la tupla por la que está esperando. Este bloqueo se realiza de manera implícita por la naturaleza de los mismos sockets que se utilizan para la comunicación, ya que son sockets bloqueantes.

La diferencia que existe entre ambas se establece en función de que, en el caso de In, si la tupla existe en el espacio de tuplas la toma y la borra, Rd no borra las tuplas, solo recupera una copia de ella, pero todo esto sucede en el maestro así es que en este caso el funcionamiento es el mismo. Ver. Fig.3.11.

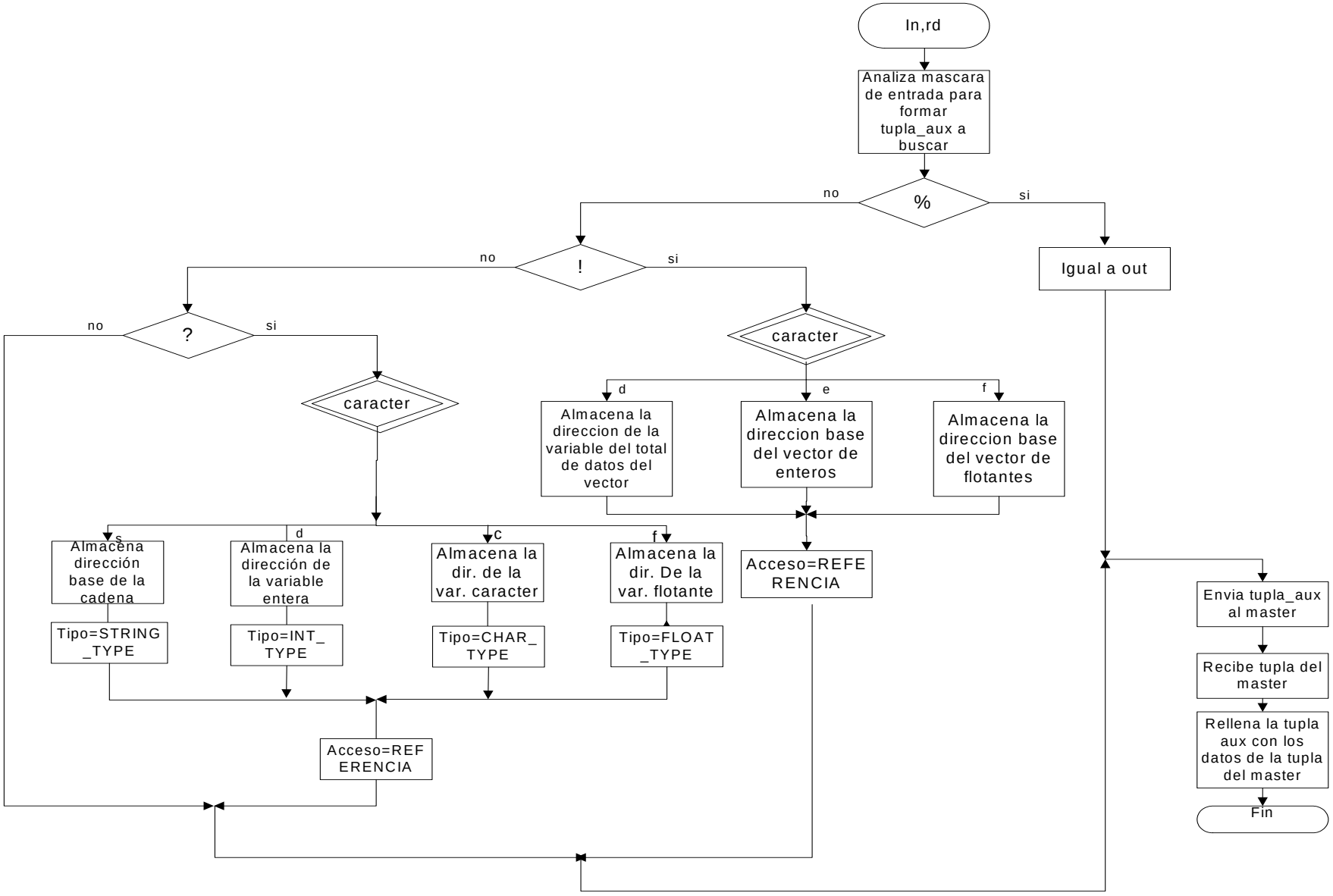


Fig.3.11 Diagrama de las Funciones In, Rd

Las funciones In y Rd forman en primera lugar la tupla prototipo que será buscada en el maestro. Para formar esta tupla prototipo (tupla_aux), analiza carácter por carácter la cadena de entrada de la instrucción, en donde:

- Si el carácter es **"%"** realiza las mismas acciones que realiza la función **Out** en el correspondiente carácter para formar una tupla con acceso por **VALOR.**
- Si el carácter es **"!"**, se trata de un arreglo unidimensional (vector) donde si el carácter posterior a éste es:
  - **"d"**: almacena la dirección de la variable que contendrá el total de datos del vector.
  - **"e"**: almacena la dirección base del vector que contendrá elementos enteros.
  - **"f"**: almacena la dirección base del vector que contendrá elementos flotantes.

  En todos los casos anteriores el tipo de acceso es por **REFERENCIA**, ya que se están almacenando direcciones memoria (apuntadores) de donde se almacenarán los datos correspondientes que retornarán del maestro.
- Si el carácter es **"?"** , se trata de variables por referencia, es decir se almacenarán direcciones de memoria y no datos explícitos. Si el carácter posterior es:
  - **"s"** : Se almacena la dirección base de la cadena de caracteres y se establece el tipo de dato como STRING_TYPE.
  - **"d"** : Se almacena la dirección de la variable que contendrá un dato entero y se establece el tipo de dato como INT_TYPE.
  - **"c"** :Se almacena la dirección de la variable que contendrá un carácter y se establece el tipo de dato como CHAR_TYPE.
  - **"f"** : Se almacena la dirección de la variable que contendrá un flotante y se establece el tipo de dato como FLOAT_TYPE.

  En todos los casos se establece el tipo de acceso por **REFERENCIA**.

Una vez creada la tupla prototipo (tupla_aux), ésta es enviada por el socket principal al maestro. Si la tupla existe, recibe una tupla con datos como respuesta del maestro (tupla_t). Por último, almacena cada dato de tupla_t en la correspondiente dirección almacenada en tupla_aux. Los datos contenidos en ésta tupla serán los utilizados en la aplicación del usuario.

3.4.5 Descripción del Diagrama de las funciones Inp, Rdp.

Estas funciones trabajan en forma muy similar a sus predecesoras In y Rd. La diferencia fundamental que existe entre ambas funciones es que Inp y Rdp no son bloqueantes, es decir solo esperan recibir un valor boleano (verdadero o falso) para saber si la tupla existe o no en el espacio de tuplas. Ver Fig. 3.12
En caso de que la tupla exista, si se desea recuperar deberá utilizar un in o un rd. Estas funciones representan una forma de censar por una tupla sin bloquear al proceso en ejecución.

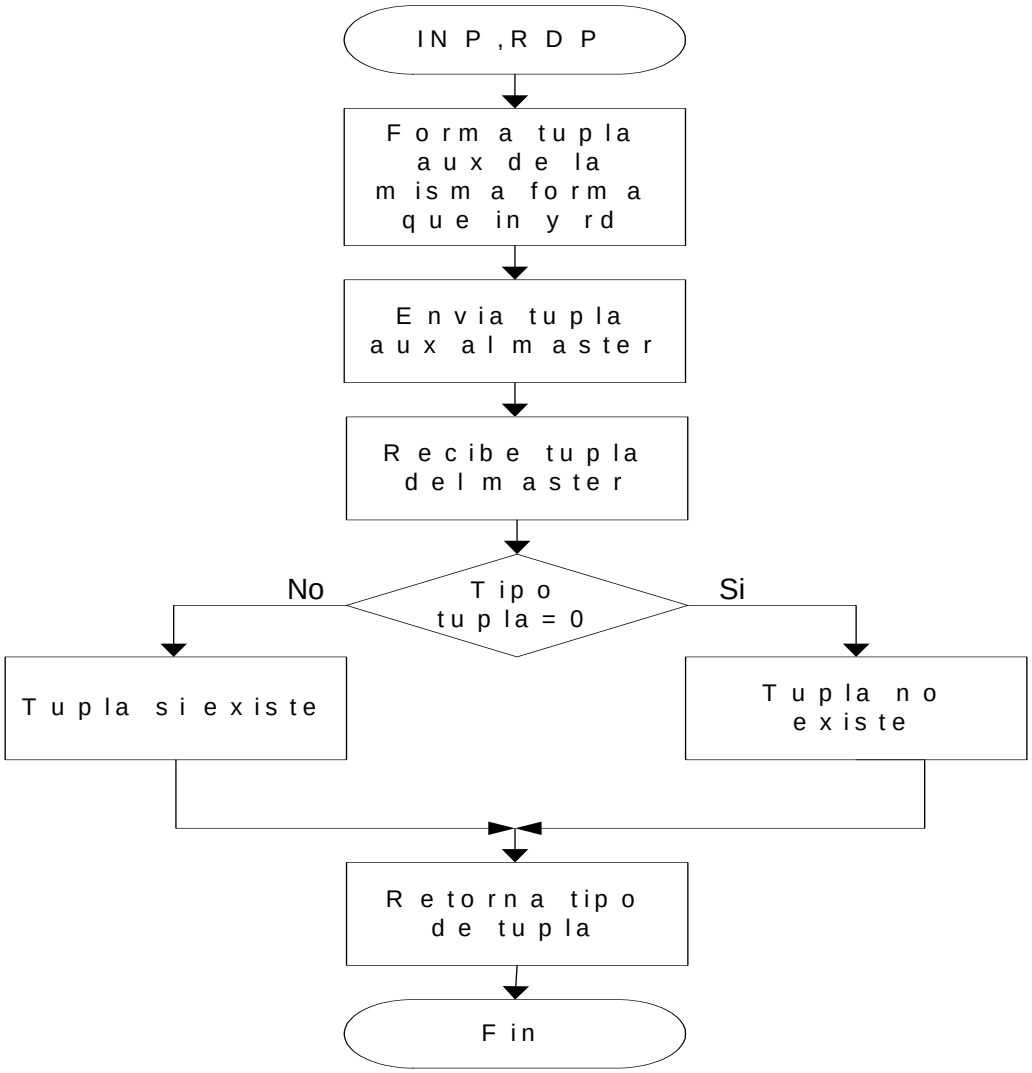


Fig. 3.12 Diagrama de las Funciones Inp y Rdp

3.4.6 Descripción del Diagrama de la función Eval.

La forma con la cual la aplicación del usuario envía código a ejecutar en los diferentes trabajadores es a través de la función eval. Ver. Fig. 3.13.

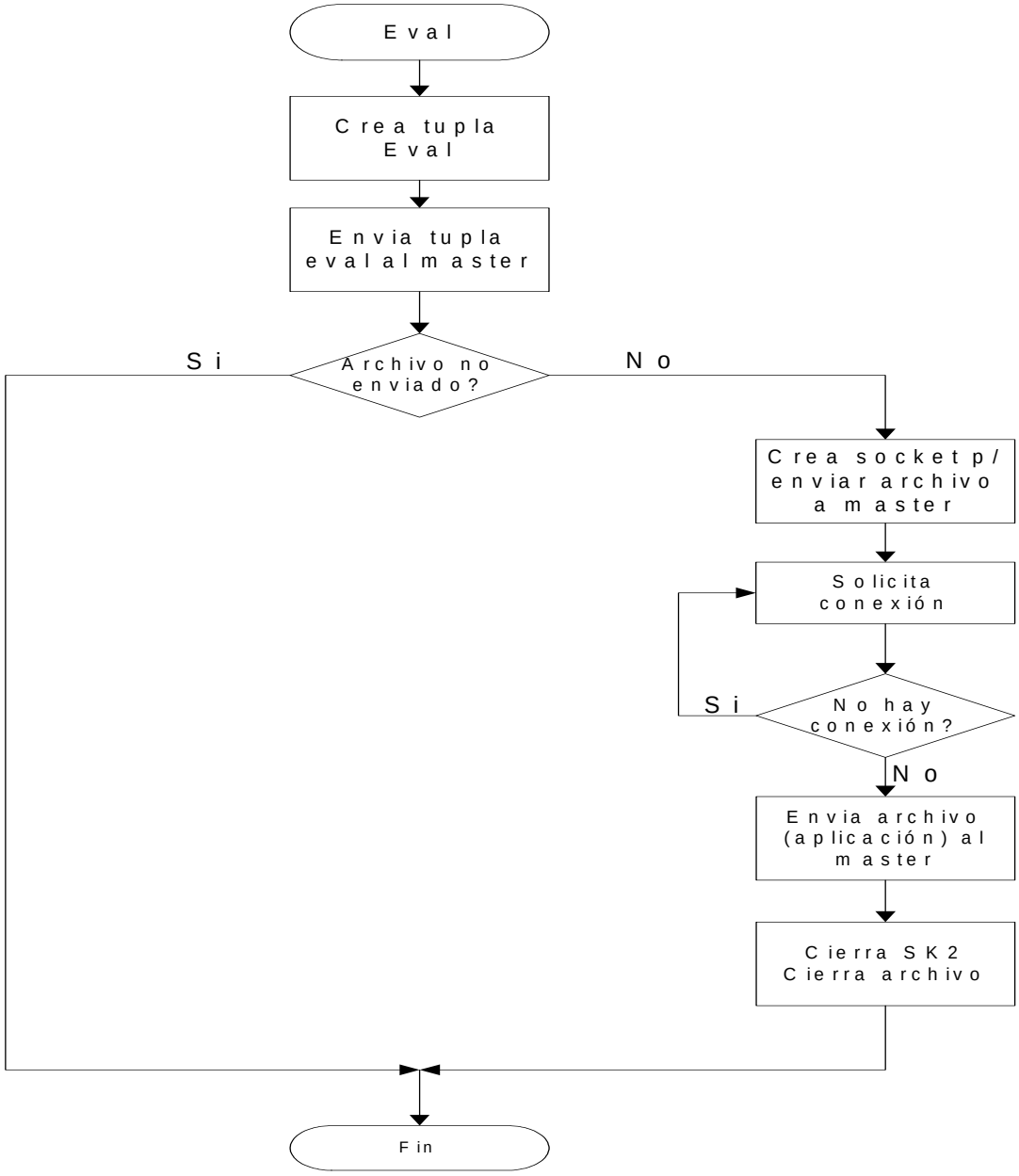


Fig. 3.13 Diagrama de la función Eval.

Crea una tupla tipo EVAL, para ser enviada al maestro por el socket principal, y verifica si el archivo con el código ejecutable no ha sido enviado ya, ya que de ser así no se vuelve a enviar para evitar consumo de tiempo en la transmisión de datos, únicamente se envía la tupla para avisar que aún existe trabajo pendiente. Si no ha sido enviado se crea el socket secundario para establecer comunicación con el maestro y enviar el archivo. Si el maestro no esta listo para la conexión, entra en un ciclo infinito hasta que se pueda establecer la conexión. Una vez conectado se inicia el envío del archivo. Una vez concluida la emisión del archivo se cierra el socket secundario.

3.4.7 Descripción del Diagrama de las funciones init_app y fin_app.

Como ya se ha mencionado antes, el código que se distribuye a los trabajadores, es ejecutado por ellos y para establecer una comunicación de datos entre trabajadores y la aplicación se utiliza un Socket Unix. Por esta razón fue necesario diseñar un par de funciones para crear y destruir dicho socket . Ver Fig. 3.14_a y 3.14_b.

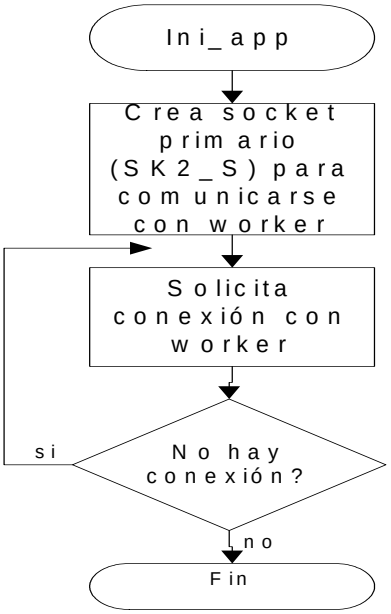


Fig. 3.14_a Diagrama de la función Ini_app

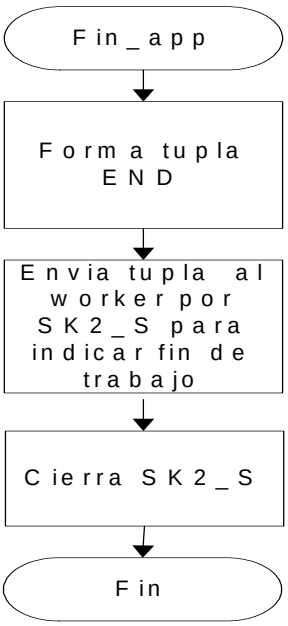


Fig. 3.14_b Diagrama de la función Fin_app

La función ini_app se deberá invocar al inicio de la aplicación del usuario que será distribuida a los trabajadores. Crea un socket de Unix , por el cual enviará y recibirá tuplas pasivas del trabajador (worker). Establece un ciclo infinito hasta lograr conexión con el trabajador.

La función fin_app, se deberá invocar al finalizar la aplicación que será distribuida a los trabajadores, cierra y destruye el socket Unix utilizado para la comunicación de datos con los trabajadores, con el objeto de finalizar correctamente la aplicación del usuario.

Al igual que estas dos funciones, se crearon versiones adicionales de las funciones In, Inp, Rd, Rdp para la aplicación que será distribuida a los trabajadores. Estas funciones realizan las mismas tareas que las anteriormente descritas, solo que a diferencia de las primeras 4, estas escriben y leen tuplas (datos) del socket de Unix, ya que los datos que comunican los envían al trabajador(worker) y no con la memoria.

Capítulo 4
Implementación de Linda-BUAP

4.1 Introducción

El sistema Linda-BUAP fue desarrollado bajo la plataforma que brinda el sistema operativo Linux Mandrake versión 9.2 y 7.2, utilizando el compilador de lenguaje C estándar, GCC y herramientas propias de la programación en Unix como procesos, y herramientas de comunicación como los sockets y llamadas al sistema.

Este sistema está basado en un modelo cliente – servidor, que es el modelo mas utilizado en el cómputo distribuido. Este documento describe a los trabajadores como clientes de una memoria virtual compartida. Los trabajadores enviarán su solicitud de trabajo y posteriormente enviará solicitudes de datos a la memoria. A su vez los trabajadores se convierten en servidores en el momento de ejecutar la aplicación del usuario (cliente).

Para el monitoreo que proporcionan los trabajadores se utilizaron funciones incluidas en la biblioteca ncurses.h de linux, ya que en el ANSI C no existen librerías como el conio.h del turbo C de borland, así que para una ejecución favorable de la aplicación “worker”, la distribución de linux sobre la que se ejecute deberá tener instalado el paquete ncurses.

En el desarrollo de las funciones de coordinación se utilizan listas de argumentos de longitud variable, para poder capturar cualquier número de argumentos como entrada de las funciones de coordinación.

El hardware utilizado para el desarrollo fueron 2 computadoras Pentium III a 450 Mhz, con 64 MB. en RAM, disco duro de 40 GB. con tarjetas Ethernet 10/100 Mbps, 2 computadoras Pentium IV 1.6 Ghz con 128 en RAM y disco duro de 80 GB y tarjetas Ethernet 10/100 Mbps. y 2 Computadoras Pentium I a 133 Mhz. con 80 MB. en RAM con 2 GB de disco duro y tarjetas ethernet 10 Mbps. También se utilizó un concentrador (hub) Ethernet a 10 Mbps.

En este capítulo se describirá en primero lugar como fue implementado el programa *“worker”*, que dará vida a las maquinas trabajadores del sistema y posteriormente se describirá como fueron implementadas las funciones de coordinación que permiten la comunicación de datos en el sistema, utilizando una memoria virtual compartida.

4.2 Estructuras de Datos y Tipos.

Los trabajadores utilizan como unidad principal de comunicación de datos las tuplas pasivas, como unidad principal de comunicación de código ejecutable las tuplas activas y para enviar o recibir mensajes de listo, trabajo pendiente o final, las tuplas de control.

En la implementación de éste sistema se utilizaron diversas definiciones que son útiles dentro del desarrollo del código de los workers (trabajadores) y de las funciones de coordinación. Estas definiciones se encuentran contenidas en la librería TLib.h y se detallan a continuación:

#define IN 1 #define OUT 2 #define RD 3 #define RDP 4 #define INP 5 #define EVAL 6 #define ACK 7 #define END 8 #define ENDA 9 #define ACK1 10	Define un valor entero numérico progresivo para cada tipo de tupla. Donde de la 1 a la 5 son tuplas pasivas, la 6 es una tupla activa y de la 7 a la 10 son tuplas de control.
#define CHAR_TYPE 0 ***(tipo char)*** #define STRING_TYPE 1 ***(tipo cadena)*** #define INT_TYPE 2 ***(tipo entero)*** #define FLOAT_TYPE 3 ***(tipo flotante)*** #define INT_ARR 4 ***(arreglo de enteros)*** #define FLOAT_ARR 5 ***(arreglo de flotante)*** #define TOT_ARR 6 ***(total de datos del arreglo)***	Define para los tipos de datos de los argumentos de las tuplas pasivas, un número entero progresivo de acuerdo al tipo de dato.
#define VALOR 0 #define REFERENCIA 1	Define un tipo de acceso para los argumentos de las tuplas pasivas.

`void in(char *mask, ...);`
`void out(char *mask, ...);`
`void rd(char *mask, ...);`
`int inp(char *mask, ...);`
`int rdp(char *mask, ...);`
`void init_linda(char *master);`
`void end_linda();`
`void eval(char *nomarch);`
	Prototipos de las funciones que representan las funciones de coordinación.

Las tuplas(pasivas, activas o de control) están representadas por la estructura ***Struct Tupla***, que tiene la siguiente representación interna:

Tipo	Ip	Tot_ dat	aten dido	sig	argumentos					
					Tvar Valor Tacceso [0]	Tvar Valor Tacceso [1]	Tvar Valor Tacceso [2]	Tvar Valor Tacceso [3]	..	Tvar Valor Tacceso [6]

Esta estructura esta definida en lenguaje C de la forma siguiente:

```
struct tupla
{
   int tipo;
   char ip[15];
   struct datos argumentos[7];
   int tot_dat;
   int atendido;
   struct tupla *sig;
};

struct datos
{ int tvar; //tipo de datos
  union tipos valor;
  int tacceso; // VALOR, REFERENCIA
};

union tipos
{ char c,*cp;
  int i,*ip, td, *tdp;
  char s[15],*sp;
  double f,*fp;
  int ve[30];
  double vf[15];

};
```

La estructura principal ***struct tupla,*** esta formada por los campos:

Tipo: representa el tipo de la tupla (en cualquiera de sus 10 posibles valores (in, out, rd, rdp, inp, eval, ack, ack1, end, enda).

ip: respresenta la dirección ip de quien envía la tupla. Esto solo se utiliza para las tuplas ACK1 y ACK.

Tot_dat: representa el total de datos (argumentos) que se envían por una tupla pasiva.

Atendido y *sig: son dos campos que son utilizados por el maestro para censar si una petición de una tupla in ya fue atendida, y para formar una lista simplemente ligada de tuplas pasivas.

Argumentos: este campo que es un arreglo que contendrá la lista de argumentos que contedrán las tuplas pasivas. Cada elemento de este arreglo es del tipo **struct datos,** la cual almacena:

Tvar: este campo indica el tipo de dato de cada argumento de la tupla estos pueden ser CHAR_TYPE, INT_TYPE, FLOAT_TYPE, STRING_TYPE, INT_ARR, FLOAT_ARR y TOT_ARR.

Tacceso: este campo indica el tipo de acceso que tendrá el argumento de la tupla, esto puede ser VALOR o REFERENCIA.

Valor: este campo almacena en sí el valor enviado o recibido a través de la tupla, o bien almacena direcciones de memoria para posteriormente almacenar en esas direcciones datos devueltos por una tupla. La **union tipos** contiene campos de tipo apuntador a cada tipo de dato posible donde se almacenarán las direcciones y también variables de tipo simple a donde se almacenaran los datos en sí. Contiene dos campos que son de tipo arreglo, que se utilizan para enviar arreglos de enteros, flotantes y cadenas.

4.3 Diseño Final.

4.3.1 Unidad de procesamiento (worker).

El programa *worker* al iniciar su ejecución crea un socket principal (Socket TCP) por el cual se conectará con el maestro para recibir y enviar tuplas pasivas. Así mismo define un socket Unix con el cual establecerá una comunicación cliente – servidor con la aplicación del usuario, y creará un tercer socket para la recepción del archivo a ejecutar. Para esto es necesario definir las variables que se muestran en la figura 4.1.

```
#define NAME "my_sock"
int  orig_sock, new_sock, clnt_len;
static struct sockaddr_un clnt_adr, serv_adr;

int create_socket;
struct sockaddr_in address,bs,des;

int fp;
int socket2=-1;
struct sockaddr_in address2;

struct tupla t,tpla;
fd_set worker,read_fds;
int fdmax;
FD_ZERO(&worker);
FD_ZERO(&read_fds);
```

Fig. 4.1 Variables Principales del Worker

Las primeras tres líneas de la figura 4.1 definen los elementos necesarios para crear y utilizar un servidor de socket Unix, ya que el worker con respecto a la aplicación se convertirá en un servidor (orig_sock) quien podrá recibir y enviar tuplas pasivas, desde y hacia la aplicación en ejecución quien será un cliente (new_sock) del worker, comunicándose a través del socket “my_sock”.

Las siguientes dos líneas definen las variables necesarias para crear el socket principal (socket de cliente *create_socket*) para conectarse con el master y poder enviar y recibir tuplas pasivas y de control.

Las siguientes tres líneas definen la variable fp que se utiliza como un puntero al archivo que se creará en el worker al ser recibido en forma correcta. Para poder realizar esta recepción es necesario crear un socket secundario (socket2) por el cual se recibirá, byte a byte el archivo proveniente del maestro para ser ejecutado posteriormente.

Para el manejo de las tuplas se declaran variables del tipo struct tupla.

Para censar los sockets y saber de cual se esta recibiendo información se utiliza la sentencia select, para la cual es necesario definir conjuntos de descriptores, así worker y read_fds son dos conjuntos de descriptores para ser utilizados en dicha sentencia, asi mismo se inicializan con la instrucción FD_ZERO.

Una vez vistas las declaraciones de las variables principales, se crea el socket principal, ver Fig. 4.2.

```
if ((create_socket = socket(AF_INET,SOCK_STREAM,0)) <= 0)
{mvwprintw(chw5,2,2,"Error al crear socket\n"); wrefresh(chw5); }
address.sin_family = AF_INET;
address.sin_port = htons(8080);
inet_pton(AF_INET,argv[1],&address.sin_addr);

  while(1)
  {
  if(connect(create_socket,(struct sockaddr*)&address,sizeof(address))<0)
          continue;
  else
          break;
  }
```

Fig. 4.2 Creación del socket principal

Se crea el socket TCP principal (create_socket) por donde se enviarán y recibirán tuplas pasivas y de control del master. Se establece un ciclo infinito para estar intentando la conexión hasta que el master se encuentre listo o disponible y evitar con esto rupturas del programa por errores del sistema.

Posterior a esto se crea el socket de Unix, como un servidor de conexiones, ya que el worker se convertira en un servidor de la aplicación del usuario, ver Fig. 4.3

```
if ((orig_sock=socket(AF_UNIX,SOCK_STREAM, 0))<0){
          perror("error se socket");
          exit(1);}
        serv_adr.sun_family=AF_UNIX;
        strcpy(serv_adr.sun_path, NAME);
        unlink(NAME);
        if(bind(orig_sock, (struct sockaddr*)
&serv_adr,sizeof(serv_adr.sun_family) +
strlen(serv_adr.sun_path))<0)
        { perror("Error de bind ");
          close(orig_sock);
          unlink(NAME);
          exit(2);}
        listen(orig_sock,1);
        clnt_len=sizeof(clnt_adr);
```

Fig. 4.3 Creación del Socket Unix

Una vez creado el socket de Unix se anexan los dos sockets anteriores a los conjuntos de worker y read_fds para poder ser evaluados por el select.
Select es una fución que verifica si el socket i ha recibido información, si es así lo atiende y si no es así pasa al siguiente socket, esto se repite hasta evaluar el descriptor de socket fdmax. Select únicamente evalúa los sockets contenidos en el conjunto de descriptores definidos (worker). Ver fig.4.4 y fig. 4.6

```
FD_SET(create_socket,&worker);
FD_SET(orig_sock,&worker);
read_fds=worker;
if(create_socket>orig_sock) fdmax=create_socket;
else fdmax=orig_sock;
monitor();
```

Fig. 4.4 Inicialización de los conjuntos de descriptores de sockets y monitor

Ya establecida la comunicación con el master, se crea y envía una tupla **ACK1**, la cuál dará aviso al master de que hay un worker(trabajador) listo para recibir trabajo a procesar, en esta tupla se envía la dirección IP del worker, ver fig. 4.5.

```
t.tipo=ACK1;
strcpy(t.ip,rmt_host);
strcpy(t.argumentos[0].valor.s,"Listo");
n=send(create_socket,&t,sizeof(struct tupla),0);
```

Fig. 4.5 Creación y envío de la tupla ACK1

Se crea un ciclo infinito para estar evaluando constantemente los sockets, y además, una vez que el trabajador termina su tarea, volver a reiniciarse para esperar nueva tarea si es que aún existiera trabajo pendiente por desarrollar en el master.

```
read_fds=worker;

    if(select(fdmax+1,&read_fds,NULL,NULL,NULL)==-1)
    {
       continue;

    }
     for(i=0;i<=fdmax;i++)
     {

       if(FD_ISSET(i,&read_fds))
```

Fig. 4.6 Select censa los sockets contenidos en el conjunto read_fds

Si el socket por el cual recibe datos es el socket principal (create_socket), indica que recibirá una tupla del master y se tendrá que evaluar para ver que tipo de tupla es para tomar acciones diferentes, ver fig. 4.7.

```
if(i== create_socket)
        {   n=recv(create_socket,&t,sizeof(struct tupla),0);
              c1++;
              if(x>15)
              {x=3;wborra(x,1,15,24,chw2);wrefresh(chw2);}
              tipo_tupla(t.tipo);
           mvwprintw(chw2,x,1,"%d- %s %s ",c1,nomtupla,t.argumentos[0].valor.s);
              wrefresh(chw2);x++;
           switch(t.tipo)
           {case ACK:
              nomarch=t.argumentos[0].valor.s;
                      if(strcmp(nomarch,"SOLICITUD")!=0)
              {
               strcpy(nomarch2,t.argumentos[0].valor.s);

               strcpy(ruta,"./");
               strcat(ruta,nomarch);
               if(fork()==0)
               {
                              if(strcmp(nomrecibido,nomarch)!=0)
                              {
                               recv_file(argv[1]);
                              }


                    execlp(ruta,(char*) 0);
                 close(new_sock);exit(0);
               }//if fork
                else {

                                 strcpy(nomrecibido,nomarch);
                                }

                      } // if strcmp solicitud
                      else continue;
             break;
```

Fig. 4.7 Tuplas recibidas del master(ACK)

Si la tupla recibida es un **ACK**, indica que hay trabajo por realizar, así guarda el nombre del archivo que va a recibir en la variable ***nomarch***, y lanza un proceso utilizando la función *fork.* En el proceso hijo se evalúa para saber si no se ha recibido ya un archivo con ese nombre, si no es así, se invoca a la función ***recv_file*** *(fig. 4.12)*, con la IP del master como argumento, tomada de la línea de comandos. Una vez recibido el archivo se manda a ejecutar utilizando la función *execlp.* Como es sabido, si alguna variable se modifica en el proceso hijo, estos cambios no se ven reflejados en el proceso padre ya que no comparten el ambiente de datos, por tal motivo es necesario que en el proceso padre se asigne el nombre del archivo recibido a la variable *nomenviado* para que permanezca

actualizado, ya que esta variable es una bandera que nos indica si hay que recibir o no el archivo nuevamente.

En caso de que la tupla recibida sea una **tupla pasiva** (In, Inp, Rd, Rdp,) o un cero, se envía la tupla a la aplicación del usuario a través del socket de Unix(new_sock).Ver Fig. 4.8.

```
            case IN:
            case RD:
            case INP:
            case RDP:
            case 0:
                      c2++;
                    //printf("%d\n",cont);cont++;
                    if(xxx>15)
                    {  xxx=3;wborra(xxx,1,15,24,chw4);wrefresh(chw4);}
                    tipo_tupla(t.tipo);
             mvwprintw(chw4,xxx,1,"%d. %s %s
",c2,nomtupla,t.argumentos[0].valor.s);
                    wrefresh(chw4);
             xxx++;
                    send(new_sock,&t,sizeof(struct tupla),0);
            break;
```

Fig. 4.8 Tuplas recibidas del master(pasivas)

Cabe mencionar que el cliente de este socket (Unix) se crea desde la aplicación a través de la función *ini_app*, que se describirá mas adelante.

Si la tupla recibida fue un END, indica que ya no hay mas trabajo pendiente en el master por ejecutar y por lo tanto el worker puede finalizar su ejecución de forma normal, cerrando en forma adecuada todos los sockets creados durante su ejecución, y borrando del disco duro local los archivos creados durante la recepción de aplicaciones del usuario para evitar la acumulación de basura, además de finalizar el monitor de tuplas, ver Fig. 4.9

```
            case END:
              mvwprintw(chw5,2,2,"FINALIZANDO
WORKER- GRACIAS POR USAR LINDA-BUAP....");
                      wrefresh(chw5);
              close(create_socket);
              close(socket2);
              close(new_sock);
              close(orig_sock);
              unlink(NAME);
              strcpy(borra,"rm -f ");
              strcat(borra,nomarch2);
              system(borra);
              fin_monitor();
              exit(0);break;
           }//switch (t.tipo)
         }//if i==create_socket
```

Fig. 4.9 Tuplas recibidas del master (END)

Por otro lado si se detecta información en el socket de Unix (orig_sock) indica que hay una aplicación intentando conectarse con el worker para poder intercambiar datos, en este caso, orig_sock es el socket principal por donde se recibirán conexiones de las aplicaciones, una vez aceptada la conexión se crea el socket new_sock. Ver fig. 4.10

```
if (i==orig_sock)
        {
          if((new_sock=accept(orig_sock,(struct sockaddr*) &clnt_adr,&clnt_len))<0)
           { mvwprintw(chw5,2,2,"Error de accept");
             close(orig_sock);
             unlink(NAME);
             exit(3);}

            if(fdmax<new_sock) fdmax=new_sock;
                  FD_SET(new_sock,&worker);
}//if i=orig_sock
```

Fig. 4.10 Creación del socket de Unix

Una vez creado el socket new_sock se agrega al conjunto de descriptores para que también sea censado por el Select, ya que por este socket se recibirán tuplas de la aplicación.

Si se detecta información por el socket de Unix de la aplicación (new_sock), recibe una tupla de la aplicación, la cual puede ser una tupla pasiva (in, out, rd, rdp) o una tupla END. Si se trata de una tupla pasiva simplemente se envía al master para ser procesada. La tupla END indica que la aplicación ha finalizado, y entonces el worker crea y envía una tupla ACK para avisar al master que ha quedado libre y listo para recibir mas trabajo en caso de que aun existan tareas pendientes en el master por ejecutar, además cierra el socket de unix (new_sock) y lo elimina del conjunto de descriptores worker para que ya no sea tomado en cuenta para ser censado y actualiza la variable fdmax, ver Fig. 4.11

En cualquiera de los casos anteriores, una vez que se ha verificado por cual socket se esta recibiendo información y que tipo de información es, se continua evaluando al siguiente socket i, hasta legar a fdmax. Una vez evaluados todos los sockets contenidos en el conjunto de descriptores, se regresa al ciclo infinito para determinar si se continua trabajando o se finaliza la aplicación worker.

El worker permanecerá activo mientras que haya trabajo por ejecutar o hasta que el master le envíe una tupla indicando que ya no has mas trabajo, o bien si el master se desconecta, recibe una señal para finalizar su ejecución en cualquier momento, aún cuando se encuentre ejecutando trabajo.

```
if (i==new_sock)
        {       recv(new_sock,&t,sizeof(struct tupla),0);
                c3++;
                if(xx>15)
                {xx=3;wborra(xx,1,15,24,chw3);wrefresh(chw3);}
                tipo_tupla(t.tipo);
                if(t.tipo==END)
                mvwprintw(chw3,xx,1,"%d- %s %s ...",c3,nomtupla,t.argumentos[0].valor.s);
                else
                mvwprintw(chw3,xx,1,"%d- %s %s ",c3,nomtupla,t.argumentos[0].valor.s);
                wrefresh(chw3);
                xx++;
              if(t.tipo==END)
            { t.tipo=ACK;
              strcpy(t.argumentos[0].valor.s,"LISTO");
              strcpy(t.ip,rmt_host);
              close(new_sock);
              FD_CLR(i,&worker);
              if(fdmax==i) fdmax--;
             }//IF TIPO_END
            n=send(create_socket,&t,sizeof(struct tupla),0);
       }//if i=new_sock
```

Fig. 4.11 Tuplas recibidas de la aplicacion

4.3.2 Función recv_file.

Esta función se activa cuando se va a recibir el código ejecutable del master para ser ejecutado por el worker. Esta función crea un segundo socket TCP (socket2) para recibir por este el archivo, y espera en un ciclo infinito por conexión con el master, en caso de que éste no se encuentre listo, ver Fig. 4.12.

```
if ((socket2=socket(AF_INET,SOCK_STREAM,0))<=0) printf("
ERROR SOCKET2 NO CREADO\n");
  address2.sin_family=AF_INET;
  address2.sin_port=htons(1850);
  inet_pton(AF_INET,master,&address2.sin_addr);

  while(1)
   {
    if(connect(socket2,(struct sockaddr*)
&address2,sizeof(address2))==-1)
     {
       continue;
     }
    else
     {
       break;
     }
```

4.12 Función recv_file, creación del segundo socket.

Una vez establecida la conexión con el socket del master se inicia la recepción del archivo byte a byte y se crea en el disco duro local de la maquina worker, una vez finalizada la recepción se cierra el descriptor del archivo y el socket creado, ver Fig. 4.13.

```
if((fp=open(nomarch,O_CREAT|O_WRONLY|O_TRUNC,S_IRWXU))<0)
    {
     mvwprintw(chw5,2,2,"ERROR ABRIENDO ARCHIVO");
     exit(EXIT_FAILURE);
    }

    while(read(socket2,&c,1)) write(fp,&c,1);
    close(fp);
    close(socket2);
```

4.13 Recepción del Archivo ejecutable

4.3.3 Implementación de las funciones de coordinación.

Las funciones de coordinación están organizadas en dos librerias:

Linda1.h que se utiliza en la aplicación principal desarrollada por el usuario, ya que esta aplicación comunica tuplas a través de un socket TCP con el master.

Linda2.h que se utiliza en la aplicación ejecutable que se distribuirá en los trabajadores(workers), ya que estos se comunican con el worker utilizando un socket de Unix.

En el sentido estricto de funcionamiento, realizan exactamente las mismas acciones, la diferencia radica en por donde reciben y envían tuplas (socket TCP o UNIX), por lo tanto solo se explicarán una sola vez mostrando el código de una de ellas.

Función Out.

La función out se encarga de colocar una tupla en el espacio de tuplas. Cuando el usuario utilice esta función deberá indicar el tipo de dato de cada variable o valor que vaya a enviar por la tupla en una plantilla similar a la que utilizan funciones como scanf o printf (ver ejemplo Fig.4.14). Esta plantilla se analiza carácter por carácter para determinar el tipo de la variable o valor y almacenarlo en la estructura tupla, para enviarla al master, ver Fig. 4.15 y 4.16.

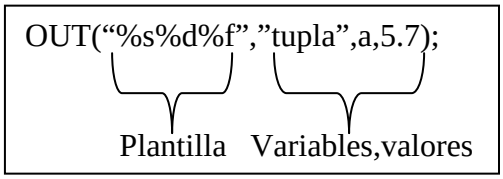


Fig. 4.14 Ejemplo de instrucción Out

Si el carácter evaluado en la plantilla es un **%** indica que es una variable de tipo simple, si el carácter siguiente es:
d: copia en el campo entero i de la estructura el valor de la variable pasada como argumento, se coloca el valor INT_TYPE en el campo tvar y el valor VALOR en el campo tacceso.

f: copia en el campo flotante f de la estructura tupla el valor de la variable pasada como argumento, se coloca el valor FLOAT_TYPE en el campo tvar y el valor VALOR en el campo tacceso.

c: copia en el campo caracter c de la estructura tupla el valor de la variable pasada como argumento, se coloca el valor CHAR_TYPE en el campo tvar y el valor VALOR en el campo tacceso.

s: copia en el campo cadena s de la estructura tupla la dirección de la cadena que se esta pasando como argumento, se coloca el valor STRING_TYPE en el campo tvar y el valor VALOR en el campo tacceso.

```
va_start(ap,mask);
  t.tipo=OUT;
  for(p=mask;*p;p+=2)
  { if(*p=='%')
     { switch(p[1])
       { case 's':
          strcpy(t.argumentos[k].valor.s,va_arg(ap,char *));
                t.argumentos[k].tvar=STRING_TYPE;
                t.argumentos[k].tacceso=VALOR;
                break;
        case 'd':
                t.argumentos[k].valor.i=va_arg(ap,long);
                t.argumentos[k].tvar=INT_TYPE;
                t.argumentos[k].tacceso=VALOR;
                break;
        case 'c':
              t.argumentos[k].valor.c=(char)va_arg(ap,int);
                t.argumentos[k].tvar=CHAR_TYPE;
                t.argumentos[k].tacceso=VALOR;
                break;
        case 'f':
                t.argumentos[k].valor.f=va_arg(ap,double);
                t.argumentos[k].tvar=FLOAT_TYPE;
                t.argumentos[k].tacceso=VALOR;
                break;
       }//SWITCH
     } // IF
```

Fig. 4.15 Función Out

Si el carácter de la plantilla es un **!** indica que se trata de un arreglo, si el carácter siguiente es:

d: se trata de el total de elementos del arreglo, y se almacena en el campo td de la estructura, se le coloca el tipo de dato TOT_ARR en el campo tvar y el acceso VALOR en el campo tacceso.

e: se trata de un arreglo de enteros, se recupera la dirección base del arreglo y se almacena cada elemento en la tupla en el campo ve, se le coloca el tipo de dato INT_ARR en el campo tvar y el acceso VALOR en el campo tacceso.

f: se trata de un arreglo de flotantes, se recupera la dirección base del arreglo y se almacena cada elemento en la tupla en el campo vf, se le coloca el tipo de dato FLOAT_ARR en el campo tvar y el acceso VALOR en el campo tacceso.

```
if(*p=='!')
     { switch(p[1])
       { case 'd':
                t.argumentos[k].valor.td=va_arg(ap,int);
                t.argumentos[k].tvar=TOT_ARR;
                t.argumentos[k].tacceso=VALOR;
                break;
         case 'e':
                i=va_arg(ap,int*);
                for(j=0;j<=t.argumentos[k-1].valor.td;j++)
                t.argumentos[k].valor.ve[j]=*(i+j);
                t.argumentos[k].tvar=INT_ARR;
                t.argumentos[k].tacceso=VALOR;
                break;
         case 'f':
                f=va_arg(ap,double*);
                for(j=0;j<=t.argumentos[k-1].valor.td;j++)
                {
                 t.argumentos[k].valor.vf[j]=*(f+j);
                }
                t.argumentos[k].tvar=FLOAT_ARR;
                t.argumentos[k].tacceso=VALOR;
                break;
       }//SWITCH
     }//if p==!
     k++;
  }//FOR
  va_end(ap);
  t.tot_dat=k;
n=send(create  socket,&t,sizeof(struct tupla),0);
```

Fig. 4.16 Función Out

Inicialmente se coloca en el campo tipo de la estructura tupla el valor OUT, para identificar el tipo de tupla.

El tipo de acceso VALOR, indica que se almacenarán los datos explícitos dentro de la estructura tupla, a excepción de las cadenas de caracteres.

Se va contabilizando argumento por argumento para tener el total de argumentos que contendrá la tupla y se almacena este dato en el campo tot_dat de la estructura tupla.

Una vez completado la formación de la tupla se envía al master por el socket principal (create_socket), ver Fig. 4.16.

Función In,Rd.

El funcionamiento de las funciones In y Rd son muy similares por eso se trataran en forma conjunta la diferencia entre estas dos instrucciones se da en el master. La primera borra una tupla del espacio de tuplas y la segunda solo toma una copia sin borrarla. Estas funciones forman un prototipo de tupla (*taux*), la cual se utiliza para buscar una tupla que iguale a este prototipo en el espacio de tuplas. Para formar este prototipo se analiza al igual que el Out carácter por carácter la plantilla proporcionada en el llamado a la función.

Cuando el carácter de la plantilla es un **%** se realizan exactamente las mismas acciones que en la función Out, solo que se utiliza la tupla prototipo *taux*, ver Fig. 4.15.

Cuando el carácter de la plantilla es una **?** se trata de valores por referencia, es decir, en la tupla *taux* se almacenan las direcciones de memoria en apuntadores a cada uno de los tipos de datos simples, en estas direcciones se almacenarán posteriormente los datos devueltos por el master en una tupla *t*. Ver Fig. 4.17.

Para cada uno de los tipos de datos, se almacenará su correspondiente en el campo tvar de la tupla *taux* (CHAR_TYPE, INT_TYPE, STRING_TYPE, FLOAT_TYPE).

Como puede observarse ahora el tipo de acceso almacenado en el campo tacceso es REFERENCIA, lo cual indica que los valores almacenados en la tupla son apuntadores (direcciones de memoria) a cada uno de los tipos de dato.

```
if(*p=='?')
    { switch(p[1])
      { case 's':
          taux.argumentos[j].valor.sp=va_arg(ap, char *);
          taux.argumentos[j].tvar=STRING_TYPE;
          taux.argumentos[j].tacceso=REFERENCIA;
          break;
        case 'd':
          taux.argumentos[j].valor.ip=va_arg(ap,long *);
          taux.argumentos[j].tvar=INT_TYPE;
          taux.argumentos[j].tacceso=REFERENCIA;
          break;
        case 'c':
          taux.argumentos[j].valor.cp=va_arg(ap,char *);
          taux.argumentos[j].tvar=CHAR_TYPE;
          taux.argumentos[j].tacceso=REFERENCIA;
          break;
        case 'f':
          taux.argumentos[j].valor.fp=va_arg(ap,double *);
          taux.argumentos[j].tvar=FLOAT_TYPE;
          taux.argumentos[j].tacceso=REFERENCIA;
          break;
      }//SWITCH
    }//IF  p=?
```

Fig. 4.17 Funciones In y Rd

Al igual que en la función Out, si el carácter de la plantilla es un **!**, indica que se trata de un arreglo de enteros o de flotantes, solo que en esta función se almacenara la dirección base del arreglo en un apuntador a enteros o flotantes según sea el caso y el tipo de acceso será REFERENCIA, ver Fig. 4.18

```
if(*p=='!')
    { switch(p[1])
      { case 'd':
            taux.argumentos[j].valor.tdp=va_arg(ap,int*);
            taux.argumentos[j].tvar=TOT_ARR;
            taux.argumentos[j].tacceso=REFERENCIA;
            break;
        case 'e':
            taux.argumentos[j].valor.ipv=va_arg(ap,int*);
            taux.argumentos[j].tvar=INT_ARR;
            taux.argumentos[j].tacceso=REFERENCIA;
            break;
        case 'f':
            taux.argumentos[j].valor.fp=va_arg(ap,double*);
            taux.argumentos[j].tvar=FLOAT_ARR;
            taux.argumentos[j].tacceso=REFERENCIA;
            break;
      }//SWITCH
```

Fig. 4.18 Funciones In y Rd

Una vez formada la tupla prototipo taux, se enviará por el socket principal (create_socket) al master y se esperará recibir una tupla t como respuesta del master.

Cuando se ha recibido la respuesta del master, para cada uno de los argumentos de la tupla prototipo taux, se verifica que su tipo de acceso sea REFERENCIA, de ser así se almacena el dato devuelto por el master en la tupla t en la dirección almacenada en la tupla taux.

En el caso de los vectores se recupera cada elemento del vector y se almacena en el espacio reservado y direccionado por la dirección base almacenada en la tupla taux. Ver Fig. 4.19.

```
n=send(create_socket,&taux,sizeof(struct tupla),0);
n=recv(create_socket,&t,sizeof(struct tupla),0);
  for(j=0; j<=t.tot_dat;j++)
  {  if(taux.argumentos[j].tacceso==REFERENCIA)
     switch(taux.argumentos[j].tvar)
     {
       case CHAR_TYPE:
          *taux.argumentos[j].valor.cp=t.argumentos[j].valor.c;
          break;
       case STRING_TYPE:
          saux=&t.argumentos[j].valor.s[0];
          taux.argumentos[j].valor.sp=saux;
          break;
       case INT_TYPE:
          *taux.argumentos[j].valor.ip=t.argumentos[j].valor.i;
          break;
       case FLOAT_TYPE:
          *taux.argumentos[j].valor.fp=t.argumentos[j].valor.f;
          break;
/********************* manejo de arreglos *********************/
       case INT_ARR:
           for(i=0;i<t.argumentos[j-1].valor.td;i++)
           {
           *(taux.argumentos[j].valor.ipv+i)=t.argumentos[j].valor.ve[i];
           }
            break;
       case FLOAT_ARR:
           for(i=0;i<t.argumentos[j-1].valor.td;i++)
          *(taux.argumentos[j].valor.fp+i)= t.argumentos[j].valor.vf[i];
            break;
       case TOT_ARR:
           *taux.argumentos[j].valor.tdp=t.argumentos[j].valor.td;
            break;


     }//SWITCH
  }//FOR
```

Fig. 4.19 Funciones In y Rd

Funciones Inp, Rdp.

Las funciones Inp y Rdp fueron implementadas igual que las dos anteriores (in y rd), la diferencia radica en la última parte, donde una vez recibida la respuesta del master si el tipo de la tupla t es 0 significa que la tupla no existe en el espacio de tuplas y un valor diferente de 0 significa que la tupla si existe en el espacio de tuplas.

Para obtener la tupla tendrían que utilizar un In o un Rd posterior a un Inp o Rdp. Como ya se ha comentado, estas instrucciones son una opción para censar por la existencia de una tupla en el espacio de tuplas sin bloquear al proceso que la solicita.

Función Eval.

Esta función se utiliza para enviar el código ejecutable del desarrollador al master. Primeramente, se crea una tupla activa tipo EVAL y se envía al master con el nombre del archivo ejecutable, esto es, para que el master se prepare para la recepción. Posteriormente si el archivo no se ha enviado ya, crea un socket TCP secundario llamado socket2 para enviar byte a byte el archivo al master, si el archivo ya fue enviado únicamente envía la tupla EVAL para avisar al master que aun hay trabajo por ejecutar con ese archivo. Una vez concluido el envío del archivo se cierra y destruye el socket2. Ver Fig. 4.20

```
t.tipo=EVAL;
strcpy(t.ip,rmt_host);
strcpy(t.argumentos[0].valor.s,nomarch);
send(create_socket,&t,sizeof(struct tupla),0);
if(nomarch != nomenviado)
{
if((socket2=socket(AF_INET,SOCK_STREAM,0))<=0)
printf("ERROR SOCKET2 CREADO\n");
address2.sin_family=AF_INET;
address2.sin_port=htons(1234);
inet_pton(AF_INET,master,&address2.sin_addr);

while(1)
{ if(connect(socket2,(struct
sockaddr*)&address2,sizeof(address2))==-1)
   continue;
  else break;
}

if((fp=open(nomarch,O_RDONLY))<0)
   {perror("abriendo archivo");exit(EXIT_FAILURE);}
 while(read(fp,&c,1)) write(socket2,&c,1); close(fp);
 nomenviado=nomarch;
 close(socket2);
```

Fig. 4.20 Función Eval

Funciones Init_linda, End_linda.

Estas funciones se utilizan al inicio y final de las aplicaciones principales desarrolladas por el usuario.

La función Init_linda se encarga de crear el socket principal (create_socket) por el cual el desarrollador se conectará con el master para enviar y recibir tuplas pasivas, ver Fig. 4.21

```
if ((create_socket = socket(AF_INET,SOCK_STREAM,0)) > 0)
   address.sin_family = AF_INET;
   address.sin_port = htons(8080);
   inet_pton(AF_INET,masterin,&address.sin_addr); /// segfault
   connect(create_socket,(struct sockaddr *)&address,sizeof(address));
```

Fig. 4.21 Función init_linda

La función end_linda se crea una tupla ENDA, la cual al ser enviada al master le indica que la aplicación ha finalizado y no necesitará mas de la memoria master, además se encarga de cerrar el socket principal (create_socket), ver Fig. 4.22

```
t.tipo=ENDA;
    strcpy(t.argumentos[0].valor.s,"APLIC");
    n=send(create_socket,&t,sizeof(struct tupla),0);
    close(create_socket);
```

Fig. 4.22 Función end_linda

Funciones Ini_app, fin_app.

Estas funciones se utilizan al inicio y al final del archivo ejecutable que se distribuye a los workers.

La función ini_app tiene como objetivo crear un socket cliente de Unix (orig_sock), el cual se conectará con el socket servidor creado en el worker. Como ya se ha mencionado, este socket sirve para enviar tuplas al worker desde la aplicación en ejecución y enviar tuplas hacia esta. Ver fig. 4.23.

La función fin_app se encarga de crear una tupla END que será enviada al worker para avisar que la ejecución de la aplicación ha finalizado y además cerrará el socket de Unix orig_sock, ver fig. 4.24

```
if((orig_sock=socket(AF_UNIX,SOCK_STREAM,0))<0)
{ perror("SOCKET");
  exit(1);
}//if(orig_sock)
serv_adr.sun_family=AF_UNIX;
strcpy(serv_adr.sun_path,NAME);
if(connect(orig_sock,(struct sockaddr *)
&serv_adr,sizeof(serv_adr.sun_family)+strlen(serv_adr.sun_path))<0)
{
perror("Connect");
exit(1);
}}
```

Fig. 4.23 Función ini_app

```
tt.tipo=END;
  strcpy(tt.argumentos[0].valor.s,"APLIC");
  write(orig_sock,&tt,sizeof(struct tupla));
  close(orig_sock);
```

Fig. 4.24 Función fin_app

4.4 Pruebas realizadas sobre el sistema Linda – BUAP.

Una forma de demostrar que el sistema Linda – BUAP funciona adecuadamente y dentro de los objetivos planteados al inicio de este proyecto de investigación, es presentar ejemplos que fueron programados utilizando otros sistemas similares como PLS Linda [1].

Hay que resaltar que estos ejemplos fueron codificados sin realizar modificaciones significativas con las propuestas existentes, lo cual demuestra que el diseño del sistema Linda – BUAP se apegó al diseño de sus predecesores en los cuales está basado.

En la mayoría de los ejemplos se presentan dos o mas versiones del mismo ejemplo con el fin de presentar alternativas en el diseño de programas paralelos y demostrar situaciones que pueden influir en el tiempo total de procesamiento, como lo es, un acceso constante a la memoria.

En cada ejemplo se da una descripción breve del mismo, se presentan las partes mas importantes de código y en los mas representativos se presentan comparaciones en función al tiempo de procesamiento con respecto a la versión secuencial y con diferente número de trabajadores.

El sistema Linda – BUAP presenta algunas restricciones, como lo es el manejo de arreglos bidimensionales y el número de elementos en los arreglos unidimensionales.

Un programa en Linda – BUAP esta dividido en dos partes: un programa principal que es el que se encarga de distribuir el trabajo entre los trabajadores (workers) y recopilar resultados para presentarlos al usuario y el código que se distribuirá entre los trabajadores en forma de un programa ejecutable.

En cada programa existe una constante llamada Nproc la cual establece el número de trabajadores con los que cuenta el sistema para ejecutar el trabajo.

Es necesario conocer la dirección IP del master y colocarla en la variable master para que el desarrollador sepa a donde tiene que enviar código y datos.

4.4.1 Proyecto 1: Hola Mundo.

Este programa tiene como objetivo mostrar como las tuplas pasivas se envían y se recuperan del espacio de tuplas. Se depositarán tantas tuplas pasivas como trabajadores existan y se le enviará a cada uno su archivo ejecutable, el cual se encargará de recuperar una tupla, incrementar la variable y volver a depositarla en el espacio de tuplas. A continuación se muestra el código del programa principal y del código ejecutable a distribuirse en los trabajadores (workers), Fig. 4.24 y 4.25.

```
#include <stdio.h>
#include <string.h>
#include "Linda1.h"
#define Nproc 200

main()
{ int i,proc; long tpi,tf,tt;
  char *master="10.34.6.6";
  init_linda(master);

  out("%s%d","cont",0);
  for(i=1; i<=Nproc; i++)
  eval("mundo");

  for(i=1;i<=Nproc;i++)
  { in("%s?d","hola",&proc);
    printf("Hola desde Worker %d\n",proc);
  }
 in("%s?d","cont",&proc);
 end_linda();
}
```

Fig. 4.24 Proy1.c (Programa Principal del usuario)

```
#include <stdio.h>
#include <stdlib.h>
#include "Linda2.h"

main()
{ int num;
  ini_app();
  in("%s?d","cont",&num);
  num++;
  out("%s%d","cont",num);
  out("%s%d","hola",num);
  fin_app();
}
```

Fig. 4.25 mundo.c (Archivo a distribuir a los workers)

La salida de este programa será el mensaje “Hola desde worker x”, donde x representará el número de worker que ejecuto esa tupla. Como en este ejemplo se utiliza como parámetro por referencia la variable proc al recuperar una tupla del espacio de tuplas (**in("%s?d","hola",&proc);**), recuperará la primera que encuentre, de esta forma se verá una salida en donde los mensajes están en desorden.

Para demostrar la diferencia entre recuperar en una tupla una variable pasada por referencia y una variable pasada por valor, se presenta una modificación de este proyecto, donde la instrucción anterior se presenta de la forma siguiente:

in("%s%d","hola",i);

Sí observamos, la variable proc ahora es sustituida por la variable i que controla al ciclo, y que no tiene el & antes, esto indica que la variable es pasada por valor, es decir se pasan los valores que va tomando i en las diferentes iteracciones, esto dará como resultado que se busque una tupla específica y por ende proporcione una salida en donde los mensajes se muestren ordenados.

4.4.2 Proyecto 2: Sumatoria de N números naturales.

A partir de este proyecto no se mostrará todo el código, solo lo más representativo, si desea analizar la aplicación completa, en el CD anexo a este documento se encentran los códigos fuente de todos los proyectos aquí descritos.

En este proyecto se presentan dos propuestas con el objetivo de demostrar que cuando se requiere demasiado acceso a la memoria el tiempo de procesamiento se ve afectado en forma negativa.

En el proyecto proy2.c se aplica un algoritmo en donde se envían tuplas con cuantos números sumará cada worker (veces), se envía el trabajo a cada worker (suma) y se envían tuplas con cada uno de los n números a sumar(dato).

```
printf("Total de numeros:");scanf("%d",&tot);
a=tot / Nproc;
resto=tot % Nproc;
for(i=1; i<Nproc; i++)
{ out("%s%d","veces",a); }
a+=resto;
out("%s%d","veces",a);
for(i=1; i<=Nproc; i++)    eval("suma");
for(i=1;i<=tot;i++)  { out("%s%d","dato",i);  }
```

Código del programa principal del usuario

Cada worker recuperará tantas tuplas como números les toque sumar sin importar cuales son esos números y deposita su tupla resultado (res).

```
in("%s?d","veces",&total);
suma=0;
for(i=1; i<=total; i++)
{ in("%s?d","dato",&a);
  suma+=a;}
out("%s%d","res",suma);
```

Código del programa enviado a los workers (suma.c)

Finalmente el desarrollador recupera cada una de las tuplas res para imprimir el resultado final de la suma.

```
for(i=1;i<=Nproc;i++)
  { in("%s?d","res",&r);
    total+=r;}
printf("Suma=%d\n",total);
```

Código del programa principal (proy2.c)

Una variante de este programa es el proy2_a.c el cual ahora determinará el intervalo de números que sumará cada worker, aquí se reduce significativamente el intercambio de información con la memoria ya que solo se enviarán tuplas “veces” para cada worker disponible, la cual contendrá el intervalo de la sumatoria.

```
a=tot / Nproc;
resto=tot % Nproc;
ini=1;
for(i=1; i<Nproc; i++)
{ out("%s%d%d","veces",a,ini);
  ini+=a;
  eval("suma2");  }
 a=a+resto;
 ini=(tot-a)+1;
 out("%s%d%d","veces",a,ini);
 eval("suma2");
```

Código del programa principal (proy2_a.c)

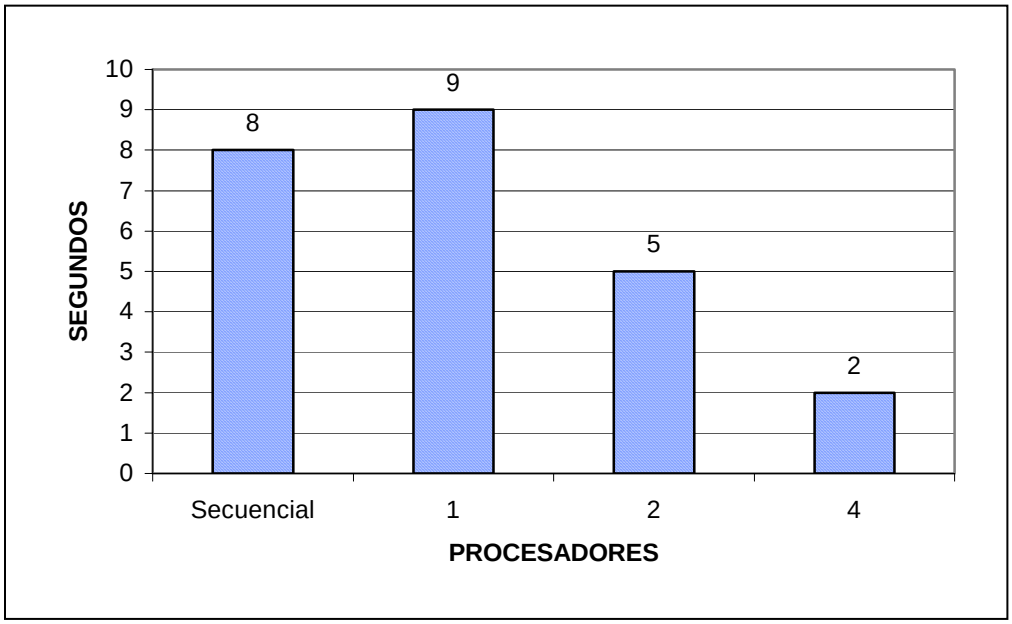
El worker, por su parte únicamente recupera una tupla veces, que contendrá de donde a donde le toca sumar y una vez realizada la suma deposita su resultado.

```
in("%s?d?d","veces",&total,&inicio);
suma=0;
for(i=inicio; i<(total+inicio); i++)
     suma+=i;
out("%s%d","res",suma);
```

Código del programa enviado a los workers (suma2.c)

En este proyecto se refleja una mejora considerable en tiempo de procesamiento de con respecto al proy2.c ya que el acceso a memoria es menor. Se realizo una comparación con una versión secuencial de este programa y con 1,2, y 4 trabajadores en el sistema Linda – BUAP, obteniendo la siguiente tabla de tiempos:

Tiempo de procesamiento Proy2_a.c con 100 000 000 datos			
Secuencial	***1 procesador***	***2 procesadores***	***4 procesadores***
8 segundos	10 segundos	5 segundos	3 segundos



4.26 Desempeño del Sistema Linda Buap para la sumatoria de 100000000 datos

Como puede observarse en la fig. 4.26 el tiempo se mejora notablemente desde los 2 procesadores, lo cual indica que la propuesta paralela de este algoritmo es mas satisfactoria que la versión secuencial en tiempo de procesamiento.

4.4.3 Proyecto 3: Suma de vectores de tamaño N.

Este algoritmo desarrolla la suma de los vectores A y B de números enteros de tamaño N, enviando al espacio de tuplas el i-ésimo elemento de A y el i-ésimo elemento de B para obtener el i-ésimo elemento de C y envía una tupla eval por cada par de datos.

```
void suma_vector(int x[],int y[], int z[])
{ int i;
 for(i=0;i<N;i++)
 { out("%s%d%d%d","operandos",x[i],y[i],i);
 eval("suma_vec"); }
```

Código del programa principal (proy3.c)

Cada worker se encargará de recuperar sus operandos, realizar la suma y depositar el resultado.

```
in("%s?d?d?d","operandos",&a,&b,&i);
c=a+b;
out("%s%d%d","suma",c,i);
```

Código del programa enviado a los workers (suma_vec.c)

Posteriormente el desarrollador se encarga de ir recuperando cada resultado y almacenarlo en la i-ésima posición del vector C.

```
for(i=0;i<N;i++)
  in("%s?d%d","suma",&z[i],i);
```

Código del programa principal (proy3.c)

Esta propuesta es poco eficiente para este sistema ya que si hablamos de arreglos con un N muy grande el acceso a memoria de cada elemento individual hace que se sacrifique el tiempo de procesamiento.

Se presenta otra versión de este algoritmo en donde cada worker recupera tantos términos como existan aún en el espacio de tuplas, para esto utiliza una bandera llamada “sumados”, la cual cada vez que se calcula un nuevo término se incrementa y se deposita en el espacio de tuplas. Los workers terminan una vez que no haya más elementos que sumar en el espacio de tuplas. En esta versión se envían tantas tuplas eval como workers existan en el sistema.

```
in("%s?d?d","sumados",&cont,&n);
while(cont<n)
 { cont++;
 out("%s%d%d","sumados",cont,n);
 in("%s?d?d?d","operandos",&a,&b,&indice);
 c=a+b;
 out("%s%d%d","suma",c,indice);
in("%s?d?d","sumados",&cont,&n); }
```

Código del programa enviado a los workers (sum_vec.c)

En esta versión se busca disminuir el acceso a la memoria, pero para vectores de tamaño muy grande puede no representar la mejor solución.

4.4.4 Proyecto 4: Suma de matrices de tamaño N*N.

Se presenta la suma de dos matrices X e Y de tamaño n*n, enviando al espacio de tuplas la i-ésima fila de X y la i-ésima fila de Y para obtener la i-ésima fila de Z.

En este proyecto se muesta como el sistema Linda – BUAP maneja arreglos para ser enviados al espacio de tuplas.

El algoritmo inicia con el envío de las filas de cada matriz al espacio de tuplas utilizando la tupla “filas”, en esta tupla se envía la i-esima fila de X y la i-esima fila de Y. Se envían tantas tuplas eval como workers existan en el sistema.

```
for(i=0;i<N;i++)
{out("%s!d!e!d!e%d","filas",tam,&x[i][0],tam,&y[i][0],i); }
  out("%s%d%d","sumados",0,tam);
 for(i=1;i<=Nproc;i++)
 eval("sum_mat");
```

Código del programa principal (proy4.c)

Cada worker recupera tuplas filas, para ejecutar la suma de la fila indicada por la variable índice, depositando el resultado de cada fila en la tupla resultados, hasta que la bandera cont, recuperada en la tupla sumados indique que ya no hay mas filas que sumar en el espacio de tuplas.

```
in("%s?d?d","sumados",&cont,&n);
while(cont<n)
{  cont++;
   out("%s%d%d","sumados",cont,n);
    in("%s!d!e!d!e?d","filas",&n,&a,&n,&b,&indice);
   for(i=0;i<n;i++)  c[i]=a[i]+b[i];
   out("%s!d!e%d","resultados",n,&c,indice);
    in("%s?d?d","sumados",&cont,&n); }
```

Código del programa enviado a los workers (sum_mat.c)

Por último el desarrollador recupera los resultados de cada una de las filas y los almacena en la matriz Z.

```
for(i=0;i<N;i++)
 {   in("%s!d!e%d","resultados",&tam,&z[i][0],i); }
```

Código del programa principal (proy4.c)

Aun cuando es mas eficiente que la suma de vectores, el sistema Linda –BUAP se encuentra limitado por el tamaño de los arreglos que se pueden enviar por las tuplas (30 elementos como máximo).

4.4.5 Proyecto 5: Multiplicación de matrices de tamaño N*N.

Programa que realiza la multiplicacion de dos matrices de n*n mediante el envío del i-esimo renglon de A y la j-esima columna de B para obtener el elemento ij-esimo de C (proyecto 5).

Inicialmente el desarrollador deposita las i-esima fila de X y la j-ésima columna de Y y la fila y la columna a la que corresponden i y j en la tupla “datos” y envía un eval con el archivo mult_mat por cada worker disponible en el sistema.

```
for(i=0;i<N;i++)
   for(j=0;j<N;j++)
out("%s!d!e!d!e%d%d","dato",tam,&x[i][0],tam,&y[j][0],i,j);
/*bandera para saber cuantos ha multiplicado  cada worker*/
   out("%s%d%d","mult",0,tam);
  for(i=1;i<=Nproc;i++)  eval("mult_mat");
```

Código del programa principal (proy5.c)

Cada worker recupera su tupla “mult” para saber si aún existen filas por multiplicar y va recuperando tuplas “dato” para obtener un elemento de la matriz C y depositarlo en la tupla “resul”.

```
in("%s?d?d","mult",&cont,&tam);
  while(cont<tam*tam)
  {  cont++;
     out("%s%d%d","mult",cont,tam);

     in("%s!d!e!d!e?d?d","dato",&tam,&a,&tam,&b,&i,&j);
    valor=0;
    for(k=0;k<tam;k++)
   {       valor+=a[k]*b[k];  }
   out("%s%d%d%d","resul",valor,i,j);
   in("%s?d?d","mult",&cont,&tam); }
```

Código del programa enviado a los workers (mult_mat.c)

Por último el desarrollador recupera cada elemento calculado por los workers los almacena en la matriz Z.

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
    {
      in("%s?d%d%d","resul",&r,i,j);
      z[i][j]=r;    }
```

Código del programa principal (proy5.c)

Este programa presenta la limitante de que para arreglos muy grandes el tiempo de procesamiento se puede volver un inconveniente.

Para mejorar el tiempo de procesamiento se propone una variante, la multiplicación de matrices por el método secuencial pero ahora enviando la fila i-ésima de A y toda la matriz B, para calcular la fila i-esima de C (proyecto 5_a).

Este ejemplo también sirve para mostrar el uso de la instrucción RD, la cual no borra las tuplas del espacio de tuplas. Con esto, la matriz B solo se envié una vez y las tuplas que forman la matriz permanecen en el espacio de tuplas, para cada worker que necesite de la matriz B.

Primeramente se envía la matriz A y la matriz B al espacio de tuplas en las tuplas “filaA” y “filaB”, posteriormente se obtiene cuantas filas de la matriz C calculará cada worker, y se deposita en la tupla “trabajo” y se envía un eval con “mult_mat2” para cada worker disponible.

```
for(i=0;i<N;i++)
  {out("%s!d!e%d","filaB",tam,&b[i][0],i);
    out("%s!d!e%d","filaA",tam,&a[i][0],i);  }
 trab=tam/Nproc;
 resto=tam%Nproc;
 for(i=1;i<=Nproc;i++) out("%s%d","trabajo",trab);
 trab+=resto;
 out("%s%d","trabajo",trab);
  for(i=1;i<=Nproc;i++) eval("mult_mat2");
```

Código del programa principal (proy5_a.c)

Cada worker recupera su tupla “trabajo” para saber cuantas filas calculará, después recupera una copia de la matriz B utilizando RD. Recupera una fila de A con su “indice” y calcula la fila “indice” de C, depositando el resultado en la tupla “filaC”.

```
in("%s?d","trabajo",&trab);
for(k=0;k<N;k++)
rd("%s!d!e%d","filaB",&tam,&b[k][0],k);

for(k=1;k<=trab;k++)
{ in("%s!d!e?d","filaA",&tam,&a,&indice);

   for(i=0;i<N;i++)
   { suma=0;
     for(j=0;j<N;j++)  suma+=a[j]*b[i][j];
     c[i]=suma;}
  out("%s!d!e%d","filaC",tam,&c,indice);}
```

Código del programa enviado a los workers (mult_mat2.c)

Por último el desarrollador recupera cada fila de C.

```
for(i=0;i<N;i++)  in("%s!d!e%d","filaC",&tam,&c[i][0],i);
```

Código del programa principal (proy5_a.c)

En este proyecto se ve una mejora considerable en el tiempo de procesamiento en el sistema con respecto a la primera versión ya que se reduce el acceso a la memoria.

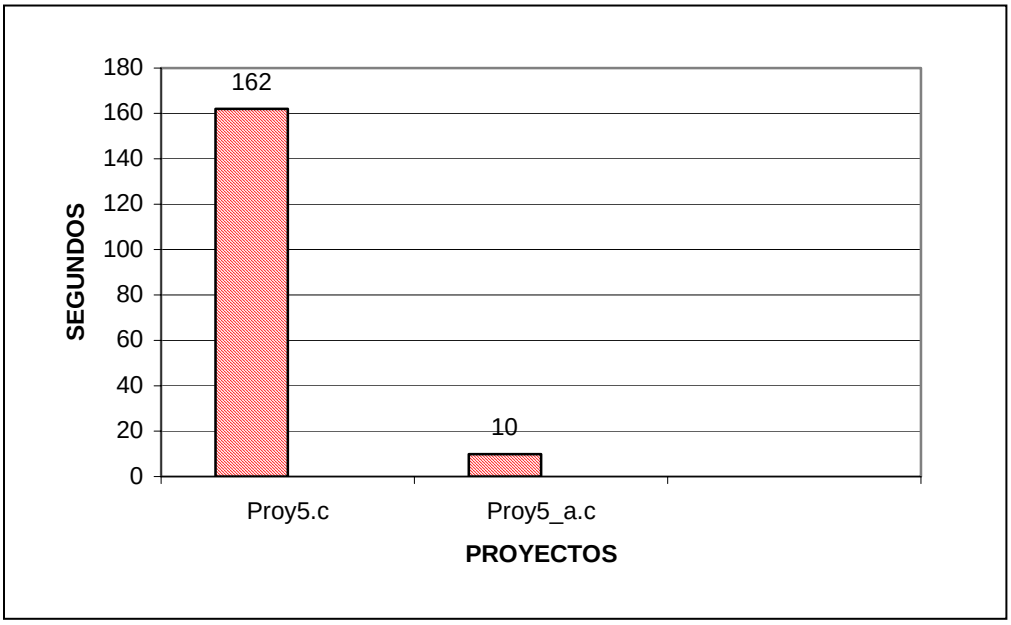


Fig. 4.27 Comparación de desempeño entre Proyecto 5 y 5_a .

Obsérvese en la fig. 4.27 como para la multiplicación de dos matrices de 30 X 30 elementos el tiempo se ve sumamente mejorado en el proyecto 5_a,

4.4.6 Proyecto 6: Inversa de una matriz.

Este proyecto deberá de encontrar la inversa multiplicativa de una matriz de *N* x *N,* siguiendo el siguiente procedimiento:

Sea **A** una matriz de *N x N*. si existe una matriz **B** de *N x N* tal que

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I},$$

en donde **I** es la identidad multiplicativa, entonces se dice que **B** es la **inversa multiplicativa de A** y se la denota por $\mathbf{B} = \mathbf{A}^{-1}$.

Sea **A** una matriz no singular (det **A** diferente de 0) de *N x N* y sea $\mathbf{A}_{ij} = (-1)^{i+j}\mathbf{M}_{ij}$, en donde $\mathbf{M}_{ij}$ es el determinante de *(N-1) x (N-1)* obtenido suprimiendo la fila número *i* y la columna número *j* de det **A**. Entonces

$$\mathbf{A}^{-1} = (1/\det \mathbf{A})\ (\mathbf{A}_{ij})^{T}.$$

Cada $\mathbf{A}_{ij}$ es simplemente el cofactor del correspondiente elemento a_{ij} en det **A**. Nótese que en la fórmula se usa la transpuesta de la matriz **A.**

Primeramente se obtienen las N submatrices de A que servirán para calcular los cofactores de la fila uno de A y enviarlas al espacio de tuplas en la tupla “bloque” junto con sus índices correspondientes.

```
void reparte_primer_fila(double x[N][N])
{int i,j,k,l,r,s,tam;double bloque[N][N];
 tam=N;
 for (j=0;j<N;j++)
{ r=0;
 for (k=0;k<N;k++)
 { if(k!=0)
   { s=0;
     for (l=0;l<N;l++)
     if(l!=j)
     { bloque[r][s]=x[k][l]; s++; }
     r++; }//if k!=0
 }//for k
for(i=0;i<N-1;i++)
{out("%s!d!f%d%d%d","bloque",tam-1,&bloque[i][0],0,j,i);
}
```

Código del programa principal (proy6_a.c)

Cada worker se encargará de calcular el determinante de cada una de las submatrices enviadas anteriormente al espacio de tuplas por medio del método de Gauss y coloca en el espacio de tuplas su resultado.

```
rd("%s?d","tamanio",&dim);
in("%s?d","nummat",&num);
for(k=0;k<tam;k++)
  in("%s!d!f?d%d%d","bloque",&tam,&a[k][0],&r,num,k);
s=num;
num++;
if (num<N)  out("%s%d","nummat",num);
factor2=1;
salida=0;
for(i=0;i<dim-1;i++)
  { p=i;
    for(k=i;k<dim;k++)
    {
     if(a[k][i]!=0)
      {
        if(a[k][i]>0)
         { if(a[p][i] < a[k][i]) p=k;
         }
      }
     else
      if(a[p][i] < - (a[k][i])) p=k;
    }//for k
    if(a[p][i]==0)
     {factor2=0;salida=1;
     }//if
```

Código del programa enviado a los workers (gauss.c)

```
      if(!salida)
  { if(p!=i)
  {factor2++;
 for(k=0;k<dim;k++)
{rentemp[k]=a[i][k];
        a[i][k]=a[p][k];
        a[p][k]=rentemp[k];
       }  //for k
      }//if p!= i
      for(j=i+1;j<dim;j++)
      {factor1=a[j][i]/a[i][i];
       for(k=0;k<dim;k++)
       a[j][k]-=factor1*a[i][k];
      }//for j
    } //if !salida
  }//for i
j=dim-1;
if(a[j][j]==0) det=0;
else
 {det=a[0][0];
  for(i=1;i<dim;i++)
   det*=a[i][i];
  if((factor2 % 2)==0) det*=-1;
  if(((r+s)%2)==1) det*=-1;
 }//else
out("%s%f%d%d" "detp" det r s);
```

Continúa código del programa enviado a los workers (gauss.c)

El desarrollador obtendrá del espacio cada uno de los *N* resultados parciales para guardarlos de acuerdo a sus índices correspondientes en B (donde B será la matriz inversa de A) y de ellos mismos se servirá para calcular el determinante global de A.

Una vez que se tiene el valor del determinante y si éste es distinto de cero, ahora toca turno al cálculo del resto de la inversa de A. Lo que se tiene que hacer es determinar el resto de los cofactores correspondientes a los elementos de A que no están en la primer fila. Por ello, en primer lugar hay que obtener las submatrices correspondientes mediante el siguiente procedimiento en el desarrollador:

```
void reparte_resto(double x[N][N])
{int ii,i,j,k,l,r,s;double bloque[N][N];int tam=N;

 for(i=1;i<N;i++)
{
  for(j=0;j<N;j++)
  {r=0;
   for(k=0;k<N;k++)
   {
    if(k!=i)
    { s=0;
```

Código del programa principal (proy6_a.c)

```
for(l=0;l<N;l++)
{if(l!=j)
 { bloque[r][s]=x[k][l]; s++; }//if
  }
  r++;}//if
   }//for k

for(ii=0;ii<N-1;ii++)
{ out("%s!d!f%d%d%d","bloque",tam-
1,&bloque[ii][0],i,j,ii);
}
  }//for j
 }//for i
}
```

Continúa código del programa principal (proy6_a.c)

Los workers deberán calcular el determinante de cada submatriz colocada en el espacio de tuplas con el mismo procedimiento “gauss”. Por último se deberán recoger todos los resultados producidos por los workers y almacenarlos en la matriz B, obtener su traspuesta y dividir cada elemento de B entre la determinante global de A, para con esto obtener la Inversa de A.

```
void obten_inversa(double b[N][N],double determ)
{ int i,j,tam;double vartemp,temp;
  for(i=1;i<N;i++)
  { out("%s%d","nummat",0);
    for(j=1;j<=N;j++) eval("gauss");
   }
  for(i=1;i<N;i++)
   for(j=0;j<N;j++)
   {
     in("%s?f%d%d","detp",&vartemp,i,j);
     b[j][i]=vartemp;

   }

  for(i=0;i<N;i++)
   for(j=0;j<N;j++)
     b[i][j]=b[i][j]/determ;

 in("%s?d","tamanio",&tam);

}
```

Continúa código del programa principal (proy6_a.c)

4.4.7 Proyecto 7: Integral de una función.

Este proyecto se encarga de calcular la integral de la función $f(x) = X^2$ dentro de un intervalo dado por el usuario($\int_a^b x^2$). La integral se calcula particionando el intervalo de integración, dado por a y b y formando rectángulos con base (b-a) /n y altura f(x) y calculando el área de cada uno de estos rectángulos. Las áreas se suman para dar como resultado el área bajo la curva.

Los valores del intervalo (a y b) y el número de particiones (tampart) seran proporcionados por el usuario.

El desarrollador primeramente determina los subintervalos para que cada trabajador calcule un número determinado de áreas. Este subintervalo se coloca en el espacio de tuplas en la tupla “rango”. Envía trabajo a cada uno de los workers con la función “integralp”.

```
tam=(long)tampart/Nproc;
extra=tampart%Nproc;
dx=(b-a)/tampart;
fin=0;
for(i=1; i<=Nproc;i++)
{ ini=fin;
  fin=fin+tam;
  if(extra!=0 && i==Nproc) fin+=extra;
  out("%s%d%d%f%f","rango",ini,fin,dx,a);}//for i
for(i=1; i<=Nproc;i++) eval("integralp");
```

Código del programa principal (proy7.c)

Cada worker se encargará de calcular el subrango de áreas recuperado de la tupla “rango” y colocará su resultado parcial en el espacio de tuplas con la tupla “area”.

```
in("%s?d?d?f?f","rango",&ini,&fin,&dx,&a);
x=ini*dx+a;
areap=0;
for(i=ini;i<fin;i++)
{fdex=x*x;
 areap+=dx*fdex;
 x+=dx;}//for i
out("%s%f","area",areap);
```

Código del programa enviado a los workers (integralp.c)

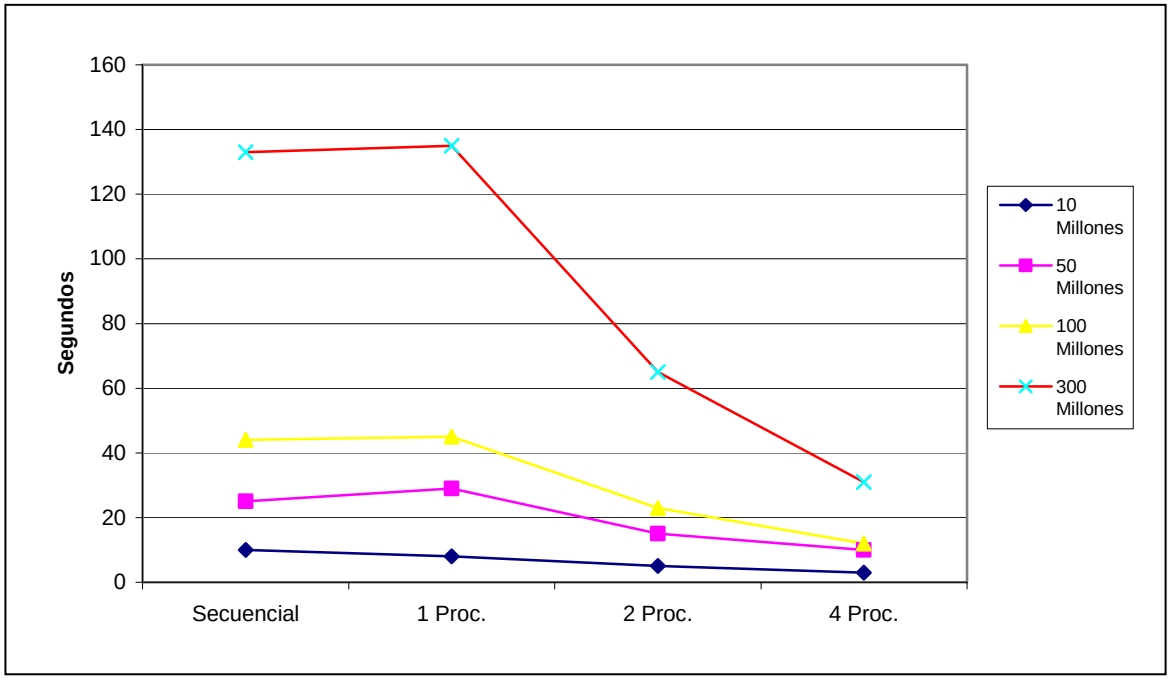
Finalmente el desarrollador se encarga de recuperar cada resultado parcial y acumularlo para obtener el resultado final.

```
for(i=1; i<=Nproc;i++)
  { in("%s?f","area",&areap);
      areatot+=areap; }
```

Continúa código del programa principal (proy7.c)

Este problema es muy representativo para demostrar la efieciencia de Linda – BUAP en problemas donde se requiere poca comunicación a la memoria y gran cantidad de cálculos en los workers.

Se hicieron pruebas con una versión secuencial de este problema y se comparó con el sistema Linda – BUAP para 2 y 4 trabajadores con el mismo número de particiones (en millones de datos) obteniendo los siguientes resultados.



Integral de f(x) por el método de la suma de areas de rectángulos (proy7)				
	10 Millones	50 Millones	100 Millones	300 Millones
Secuencial	10	25	44	133
1 proc.	8	29	45	135
2 proc.	5	15	23	65
4 proc.	3	10	12	31

Fig. 4.28 Comparativa de Tiempos de ejecución del proyecto 7

Como puede observarse en la fig. 4.28 el tiempo de procesamiento del algoritmo secuencial se reduce casi en un 48% cuando se ejecuta en el sistema Linda – BUAP con 2 workers (trabajadores conectados) y conforme se aumenta el número de trabajadores se va reduciendo significativamente. Hay que considerar que debido a la latencia de la transmisión por la red, se llegará a un número de workers en donde ya no se observe mejora del tiempo de procesamiento.

4.4.8 Proyecto 8: Multiplicación de Matrices por el método de Canon.

Este algoritmo y el siguiente son los más apropiados para sistemas de paso de mensajes y la arquitectura de paso de mensajes que mas se ajusta a las matrices es una malla.

Este algoritmo utiliza una malla de procesadores con conexiones entre los elementos de cada lado (toro), para desplazar los elementos de A hacia la izquierda y los de B hacia arriba.

Si se enumeran los procesadores de acuerdo a su posición en la malla, el procesador *P(i,j)* contiene los elementos *(i,j),* de las matrices A, B y C. A partir de esta distribución de datos, se reubican los datos de las matrices A y B en la malla, de forma que si se tiene una malla de P x P procesadores, el procesador *P(i,j)* tendrá asignado el elemento *A(i,(j+i) mod P)* y también el elemento *B((i+j)mod P,j)* para calcular el elemento *C(i,j)*.

A partir de la reubicación inicial, se realizan iterativamente los pasos de:
- Multiplicación local de los datos asignados en cada procesador para el cálculo de un resultado parcial
- Rotación a la izquierda de los elementos o submatrices de A
- Rotación hacia arriba de los elementos o submatrices de B

Y después de P de estos pasos se tienen los valores de la matriz C totalmente calculados.

En el siguiente fragmento de código se muestra la distribución inicial de los elementos de las matrices enviado cada elemento de la matriz A en la tupla “PijA” y los indices de su posición (i,j), y los elementos de la matriz B en la tupla “PijB” y los índices de su posición (i,j). Se envia también la tupla eval con el archivo “single_mult2” para que cada worker calcule un elemento parcial de C.

```
void dist_init(int A[Q][Q],int B[Q][Q])
{
 int i,j,r;
 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  {
out("%s%d%d%d%d%d","PijA",A[i][(j+i)%P],i,(j+i)%P,i,j);

out("%s%d%d%d%d%d","PijB",B[(j+i)%P][j],(j+i)%P,j,i,j);
eval("single_mult2");}}
```

Código del programa principal (proy8.c)

El archivo single_mult2.c contiene el código siguiente, el cual calculará la multiplicación de los elementos recuperados de el espacio de tuplas, los multiplica y coloca el resultado en la tupla “parcial”, la cual contendrá el resultado de la multiplicación y los índices (i,j) del elemento de C al que corresponde este resultado.

```
in("%s?d?d?d?d?d","PijA",&a,&pia,&pja,&i,&j);
in("%s?d?d?d%d%d","PijB",&b,&pib,&pjb,i,j);
c=a*b;
out("%s%d%d%d","parcial",c,i,j);
```

Código del programa enviado a los workers (single_mult2.c)

La función matriz_mult realiza las rotaciones siguientes y reenvía los elementos rotados al espacio de tuplas en las tuplas PijAM y PijBM y reenvía trabajo a los workers y recupera cada resultado parcial y lo acumula hasta completar las P iteracciones.

```
void matriz_mult(int A[Q][Q],int B[Q][Q])
{ int i,j,k,c,pia,pja,pib,pjb,numA,numB,r,s;

 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
   C[i][j]=0;
/**************** recupera primeros resultados parciales
**************/
 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  { in("%s?d%d%d","parcial",&c,i,j);
   C[i][j]=C[i][j]+c;
  }
/********** vuelve a distribuir recorriendo elementos **********/
for(k=1;k<P;k++)
{
 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  { out("%s%d%d%d%d%d","PijAM",A[i][(j+i)%P],i,(j+i)%P,i,j);
    out("%s%d%d%d%d%d","PijBM",B[(j+i)%P][j],(j+i)%P,j,i,j);
  }

 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
   {
    if(j+k>=P) in("%s?d?d?d%d%d","PijAM",&numA,&pia,&pja,i,j+k-P);
    else  in("%s?d?d?d%d%d","PijAM",&numA,&pia,&pja,i,j+k);
    out("%s%d%d%d%d%d","PijA",numA,pia,pja,i,j);
    if(i+k>=P) in("%s?d?d?d%d%d","PijBM",&numB,&pib,&pjb,i+k-P,j);
    else  in("%s?d?d?d%d%d","PijBM",&numB,&pib,&pjb,i+k,j);
    out("%s%d%d%d%d%d","PijB",numB,pib,pjb,i,j);
    eval("single_mult2");
 }//forj
 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  {in("%s?d%d%d","parcial",&c,i,j);
   C[i][j]=C[i][j]+c;
  }
 }//for k
```

Continúa código del programa principal (proy8.c)

4.4.9 Proyecto 9: Multiplicación de matrices por el algoritmo de Fox.

Su funcionamiento es muy similar al de canon en cuestión de la distribución de los elementos de las matrices A y B en una malla toro con los extremos interconectados entre sí, solo que en este caso no se define ningún paso inicial de reubicación de los datos. El algoritmo se define iterativamente de la siguiente forma:
En la iteración o paso K:

- El dato de la matriz A almacenado en el procesador de la posición i,i+k mod P se envía a todos los procesadores de la fila i de la malla.
- Se multiplican en cada procesador los datos de A recibidos con los datos de B almacenados localmente, obteniendo un resultado parcial de la matriz C.
- Se rotaran los datos de B hacia arriba tal como en el algoritmo de Canon.

En cada iteración realiza la rotación de los elementos de B y realiza el envío de todos los elementos de A en la fila i(su posición) y envía la tupla eval con el archivo fox para que los trabajadores realicen sus cálculos.

Código del programa principal (proy9.c)

```
void matriz_mult(int A[Q][Q],int B[Q][Q])
{ int i,j,k,c,pia,pja,pib,pjb,numA,numB,r,s;

 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
      C[i][j]=0;

for(k=0;k<P;k++)
{
 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
   out("%s%d%d%d%d%d","PijBM",B[i][j],i,j,i,j);

 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  { if(i+k>=P) in("%s?d?d?d%d%d","PijBM",&numB,&pib,&pjb,i+k-
P,j);
    else  in("%s?d?d?d%d%d","PijBM",&numB,&pib,&pjb,i+k,j);
    out("%s%d%d%d%d%d","PijB",numB,pib,pjb,i,j);
    out("%s%d%d%d%d%d","PijA",A[i][(k+i)%P],i,(i+k)%P,i,j);
    eval("fox");
  }

 for(i=0;i<P;i++)
  for(j=0;j<P;j++)
  {in("%s?d%d%d","parcial",&c,i,j);
   C[i][j]=C[i][j]+c;
  }
}//for k
}
```

La function fox.c que se ejecuta en cada worker, recupera los elementos de la matriz A y B que le correspondenl, realiza su calculo y coloca en el espacio de tuplas el resultado parcial de C. El desarrollador recupera sus resultados parciales y los acumula par que al completar las P iteraciones se obtenga la multiplicación de las matrices.

```
in("%s?d?d?d?d?d","PijA",&a,&pia,&pja,&i,&j);
in("%s?d?d?d%d%d","PijB",&b,&pib,&pjb,i,j);
c=a*b;
out("%s%d%d%d","parcial",c,i,j);
```

Código del programa enviado a los workers (fox.c)

Conclusiones

En este trabajo se desarrolló un sistema de procesamiento paralelo llamado Linda-BUAP. Dicho sistema es una combinación hardware–software, donde el hardware consta de un cluster de computadoras personales con arquitectura MIMD con una memoria virtual compartida, y el software es un worker, quién representa a una unidad de procesamiento, el cuál se encargara de ejecutar aplicaciones de usuario. El worker funciona como un cliente, el cuál al conectarse con el master (memoria) solicita y recibe el código que se ejecutará y representa una aplicación del usuario y datos de la memoria. A su vez, al estar interactuando con la aplicación se comportará como un servidor de datos para esta aplicación.

El sistema Linda – BUAP se presenta como una opción para resolver problemas que requieren de una gran cantidad de operaciones aritméticas y lógicas y poca comunicación con la memoria.

Se realizaron varios proyectos sobre este sistema, lo cual demuestra que:

- Se cuenta con un sistema estable para el desarrollo de aplicaciones en una máquina paralela a bajo costo, para uso didáctico y de investigación.

- Los tiempos obtenidos en las pruebas fueron los esperados ya que se cuenta con resultados obtenidos de los mismos problemas en otros sistemas similares.

- Los programas desarrollados como parte de las pruebas, fueron tomados de proyectos desarrollados en PLS Linda , y al implementarlos en Linda – BUAP no se les hizo ninguna modificación lo cual demuestra que es un Linda puro.

- Se cuenta con el código fuente para el desarrollo de futuras aplicaciones distribuidas o paralelas.

- Es un sistema que en su implementación y uso no implica ninguna inversión económica, ya que se utilizaron máquinas que se pueden tener incluso en casa, conectadas a través de una red LAN, utilizando el sistema operativo Linux y el compilador del lenguaje ANSI C que viene incluido con la distribución de Linux.

El sistema cumple con el objetivo planteado, ya que es un sistema paralelo enfocado a la educación y a la investigación, de bajo costo y con tiempos de respuesta adecuados.

Este sistema sienta las bases para una gran cantidad de proyectos en el área del paralelismo ya que a partir de él se pueden construir diversas arquitecturas paralelas como sistemas de memoria distribuida, sistema de memoria compartida

distribuida, sistemas con tolerancia a fallas, servidores de memoria, sistemas operativos paralelos, etc.

Aportaciones.

El sistema Linda – BUAP, fue inspirado en el sistema PLS Linda. Al hacer un análisis de éste modelo se observó que podían realizarse ciertas mejoras para hacer más eficiente el sistema. Las aportaciones a este sistema se enumeran a continuación:

- En Linda – BUAP no se utiliza un preprocesador para traducir instrucciones Linda-BUAP a C, en su lugar las instrucciones están organizadas en librerías *.h*, las cuales solo se incluyen en las aplicaciones para poder utilizar de forma natural las instrucciones Linda – BUAP y compilar de forma normal la aplicación.

- En Linda – BUAP los trabajadores (workers) finalizan de forma normal después de cada ejecución. En el PLS Linda, los trabajadores quedaban bloqueados después de la ejecución de una aplicación, así se tenía que reiniciar la computadora y volver a ejecutar la aplicación worker cada vez, lo cual representaba algo ineficiente, en Linda – BUAP los trabajadores finalizan aún y cuando el master salga de ejecución.

- En Linda – BUAP los trabajadores (workers) soportan la reasignación de trabajo, es decir, si un trabajador termina, y si aún existiera trabajo pendiente en el master se le reasigna tarea hasta concluir la aplicación, en el modelo Linda, se tenia que definir el trabajo para el número exacto de trabajadores (workers) reales en el sistema.

- Linda – BUAP permite un mejor monitero de las instrucciones que se ejecutan los trabajadores, en PLS Linda el monitoreo era escueto y no se entendía claramente cual era el significado de la información presentada.

- En Linda – BUAP se pueden utilizar funciones de la biblioteca de C en las funciones que se ejecutan en los trabajadores (archivos ejecutables), en PLS Linda no se podían utilizar este tipo de funciones porque no eran reconocidas fuera del entorno del desarrollador, ya que era una función dependiente del programa principal, y aquí se toma ventaja de que es un archivo ejecutable independiente hasta cierto punto.

- En Linda – BUAP se pueden enviar mensajes a la salida estándar en las funciones que ejecutan los trabajadores, aunque no es recomendable pero puede usarse como una herramienta para poder brindar una herramienta de

depuración de las aplicaciones que se ejecutan en los trabajadores, cosa que en PLS Linda no se podía realizar.

- En Linda – BUAP se realiza la compilación separada de la aplicación principal y la función que se enviará a los workers, dando con esto una visión al usuario de que es exactamente lo que los workers recibirán para su ejecución.

Perspectivas.

Como todo sistema en su primera versión, éste es perfectible, el Sistema Linda – BUAP presenta algunas restricciones, por ejemplo el manejo de arreglos se encuentra limitado por el tamaño de la estructura que se envía a través de los sockets, ya que se intento establecer arreglos con una longitud de 255 elementos dentro de la estructura que representa a las tuplas y presentó problemas en la recepción de los datos, por esa razón se estableció el tamaño de 30 elementos como máximo para enviar un vector en una tupla al espacio de tuplas. Una mejora sería que en lugar de enviar bloques de 30 elementos, se pudiera enviar el arreglo completo sin importar el tamaño, inclusive arreglos bidimensionales, ya que en Linda – BUAP las matrices se tienen que manipular como vectores a la hora de formar una tupla para ser envíada o recuperada del espacio de tuplas, éste envío de las matrices respresenta un tráfico cargado en la transmisión dentro de la red y se sacrifica tiempo de procesamiento, por esta razón los proyectos presentados en este trabajo de tesis que implican el uso de arreglos no fueron tomados como representativos en cuestión de mejora de tiempo de procesamiento.

Otra perspectiva de este sistema para futuras versiones, es implementarlo en un paradigma orientado a objetos, para que permita resolver problemas paralelos utilizando las características potentes de la programación orientada a objetos.

Dentro de las mejoras en cuestión del desempeño, se puede pensar en el planteamiento de un modelo de memoria virtual distribuida – compartida. Es decir tener una memoria central, y que al mismo tiempo se tuviera una pequeña memoria de acceso rápido y en donde se encontraran los datos mas utilizados en forma local en cada uno de los trabajadores (workers), es decir presentar un modelo de Multiples Espacios de Tuplas.

En cuestión de la seguridad, se puede implementar un mecanismo para lograr un sistema más estable y tolerante a fallas.

Tomando como base este sistema, se pueden crear versiones en otros lenguajes tales como Java, Visual C++ y otros sistemas operativos, tales como Windows, Mac-Os o Solaris.

Aun y cuando se presenta una interfaz amigable en el sistema, se puede implementar una interfaz mejorada, en donde se muestre el contenido completo de las tuplas que se reciben y envían al master, y las tuplas que provienen y se

envían a la aplicación del usuario, barras de progreso que indicarán al usuario el progreso en el envío y recepción de código ejecutable entre el worker y el master También se podría implementar un monitoreo a base de ventanas (XWin) que fuera más vistoso.

En conclusión este sistema, aun cuando puede ser mejorado, es una herramienta que puede ser utilizado para la investigación y enseñanza, ya que muestra de una manera explicita la forma en como esta implementado un sistema de procesamiento paralelo.

BIBLIOGRAFIA

[1] La UNAM, protagonista del supercómputo en México (Septiembre 2004)
http://biblioteca.dgsca.unam.mx/nl/productos/boletines/msg00014.html

[2] Historia del supercómputo en la UNAM (Septiembre 2004)
http://www.super.unam.mx/super/index.php?cont=historia

[3] Artículo sobre supercomputadoras (Septiembre 2004)
http://matap.dmae.upm.es/WebpersonalBartolo/articulosdivulgacion/superordenadores.htm

[4] Historia de las computadoras (Agosto 2004)
http://www.dte.eis.uva.es/Docencia/ETSII/SMP/Ha_SComp/historia.html

[5] Linux on line! (Septiembre 2004)
www.linux.org

[6] Clusters en México, Historia de los Clusters (Agosto 2004)
http://clusters.unam.mx/Historia/index.php

[7] Sobre Beowulf (Septiembre 2004)
http://www.beowulf.org/

[8] Sobre Mosix y Openmosix (Septiembre 2004)
http://openmosix.sourceforge.net

[9] Web oficial de OpenMosix (Septiembre 2004)
http://openmosix.org

[10] Sobre Piraña (Septiembre 2004)
http://www.redhatla.com/detalle.php?IDSECCION=123&IDCONTENIDO=179

[11] PVM / MPI (Septiembre 2004)
http://orion.cecalc.ula.ve/recursos/software/fichasoftware/parser.php?URL=PVM.xml&XSL=software.xsl
http://orion.cecalc.ula.ve/recursos/software/fichasoftware/parser.php?URL=MPI.xml&XSL=software.xsl

[12] GOOGLE (Diciembre 2004)
http://www.google.com

[13] Clusters en México, Proyectos de clusters desarrollados en México
http://clusters.unam.mx/Clustersmx/index.php (Septiembre 2004)

[14] Introducción a PVM y MPI
Francisco Javier Gómez Arribas
Universidad Autónoma de Madrid (Octubre 2004)
http://www.ii.uam.es/~fjgomez/intropvm.html

[15] Coordination Models
Cinvestav (Noviembre 2004)
delta.cs.cinvestav.mx/~oolmedo/Coordination/1.pdf

[16] How to write parallel programs: A guide to perplexed
D. Gelernter and N. Carriero
ACM Computing Surveys, September 1989.
Pp. 323-357

[17] Coordination Models and Languages
George A. Papadopoulos and Farhard Arbad
Software Engineering (SEN)
SEN-R9834 December 31, 1998
Centrum voor Wiskunde
Pp. 6-8

[18] Efficient Parallel Programming with Linda
Ashis Deshpande and Martin Schultz
Yale University
Pp. 1,4,6,7

[19] Designing a Coordination Model for Open Systems
Thilo Kielmann
University of Siegen, Germany.
Pp. 3-5

[20] Tuple Spaces
Andrew B. Sudell (Octubre 2004)
http://c2.com/cgi/wiki?TupleSpace
http://webseitz.fluxent.com/wiki/TupleSpace
http://www.op.net/~asudell/is/linda/node4.html

[21] Linda Operations
Andrew B. Sudell (Octubre 2004)
http://www.op.net/~asudell/is/linda/node5.html

Kernighen Brian N y Ritchie Dennos M., El lenguaje de programación C
Segunda Edición, 1991
Prentice Hall

Kernighan Brian W. y Pike Rob, El entorno de programación Unix
Primera Edición, 1987
Prentice Hall

Robbins Kay A. y Robbins Steven,Unix Programación Práctica
Primera Edición, 1997
Prentice Hall

Shapley Gray John, Interprocess Communication in Unix
Second Edition, 1998
Prentice Hall

Valentín Gil Roberto y Cruz Almanza Graciano, Manual de Programación con PCS-Linda
F.C.C, BUAP, 2000

Anexo A

Manual de Usuario

El sistema Linda – BUAP cuenta con 2 programas, incluidos en el CD anexo a este documento:

Master. En la carpeta *master* del CD se encuentra un programa ejecutable llamado master. Se deberá copiar este programa en la máquina que desempeñará el papel de maestro. La distribución de Linux utilizada deberá tener instalado el paquete ncurses para poder observar el monitoreo.

Para ejecutar el programa master, deberá escribir la siguiente línea:

```
./master
```

Esto será suficiente para que el master se encuentre listo para recibir tuplas.

Para terminar la ejecución del master deberá presionar <ctrl.><c>.

Worker. En la carpeta *worker* se encuentran 3 archivos ejecutables:

- Worker, que ejecutará un worker con monitoreo para computadoras configuradas con resolución de 800 x 600 o superior.
- Worker2, que ejecutará un worker con monitoreo para computadoras configuradas con resolución de 640 X 480.
- Worker3, que ejecutará un worker sin monitoreo, para máquinas que no cuentan con el paquete ncurses instalado en la distribución de Linux.

En cualquiera de los 3 casos anteriores, para ejecutar el programa worker deberá escribir la siguiente línea:

```
./worker  <IP del master>
```

Esto será suficiente para que el worker esté listo para recibir trabajo.

Desarrollador. La máquina que desempeñará la función de desarrollador, deberá tener instalada el compilador de C gcc, que viene en todas las distribuciones de Linux, además deberá tener los archivos Linda1.h y Linda2.h proporcionados en la carpeta *ejemplos* del CD anexo a este documento.

Como ya se ha mencionado anteriormente los programas desarrollados por el usuario estarán divididos en dos:

a) Una aplicación principal que será la que se ejecutará y se encargará de distribuir las tareas en los workers.
b) Una segunda aplicación que contendrá la tarea a ejecutar en cada worker.

El archivo Linda1.h deberá incluirse en la aplicación principal del usuario (a), el archivo Linda2.h deberá incluirse en la aplicación que se distribuirá a los workers (b).

```
/* Programa que escribe en la pantalla del desarrollador los
mensajes Hola Mundo provenientes de los diferentes wks
activos el despliegue se realiza en forma ordenada */

#include <stdio.h>
#include <string.h>
#include "Linda1.h"
#define Nproc 200

main()
{ int i,proc; long ti,tf,tt;
  char *master="10.34.6.6"; (IP del master)
  init_linda(master);
  out("%s%d","cont",0);
  for(i=1; i<=Nproc; i++)  eval("mundo");
  for(i=1;i<=Nproc;i++)
  {in("%s%d","hola",i);
  printf("Hola desde Worker %d\n",i);}
  in("%s?d","cont",&proc);
 end_linda();
}
```

Programa Principal en Linda – BUAP (PROY1_A.C)

```
/* FUNCION QUE SE DISTRIBUYE A LOS WK PARA
PROY1_A.C*/
#include <stdio.h>
#include <stdlib.h>
#include "Linda2.h"

main()
{ int num;
  ini_app();
  in("%s?d","cont",&num);
  num++;
  out("%s%d","cont",num);
  out("%s%d","hola",num);
  fin_app();
}
```

Programa que se distribuye a los workers (MUNDO.C)

La compilación de estos programas se hará por separado, y será como comúnmente se hace con cualquier aplicación desarrollada en C para linux, como se muestra a continuación:

```
gcc proy1_a.c –o p1_a
```

```
gcc mundo.c –o mundo
```

Observe que el nombre del archivo que se envía a los workers debe corresponder al nombre proporcionado en la tupla eval del código fuente de la aplicación principal.

Posterior a esto, para iniciar la ejecución de su programa en Linda – BUAP, deberá ejecutar el programa principal de la forma siguiente:

```
./p1_a
```

En el CD , en la carpeta *ejemplos*, se encuentran varios programas que podrá analizar y ejecutar para poder entender mas claramente como trabaja el sistema Linda – BUAP V.1.0.