Algoritmos de compresión paralela

Autores

Alfredo Anderson
Delio Dirazar

Directores

Armando De Giusti
Hugo Ramón
Claudia Russo

Indice

Introducción

Tareas realizadas

El objetivo planteado inicialmente fue analizar la viabilidad de distribuir un compresor de datos en una red de procesadores.

Además de elegir el algoritmo a implementar y definir alternativas de distribución debíamos seleccionar un lenguaje y un sistema operativo que soporten las herramientas de multiprocesamiento necesarias para la implementación de las versiones distribuidas.

Nuestro primer paso fué realizar la implementación de dos compresores de datos basados en el mismo algortimo, uno distribuido y otro lineal, y analizar ventajas y desventajas entre ambas implementaciones.

El algoritmo de compresión en el que basamos las implementaciones fue el algoritmo de Huffman semiestático (ver Apéndice C2). Básicamente por ser un estándar de compresión de datos ampliamente divulgado, simple, efectivo, y con una importante base teórica.

Todas las implementaciones fueron realizadas utilizando el lenguaje de programación C, aumentado, en los casos en que fue necesario, con librerías correspondientes a las distintas herramientas.

Trabajamos sobre la implementación LINUX del sistema operativo UNIX.

Utilizamos PCs Intel pero debido a la portabilidad de todo el software utilizado (C, librerías de IPC, PVM, MPI, ...) es posible extender nuestro trabajo a otras plataformas UNIX (HP, SUN, DIGITAL, ...).

Una vez elegidos el algoritmo, el lenguaje de programación y las herramientas de software, el sistema operativo y la plataforma realizamos una implementación lineal del compresor de datos cuya performance (utilización de recursos, radios de compresión, ...) nos sirviera de referencia.

Luego paralelizamos esta implementación en un sólo procesador para familiarizarnos paulatinamente con las herramientas de intercomunicación de procesos y de multiprocesamiento (semáforos, memoria compartida, creación y manipulación de procesos, ...).

A continuación distribuimos la implementación anterior en una red de procesadores utilizando un esquema cliente/servidor y resolviendo la comunicación entre procesos "remotos" utilizando sockets.

Luego definimos una nueva política de paralelización/distribución del algoritmo lo que nos condujo a dos nuevas implementaciones equivalentes a las anteriores pero realizadas con un enfoque distinto. Una fue la paralelización en un sólo procesador del algortimo original utilizando la nueva opción y otra fue la distribución de esta última.

Por ese entonces se nos presentó la posibilidad de trabajar con dos conocidas herramientas para la implementación de programas distribuidos/paralelos, PVM (ver Apéndice H7) y MPI (ver Apéndice H8), ambas disponibles en la plataforma que habíamos seleccionado. Utilizamos como base para las nuevas implementaciones la implementación distribuida que había arrojado mejores resultados.

El objetivo de esta etapa era realizar una comparación práctica entre estas herramientas entre sí y entre estas y nuestro compresor.

Al realizar la implementación distribuida del algoritmo utilizando MPI surgió la idea de espera selectiva en la atención de las respuestas de los servidores. Esto nos condujo a una nueva implementación distribuída.

En este punto habíamos confirmado que la distribución del algoritmo permitía mejorar en gran medida la velocidad de ejecución. Por lo que decidimos realizar la distribución sobre un algoritmo de compresión más eficiente. Para que el resultado final sea un compresor (distribuido) mejor en cuanto a velocidad que los compresores lineales y con radios de compresión cercanos a los de los compresores reales (gzip, pkzip, ...).

La elección obvia era LZW pero llegó a nuestras manos un artículo en el cual se presentaba una transformada poco conocida, que permitía alcanzar radios de compresión mejores que los alcanzables con el pkzip o aún con el gzip. Estamos hablando de la transformada Burrows Wheeler (BWT). Luego de obtener los fuentes y un conjunto de archivos para probar compresores (Calgary Compression Corpus), realizamos comparaciones contra el mejor compresor comercial que conseguimos (gzip) y en efecto, comprobamos, que esta nueva transforma permitía alcanzar mejores radios de compresión.

Ahora sólo restaba distribuir esta nueva transformada.

Aqui se nos presentó el mismo problema que tuvimos inicialmente cuando debíamos distribuir el algoritmo de Huffman. Esto no implicaba nada nuevo pero se nos ocurrió algo que a nuestro modo de ver derivó en lo más interesante de nuestro trabajo: la creación de un mecanismo general que permita extender una implementación lineal de un algoritmo a una implementación distribuida. Nosotros lo acotamos a compresores, descompresores y filtros (lineales) pero el mismo puede ser extendido sin problemas a un gran número de aplicaciones. Por ejemplo, un esquema similar puede ser utilizado para acelerar el procesamiento de señales en general.

Realizamos la implementación de este mecanismo y lo probamos con un compresor/descompresor basado en la transformada BWT, con el comando cat utilizado como un filtro y con los compresores/descompresores gzip/gunzip y compress/uncompress.

Estas pruebas nos permitieron, entre otras cosas, comprobar el correcto funcionamiento de la herramienta, evaluar la utilidad de la misma, y verificar la facilidad con la que se pueden generar versiones distribuidas de compresores, descompresores y filtros a partir de implementaciones lineales de los mismos.

Organización del libro

El libro está dividido en tres partes.

La primera parte contiene los trabajos realizados tomando como método de compresión el algoritmo de Huffman.

La segunda parte está basada en la herramienta para la distribución del procesamiento.

Por último, en la tercera parte se incluyen varios apéndices asociados con la compresión de datos y las herramientas de desarrollo utilizadas.

Parte I: Análisis sobre la viabilidad de la distribución de un compresor de datos basado en el algoritmo de Huffman

Capítulo 1

Introducción

En esta parte del libro se presentan las diferentes implementaciones del algoritmo Huffman y los resultados de la ejecución de varios lotes de pruebas.

En el Capítulo 2 se detalla la implementación lineal básica del algoritmo de Huffman realizada para tener un punto de referencia a partir del cual medir el desempeño de las versiones distribuidas.

En el Capítulo 3 se detalla una implementación paralelizada del algoritmo de Huffman utilizando un único diccionario global.

En el Capítulo 4 se detalla una implementación distribuida del algoritmo de Huffman utilizando un único diccionario global.

En el Capítulo 5 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para las tres implementaciones anteriores junto con algunas conclusiones relevantes.

En el Capítulo 6 se detalla una implementación paralelizada del algoritmo de Huffman utilizando un diccionario por cada bloque de procesamiento.

En el Capítulo 7 se detalla una implementación distribuida del algoritmo de Huffman utilizando un diccionario por cada bloque de procesamiento.

En el Capítulo 8 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para la versión lineal y las dos versiones distribuidas junto con algunas conclusiones relevantes.

En el Capítulo 9 se detalla una implementación distribuida similar a la del Capítulo 7 realizada utilizando PVM.

En el Capítulo 10 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para la versión distribuida del Capítulo 7 y para la versión distribuida del Capítulo 9 junto con algunas conclusiones relevantes.

En el Capítulo 11 se detalla una implementación distribuida en parte similar a la del Capítulo 7 realizada utilizando MPI.

En el Capítulo 12 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para la versión lineal y las versiones distribuidas correspondientes a los capítulos 7, 9 y 11 junto con algunas conclusiones relevantes.

En el Capítulo 13 se detalla una implementación distribuida del algoritmo de Huffman utilizando un diccionario por cada bloque de procesamiento y espera selectiva en la atención de las respuestas de los servidores.

En el Capítulo 14 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para las versiones distribuidas correspondientes a los capítulos 7, 11 y 13 junto con algunas conclusiones relevantes.

En el Capítulo 15 se presentan un cuadro comparativo y conclusiones generales asociadas a esta parte del libro.

Capítulo 2

Huffman lineal

Introducción y descripción del proceso de codificación

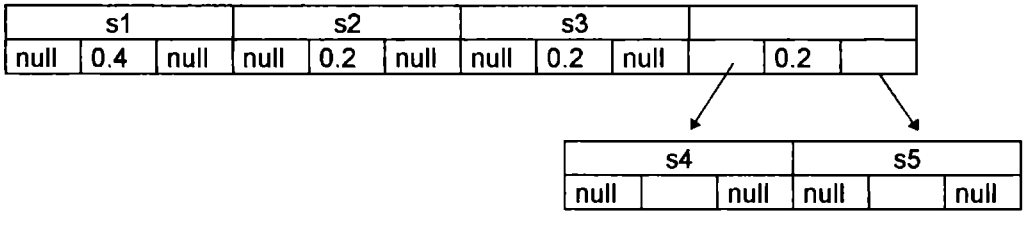
A continuación describiremos la implementación lineal del algoritmo de Huffman. Esta se utilizará como base en el análisis de ventajas y desventajas de distribuir/paralelizar. Para que las comparaciones tengan validez, todas las versiones distribuidas/paralelas son variaciones de esta implementación básica. Asumimos que el lector está familiarizado con la codificación de Huffman, por lo que sólo describiremos el método particular de construcción del código utilizado. También describiremos los procesos de codificación y decodificación. Para profundizar sobre la codificación de Huffman consultar el apéndice C2.

El algoritmo recodifica los símbolos de la fuente de acuerdo a la estructura probabilística de la misma, asignando menos unidades de información a los símbolos más probables y más a los menos probables. Para realizar esta recodificación se analiza la estructura probabilística de la fuente. Se genera el histograma de la misma, a partir de este se genera un Código de Huffman, y por último se recodifica la fuente.

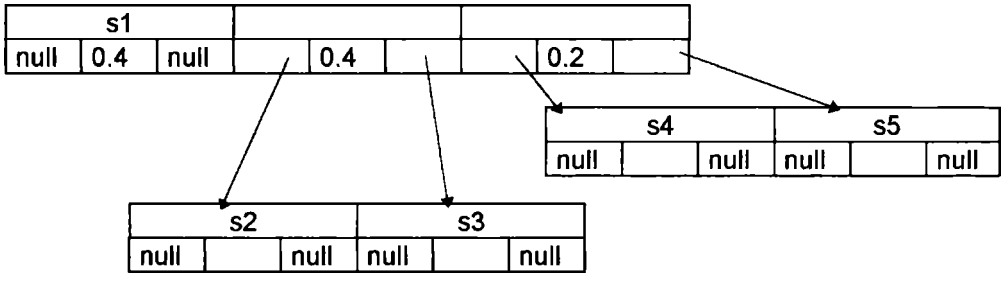
El algoritmo es el siguiente. Inicialmente se calculan las probabilidades de ocurrencia de los símbolos de la fuente (Crear_histograma). A continuación se procede a generar el código a partir de las probabilidades calculadas (Armar_Arbol). Para la generación del código se utiliza un arreglo de nodos hojas que contienen los símbolos y sus ocurrencias o probabilidades.

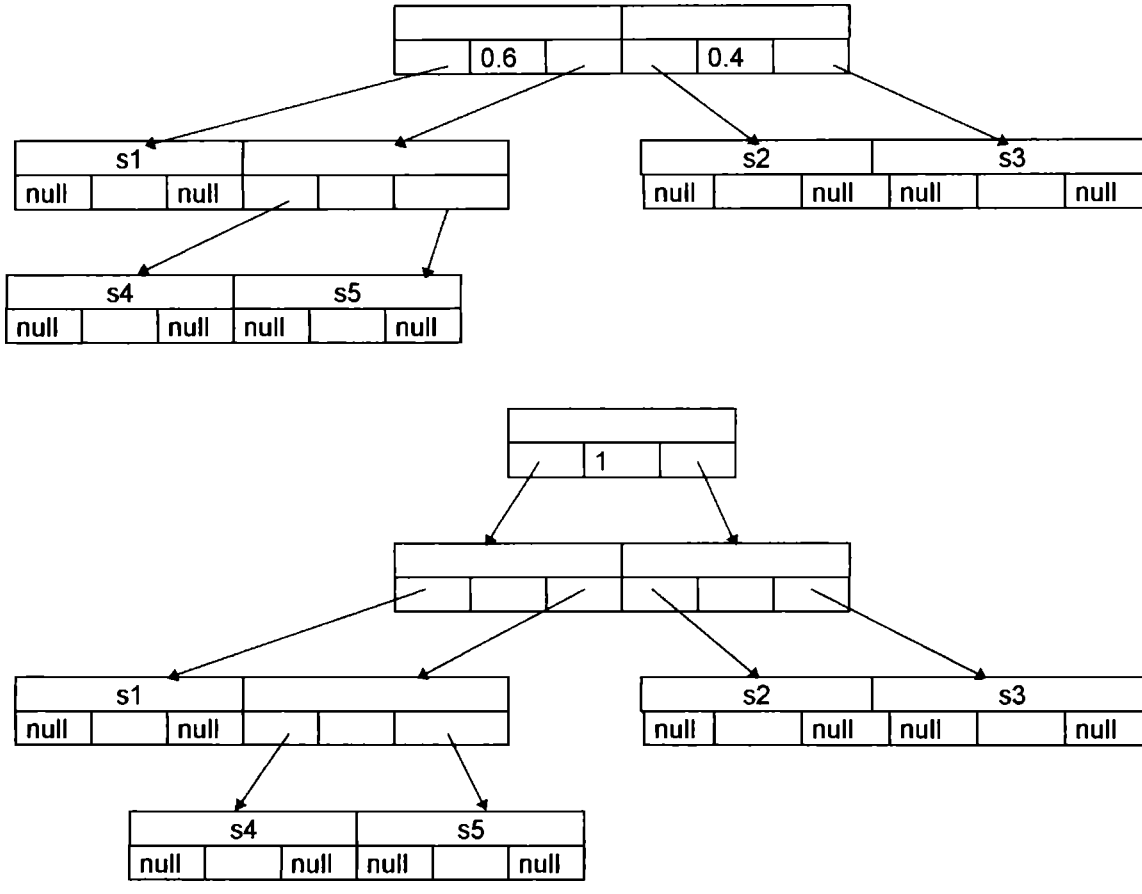
s1			s2			s3			s4			s5		
null	0.4	null	null	0.2	null	null	0.2	null	null	0.1	null	null	0.1	null

Se recorre el arreglo para encontrar los nodos de mínimas ocurrencias. Una vez identificados estos nodos, al mínimo se lo elimina del arreglo y al nuevo mínimo se le suma la cantidad de ocurrencias del eliminado y se le insertan un nuevo hijo izquierdo y un nuevo hijo derecho. El hijo izquierdo contiene el carácter, el hijo izquierdo y el hijo derecho del nuevo mínimo; el hijo derecho contiene el carácter, el hijo izquierdo y el hijo derecho del mínimo eliminado.



Este proceso se repite hasta que solo quede un nodo en el arreglo.





Una vez generado el árbol de codificación, lo pasamos a un arreglo para tener acceso directo a la palabra de código correspondiente a un símbolo (Armar_codigo_y_header). Se recorre el árbol en *pre-order* guardando el camino corriente (agregando a la palabra de código un '0' para el hijo izquierdo y un '1' para el hijo derecho). Cada vez que se alcanza una hoja, se asigna a la celda del arreglo correspondiente al caracter contenido en la hoja, el camino corriente.

arreglo[s1] = "00"
arreglo[s2] = "10"
arreglo[s3] = "11"
arreglo[s4] = "010"
arreglo[s5] = "011"

Además de generar el arreglo de acceso directo se aprovechó esta función para generar la parte del header del archivo comprimido correspondiente al código de la siguiente manera. Cada vez que se saltaba de un nodo a otro se grababa un 0, cada vez que se alcanzaba una hoja se grababan un 1 y el caracter contenido en la hoja. Como veremos más adelante, sólo con esto se puede reconstruir el árbol de decodificación. La secuencia de salida para nuestro ejemplo sería la siguiente: 001's1'01's4'1's5'01's2'1's3'.

A continuación se procede a realizar la codificación (Codificar) utilizando la siguiente secuencia de instrucciones:

```
while( (c = getch(ifp)) != EOF )
        escribir_bits(arreglo[c]);
```

Refinamiento del proceso de compresión

```
Crear_histograma(Archivo_de_entrada, &Histograma);
Armar_arbol(Histograma, &Arbol);
Armar_codigo_y_header(Arbol, &Header, &Codigo);
Grabar_header(Archivo_de_salida, Header);
Codificar(Archivo_de_entrada, Codigo,Archivo_de_salida);
```

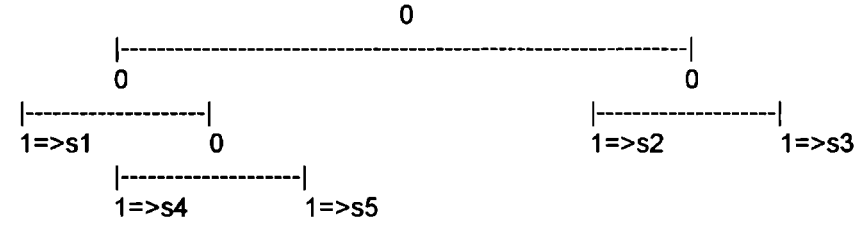
Descripción del proceso de descompresión

A continuación describimos el proceso de descompresión. Esto implica la lectura del Header (Tamaño del archivo descomprimido y Árbol de decodificación comprimido), la reconstrucción del Árbol de decodificación y la decodificación propiamente dicha.

El proceso de reconstrucción del árbol a partir del header es el siguiente:

1. Leer bit ,
2. Si el valor del bit es '0', crear un nodo y realizar el paso 1, primero para el hijo izquierdo y luego para el hijo derecho.
3. Si el valor del bit es '1', leer el símbolo (8 bits), crear un nodo hoja y asignarle el símbolo leído

Veamos como quedaría el árbol reconstruido para la salida de nuestro ejemplo: 001's1'01's4'1's5'01's2'1's3' .



El proceso de decodificación se realiza de la siguiente manera:

1. nos posicionamos en la raíz del árbol de decodificación
2. leemos un bit
3. si el bit es '0', nos movemos al hijo izquierdo, si es '1' al hijo derecho
4. si el nodo en el que estamos es una hoja, grabamos el caracter y recomenzamos en el punto 1.
5. si el nodo no es una hoja, saltamos al punto 2.

Notar que a este proceso le falta el criterio de terminación que consiste en contar la cantidad de caracteres que fuimos sacando hasta llegar al tamaño del archivo original. Esto fue ignorado deliberadamente para facilitar la comprensión.

Refinamiento del proceso de descompresión

```
Rearmar_arbol(Archivo_de_entrada,&Arbol);
Decodificar(Archivo_de_entrada, Arbol, Archivo_de_salida);
```

Estructura del archivo comprimido

Tamaño del Archivo Original
Código Comprimido (Árbol de Decodificación)
Datos recodificados

El tamaño del archivo original es necesario ya que no siempre se utilizan todos los bits del último byte del archivo de salida y puede ser que estos bits "sobrantes" conformen una palabra de código válida.

Capítulo 3

Huffman paralelo con diccionario único (paralelizado monoprocesador)

Introducción

Una vez implementado el algoritmo de Huffman lineal descripto en el capítulo anterior, estábamos en condiciones de paralelizarlo.

Analizando el código del programa anterior notamos que podíamos utilizar varios procesos similares que cooperaran en la realización del trabajo. Esto nos permitiría aprovechar los tiempos muertos de E/S. En el caso anterior, cada vez que el programa debía realizar una E/S, efectuaba una llamada al sistema y generalmente se quedaba bloqueado hasta que esta se completara (para más detalles referirse a los capítulos 3 y 6 de [Bach86]). En esta nueva implementación, cada vez que un proceso se bloquee en una E/S, probablemente exista otro que pueda efectuar un uso productivo de la CPU.

La idea general es la siguiente, cada vez que se debe procesar un gran conjunto de datos, en vez de realizar el procesamiento secuencialmente sobre todo el conjunto, este se divide en varios bloques que son procesados en forma paralela por un pool de procesos cooperativos.

Vale resaltar que aunque estos procesos trabajen sobre bloques de datos disjuntos deben sincronizarse para acceder en forma "segura" a los recursos compartidos (en nuestro caso archivos y segmentos de memoria compartida).

La forma en que llevamos a la práctica esta idea fue la siguiente: inicialmente se cacula el número de tareas que a realizar y con este valor se setea un contador compartido por el pool de procesos cooperativos. Estos procesos utilizan este contador para coordinarse mediante el siguiente protocolo: si queda alguna tarea pendiente, decrementar la cantidad de tareas pendientes en uno y realizar una. Si no, terminar.

Una aproximación al código final puede ser la siguiente: Un proceso (padre) setea la cantidad de tareas a realizar de acuerdo al tamaño del archivo y al tamaño de los bloques de E/S, luego dispara M procesos (hijos) que se encargan de realizar c/u de las tareas.

El proceso principal se queda esperando la terminación de los M procesos subordinados. Cuando esto sucede, la tarea total fue realizada y él puede continuar su ejecución.

```
PROCESO_PADRE::
        tareas_pendientes = Calcular_el_numero_total_de_tareas(fs, bs);
        for(i=0; i<M; i++) run PROCESO_HIJO[i];
        esperar_finalizacion();
        ...
```

Como dijimos, los procesos subordinados deben sincronizarse para realizar su labor correctamente. Por ejemplo, el contador de tareas compartido debe ser accedido en forma exclusiva para evitar la interferencia.

El acceso exclusivo al contador de tareas compartido fue modelizado de la siguiente forma:

```
PROCESO_HIJO::
        fin = FALSE;
        do
                P(trabajos_pendientes);
                if ( !Hay_trabajos_pendientes )
                        fin = TRUE;
                else
                        Decrementar_trabajos_pendientes;
                V(trabajos_pendientes);
                if (!fin)
                        realizar_tarea();
        } while (!fin);
```

Analizando el código de la implementación lineal del algortimo de Huffman notamos que este método se podía aplicar sobre tres etapas: la generación del histograma, la codificación, y la decodificación.

A continuación pasaremos a describir cada caso en particular.

Generación del histograma

En vez de que un solo proceso cree el histograma, se divide el archivo en slots y se disparan procesos contadores que van tomando slots y generando el histograma correspondiente. Cuando ya no hay slots disponibles suman el histograma local a un histograma global y terminan. Cuando todos terminan, el histograma global es el histograma del archivo de entrada.

```
Generar_Histograma_en_paralelo::
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_contadores(Numero_de_procesos_contadores);

Contador[1..M]::

        fin = FALSE;
        do
                P(s_shmem->trabajos_pendientes);
                if ( !Hay_trabajos_pendientes ) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) crear_sub_histograma( sub_histograma );
        } while (!fin);
        P(s_shmem->histograma);
        Sumar( s_shmem->histograma, sub_histograma );
        V(s_shmem->histograma);
```

Codificación

En vez de que un solo proceso codifique todo el archivo, se divide el archivo en slots y se dispararan procesos codificadores que toman un slot y lo codifican. A diferencia del procedimiento anterior, al terminar de procesar cada slot los procesos deben realizar una escritura en un archivo compartido. Obsérvese además que los procesos deben agregar información referente a la ubicación del slot en el archivo original (ya que los slots comprimidos no necesariamente estarán ordenados) y al tamaño del slot comprimido.

```
Codificar_en_paralelo::
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_codificadores(Numero_de_procesos_codificadores);

Codificador[1..M] ::

        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                Tomar_desplazamiento_en_el_archivo_de_entrada(ifo);
                        Leer(Archivo_de_entrada, buffer_in, Tamano_de_slot);
                        V(s_shmem->input_file);
                        comprimir_slot( buffer_in , buffer_out, obs);
                        P(s_shmem->output_file);
                        write(_ofh, &ifo, sizeof(ifo));    /* offset del slot en el input file */
                        write(_ofh, &obs, sizeof(obs)); /* longitud en bits del slot comprimido */
                        write(_ofh, buffer_out, obs/8 + ((obs%8)+7)/8 );
                        V(s_shmem->output_file);
                }
        } while (!fin);
```

Decodificación

En vez de que un solo proceso decodifique todo el archivo, se divide el archivo en slots y se dispararan procesos decodificadores que toman un slot y lo decodifican.

Cabe aclarar que, una vez regenerado un slot, los procesos se posicionan en el archivo de salida para grabar el slot en su posición original.

```
Decodificar_en_paralelo::

        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_decodificadores(Numero_de_procesos_decodificadores);
```

Decodificador [1..M]::

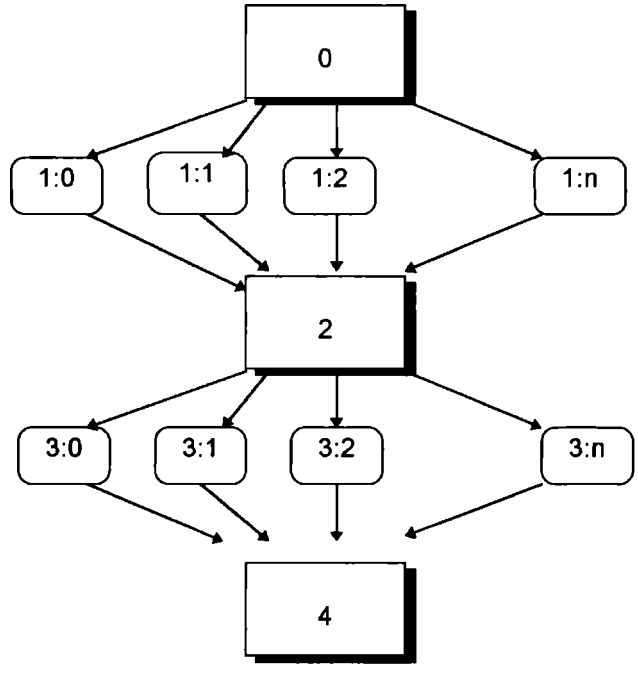
```
fin = FALSE;
do {
        P(s_shmem->trabajos_pendientes);
        if (!Hay_trabajos_pendientes) fin = TRUE;
        else Decrementar_trabajos_pendientes;
        V(s_shmem->trabajos_pendientes);
        if (!fin) {
                P(s_shmem->input_file);
                read(_ifh, &ofo, sizeof(ofo));
                read(_ifh, &ibs, sizeof(ibs));
                bytes_leidos = read(_ifh, buffer_in, (ibs/8) + ((ibs%8)+7)/8);
                V(s_shmem->input_file);
                descomprimir_slot( buffer_in , bytes_leidos, buffer_out, ibs, &obs);
                P(s_shmem->output_file);
                lseek(_ofh, ofo, SEEK_SET);
                write(_ofh, buffer_out, obs);
                V(s_shmem->output_file);
        }
} while (!fin);
```

Refinamiento

Compresor

```
Crear_histograma_en_paralelo(Archivo_de_entrada, &Histograma);
Armar_arbol(Histograma, &Arbol);
Armar_codigo_y_header(Arbol, &Header, &Codigo);
Grabar_header(Archivo_de_salida, Header);
Codificar_en_paralelo(Archivo_de_entrada, Codigo, Archivo_de_salida);
```

Gráficamente,



0 (Main):
Setear el número de trabajos en función del tamaño del archivo y del tamaño del bloque de E/S.
Disparar los contadores (procesos 1:i).

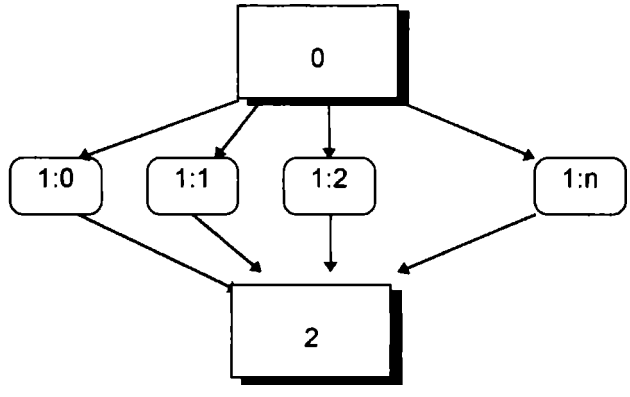
1: i (Contadores):
Mientras haya tareas pendientes
Leer slot.
Actualizar el histograma.
Agregar el histograma local al histograma general.

2 (Main):
Armar el diccionario.
Grabar header en archivo de salida.
Setear el número de trabajos en función del tamaño del archivo y del tamaño del bloque de E/S.
Disparar los compresores (procesos 3:i).

3:i (Compresores):
Mientras haya tareas pendientes
Leer slot.
Comprimir slot.
Grabar header de slot y slot comprimido en archivo de salida.
4 (Main):
Liberar recursos y finalizar.

Descompresor

Rearmar_arbol(Archivo_de_entrada, &Arbol);
Decodificar_en_paralelo(Archivo_de_entrada, Arbol, Archivo_de_salida);



0 (Main):
Leer el header del archivo comprimido.
Reconstruir el árbol de decodificación en memoria compartida.
Setear el número de trabajos.
Disparar los descompresores (procesos 1:i).

1: i (Descompresores):
Mientras haya tareas pendientes
Leer header de slot y slot comprimido.
Descomprimir slot.
Grabar el slot (en el offset original) en el archivo de salida.

2 (Main):

Liberar recursos y finalizar.

Estructura del Archivo Comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Código Comprimido - Inorder del "Arbol de Huffman"
Slots Comprimidos

- **Estructura de los Slots Comprimidos:**

Offset del slot descomprimido en el archivo original
Tamaño en bits del slot comprimido
Slot comprimido

Capítulo 4

Huffman paralelo con diccionario único (paralelizado multiprocesador)

Introducción

Una vez realizada la implementación paralelizada monoprocesador, el paso siguiente era distribuir el compresor en una red de procesadores. Esto podía ser realizado moviendo el procesamiento intensivo en los procesos contadores, codificadores y decodificadores a otros procesadores a través de una arquitectura cliente/servidor.

La idea general es la siguiente, los procesos contadores, codificadores y decodificadores ya no realizan su trabajo localmente, sólo se encargan de la sincronización de la E/S y de la comunicación con los servidores. Estos últimos son ahora los encargados de realizar el procesamiento intensivo (generar un histograma, codificar o decodificar un bloque de datos).

En resumen, utilizando el método de paralelización descripto en el capítulo anterior trasladamos el procesamiento intensivo de los procesos contadores, codificadores y decodificadores a otros procesadores.

La estructura de este algorítmo es similar a la estructura del algoritmo del capítulo anterior, es decir se crean los M subprocesos y cada uno se encarga de realizar una tarea por vez en forma coordinada y cooperativa. Pero la realización de la tarea no implica el cómputo intensivo local sino la comunicación con servidores remotos.

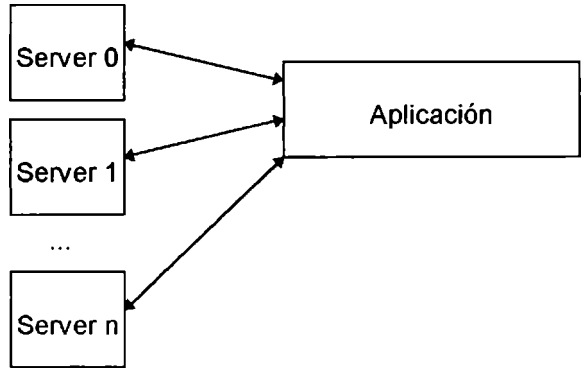
El nuevo formato general es estos procesos es el siguiente:

```
PROCESO_HIJO[1..M]::
        fin = FALSE;
        do {
                P(trabajos_pendientes);
                if ( !Hay_trabajos_pendientes ) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(trabajos_pendientes);
                if (!fin)
                        send ( requerimiento , &resultado )
        } while (!fin);
```

Notar que con este simple esquema, se realiza una utilización muy cercana a la óptima de los recursos de procesamiento ya que las tareas se asignan dinámicamente a los servidores a medida que estos están disponibles; esto generará naturalmente una buena política de distribución de los trabajos entre los servidores.

El método de comunicación utilizado entre los clientes y los servidores fue el usual en los sistemas UNIX. Los servidores "escuchan" por requerimientos en un determinado port, cuando uno llega, forkean un proceso que se conecta con el cliente para satisfacer sus necesidades y ellos continuan escuchando a la espera de mas requerimientos. Para más datos referirse a los apéndices H5 y H6.

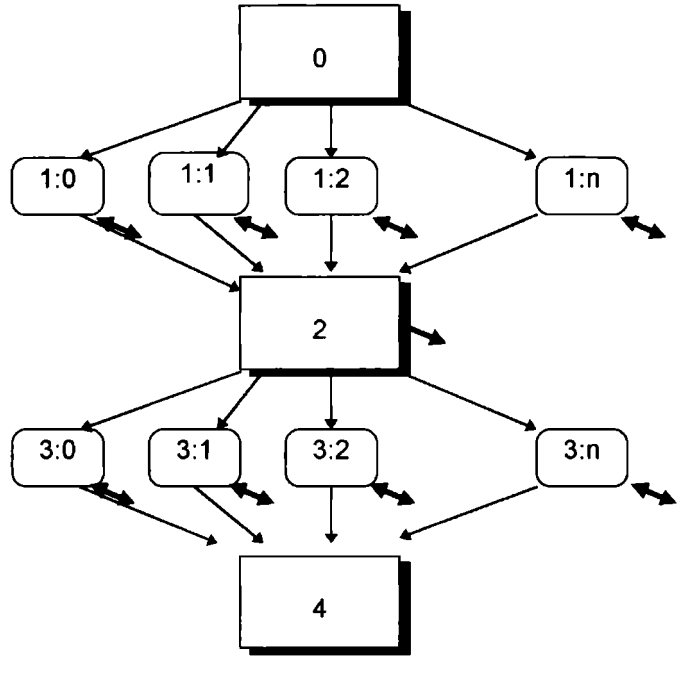
Gráficamente,



Veamos más detalladamente las estructuras del compresor, del descompresor y de los servidores.

Compresor

Puede verse que esta es similar a la del compresor descripto en el capítulo anterior, excepto por los siguientes detalles: procesos contadores utilizados para sincronización y comunicación, distribución del código, procesos codificadores utilizados para sincronización y comunicación.



Notas:

——→ : fork/join

←——→ : comunicación con servidor

0 (Main):

Setear el número de trabajos en función del tamaño del archivo y del tamaño de bloque de E/S.

Disparar los contadores (procesos 1:i).

1: i (Contadores):
    Mientras haya tareas pendientes
        Leer slot.
        Enviar slot al servidor asociado (contador).
    Recibir histograma de los bloques enviados.
    Agregar el histograma local al histograma general.

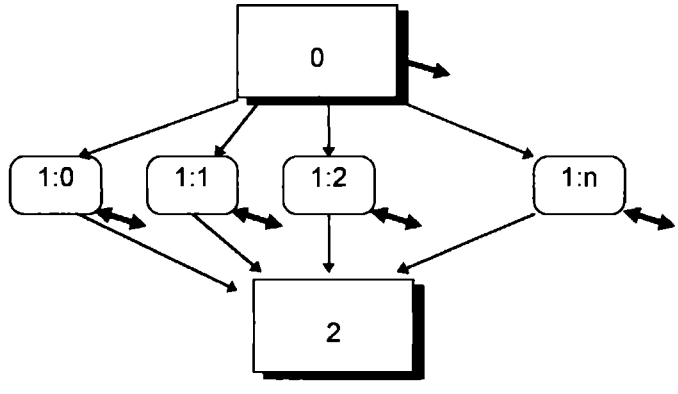
2 (Main):
    Armar el diccionario.
    Grabar header en archivo de salida.
    Distribuir el diccionario a los servidores.
    Disparar los compresores (procesos 3:i).

3:i (Compresores):
    Mientras haya tareas pendientes
        Leer slot.
        Enviar slot al servidor asociado (compresor).
        Recibir slot comprimido.
        Grabar header de slot y slot comprimido en archivo de salida.

4 (Main):
    Liberar recursos y finalizar.

Descompresor

Similar a la del descompresor descripto en el capítulo anterior, excepto por la distribución del código y por la utilización de los procesos decodificadores utilizados para sincronización y comunicación.



Notas:
——→ : fork/join
←→ : comunicación con servidor

0 (Main):
    Leer el header del archivo comprimido.
    Enviar el diccionario a los servers.
    Disparar los descompresores (procesos 1:i).

1: i (Descompresores):
    Mientras haya tareas pendientes
        Leer header de slot y slot comprimido.

Enviar slot comprimido al servidor asociado (descompresor).
Recibir slot descomprimido.
Grabar el slot (en el offset original) en el archivo de salida.

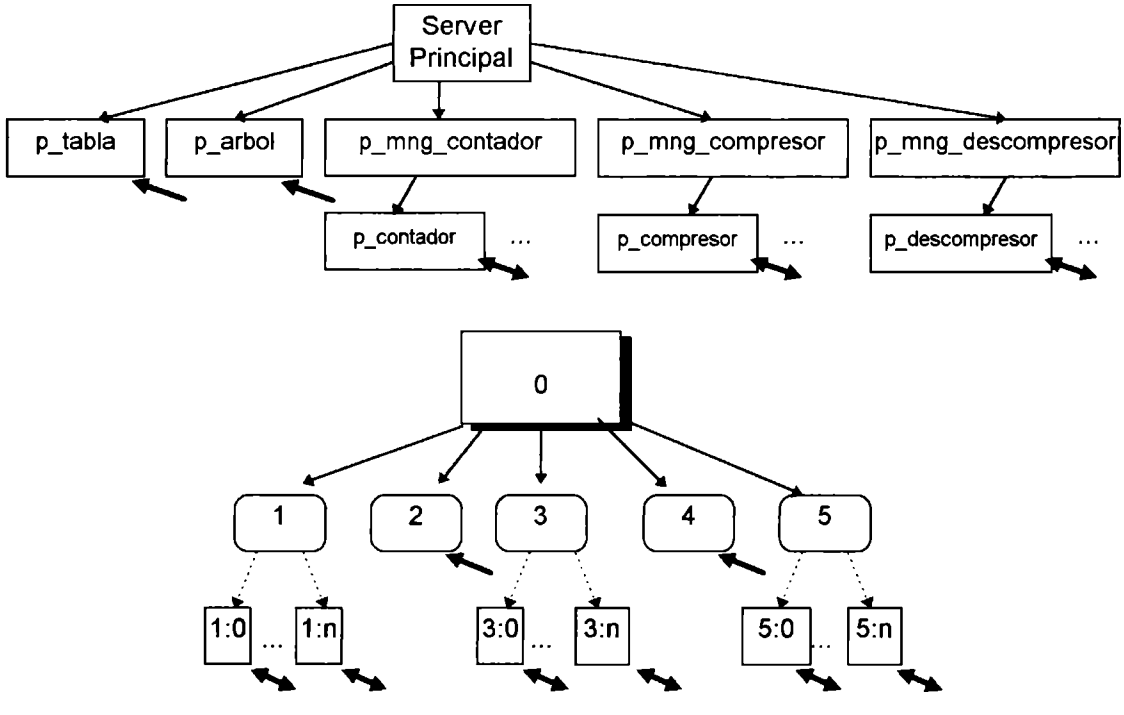
2 (Main):
Liberar recursos y finalizar.

Servidores

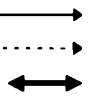
Los servidores son los encargados de atender los requerimientos de la aplicación principal.

Existen 3 tipos básicos de servicios, Generar Histograma, Comprimir y Descomprimir. Existen también dos servicios especiales que son utilizados para inicializar el diccionario en sus dos formatos: inicialización de la tabla utilizada por los procesos compresores e inicialización del árbol utilizado por los procesos descompresores.

El funcionamiento de un servidor es como sigue. El proceso principal dispara cinco procesos, p_tabla que graba el diccionario en forma de tabla en memoria compartida, p_arbol que graba el diccionario en forma de árbol en memoria compartida, p_mng_contador el cual cada vez que recibe un requerimiento CONTADOR dispara un proceso contador (p_contador) y establece la comunicación entre este y el cliente, p_mng_compresor el cual cada vez que recibe un requerimiento COMPRESOR dispara un proceso compresor (p_compresor) y establece la comunicación entre este y el cliente y por último, p_mng_descompresor el cual cada vez que recibe un requerimiento DESCOMPRESOR dispara un proceso descompresor (p_descompresor) y establece la comunicación entre este y el cliente.



Notas:



0 (Deamon principal):
Inicializar datos globales (memoria compartida para los dos formatos posibles del código).
Disparar procesos servidores (1..5).

1 (Servidor Creador de Contadores):
Recibir requerimiento.
Disparar Servidor Contador y asociarlo al cliente.

1:i (Servidor Contador):
Mientras no finalicen los requerimientos
Recibir un slot.
Actualizar el histograma local.
Retornar el histograma al cliente.

2 (Servidor de inicialización para la compresión):
Inicializa el diccionario global para la compresión.

3 (Servidor Creador de Compresores):
Recibir requerimiento.
Disparar Servidor Compresor y asociarlo al cliente.

3:i (Servidor Compresor):
Mientras no finalicen los requerimientos
Recibir un slot.
Comprimir slot.
Retornar el slot comprimido al cliente.

4 (Servidor de inicialización para la descompresión):
Inicializa el diccionario global para la descompresión.

5 (Servidor Creador de Descompresores):
Recibir requerimiento.
Disparar Servidor Descompresor y asociarlo al cliente.

5:i (Servidor Descompresor):
Mientras no finalicen los requerimientos
Recibir un slot.
Descomprimir slot.
Retornar el slot descomprimido al cliente.

Refinamiento

Compresor

```
Crear_histograma_en_paralelo(Archivo_de_entrada, &Histograma);
Armar_arbol(Histograma, &Arbol);
Armar_codigo_y_header(Arbol, &Header, &Codigo);
Grabar_header(Archivo_de_salida, Header);
Distribuir_codigo(Codigo);
```

```
Codificar_en_paralelo(Archivo_de_entrada, Archivo_de_salida);
```

El procedimiento Crear_histograma_en_paralelo se paralelizó y distribuyó de la siguiente forma:

Al igual que en el capítulo anterior, en vez de que un solo proceso cree el histograma, se divide el archivo en slots, pero en este caso se dispararán procesos que se comunicarán con procesos contadores que servirán sus requerimientos.

```
Crear_histograma_en_paralelo::

    Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot)
    Disparar_procesos_contadores_com(Numero_de_procesos_contadores_com)

Proceso_contador_com[i:1..M] ::

    socket  = Abrir_socket( CONTADOR );
    fin = FALSE;
    do {
        P( s_shmem->trabajos_pendientes);
        if ( !Hay_trabajos_pendientes ) fin = TRUE;
        else Decrementar_trabajos_pendientes;
        V(s_shmem->trabajos_pendientes);
        if (!fin) enviar_slot( socket );
    } while (!fin);
    read( socket , &sub_histograma );
    close( socket );
    P(s_shmem->histograma);
    Sumar( s_shmem->histograma, sub_histograma );
    V(s_shmem->histograma);
```

El procedimiento Codificar_en_paralelo se paralelizó y distribuyó de la siguiente forma:

En vez de que un solo proceso codifique todo el archivo, se divide el archivo en slots y se dispararán procesos que se comunicarán con los procesos codificadores en los servers.

```
Codificar_en_paralelo::

    Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot)
    Disparar_procesos_codificadores_com(Numero_de_procesos_codificadores_com)

Proceso_codificador_com[i:1..M] ::

    socket  = Abrir_socket(CODIFICADOR);
    fin = FALSE;
    do {
        P(s_shmem->trabajos_pendientes);
        if (!Hay_trabajos_pendientes) fin = TRUE;
        else Decrementar_trabajos_pendientes;
        V(s_shmem->trabajos_pendientes);
        if (!fin) {
            P(s_shmem->input_file);
            Tomar_desplazamiento_en_el_archivo_de_entrada(ifo);
            Leer(Archivo_de_entrada, buffer_in, Tamano_de_slot);
```

```
                    V(s_shmem->input_file);
                    comprimir_slot( socket, buffer_in , buffer_out, obs); /* Escribir y leer de un
socket */
                    P(s_shmem->output_file);
                    write(_ofh, &ifo, sizeof(ifo)); /* offset del bloque en el input file */
                    write(_ofh, &obs, sizeof(obs)); /* longitud en bits del buffer */
                    write(_ofh, buffer_out, obs/8 + ((obs%8)+7)/8 );
                    V(s_shmem->output_file);
            }
    } while (!fin);
```

Descompresor

```
    Leer_header(Archivo_de_entrada, &Header);
    Distribuir_header(Header);
    Decodificar_en_paralelo(Archivo_de_entrada, Archivo_de_salida);
```

El procedimiento Decodificar_en_paralelo se paralelizó y distribuyó de la siguiente forma:

Al igual que en el capítulo anterior, en vez de que un solo proceso decodifique todo el archivo, se divide el archivo en slots pero en este caso se dispararán procesos que se comunicarán con los procesos decodificadores.

Decodificar_en_paralelo::

```
    Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
    Disparar_procesos_decodificadores_com(Numero_de_procesos_decodificadores_com);
```

Proceso_decodificador_com[i:1..M] ::

```
    socket  = Abrir_socket( DECODIFICADOR );
    fin = FALSE;
    do {
            P(s_shmem->trabajos_pendientes);
            if (!Hay_trabajos_pendientes) fin = TRUE;
            else Decrementar_trabajos_pendientes;
            V(s_shmem->trabajos_pendientes);
            if (!fin) {
                    P(s_shmem->input_file);
                    read(_ifh, &ofo, sizeof(ofo));
                    read(_ifh, &ibs, sizeof(ibs));
                    bytes_leidos = read(_ifh, buffer_in, (ibs/8) + ((ibs%8)+7)/8);
                    V(s_shmem->input_file);
                    descomprimir_slot( socket, buffer_in , bytes_leidos, buffer_out, ibs, &obs);
                    /* Escribir y leer de un socket */
                    P(s_shmem->output_file);
                    lseek(_ofh, ofo, SEEK_SET);
                    write(_ofh, buffer_out, obs);
                    V(s_shmem->output_file);
            }
    } while (!fin);
```

Servidor

El funcionamiento de un servidor es como sigue. El proceso principal dispara cinco procesos, p_tabla que graba el diccionario en forma de tabla en memoria compartida, p_arbol que graba el diccionario en forma de árbol en memoria compartida, p_mng_contador el cual cada vez que recibe un requerimiento CONTADOR dispara un proceso contador (p_contador) y establece la comunicación entre este y el cliente, p_mng_compresor el cual cada vez que recibe un requerimiento COMPRESOR dispara un proceso compresor (p_compresor) y establece la comunicación entre este y el cliente y por último, p_mng_descompresor el cual cada vez que recibe un requerimiento DESCOMPRESOR dispara un proceso descompresor (p_descompresor) y establece la comunicación entre este y el cliente.

```
Servidor::

        Disparar({p_tabla, p_arbol, p_mng_contador, p_mng_compresor, p_mng_descompresor})
        Esperar la finalización de los procesos disparados

p_tabla::

        while(1)
                Recibir_codigo(sm->codigo);

p_arbol::
        while(1) {
                Recibir_header(header);
                Rearmar_arbol(header, sm->arbol);
        }

p_mng_contador::
        while (1) {
                Recibir un requerimiento de un contador
                Disparar un p_contador y comunicarlo con el cliente
        }

p_mng_compresor::
        while (1) {
                Recibir un requerimiento de un compresor
                Disparar un p_compresor y comunicarlo con el cliente
        }

p_mng_descompresor::
        while (1) {
                Recibir un requerimiento de un descompresor
                Disparar un p_descompresor y comunicarlo con el cliente
        }

p_contador::

        Inicializar el histograma local(histograma);
        while (hay bloques) {
                Recibir(bloque);
                Actualizar_histograma(bloque, histograma);
        }
        Enviar(histograma);
```

p_compresor::

```
while (hay bloques) {
        Recibir(bloque);
        Comprimir_bloque(bloque, sm->codigo, bloque_comprimido);
        Enviar(bloque_comprimido);
}
```

p_descompresor::

```
while (hay bloques) {
        Recibir(bloque);
        Descomprimir_bloque(bloque, sm->arbol, bloque_descomprimido);
        Enviar(bloque_descomprimido);
}
```

Observación:

Los procesos anteriores están bloqueados hasta recibir los datos.

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Código Comprimido - Inorder del Arbol de Huffman
Slots Comprimidos

- **Estructura de los Slots Comprimidos:**

Offset del slot descomprimido en el archivo original
Tamaño en bits del slot comprimido
Slot comprimido

Nota:

La estructura del archivo comprimido es idéntica a la estructura del archivo comprimido descripta en el capítulo anterior.

Capítulo 5

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Algoritmo lineal vs. algoritmo paralelizado monoprocesador

En los mejores casos los tiempos de compresión/descompresión del algoritmo paralelizado eran iguales a los del lineal, pero en la mayoria de los casos eran peor. En los casos en que era igual, se estaban explotando los tiempos muertos de E/S. Con esto concluimos que, por un lado se alcanzaba el objetivo por el cual se había paralelizado, es decir, realizar un uso productivo de la CPU en los "tiempos muertos" de E/S (cuando un proceso se bloquea en una I/O otros pueden realizar una utilización productiva de la CPU.), pero, por otro lado, el tiempo que se ganaba con este uso intensivo de la CPU no compenzaba el tiempo que se perdía con el manejo de procesos.

Algoritmo lineal vs. algoritmo paralelizado multiprocesador

Datos recogidos de las pruebas

Tablas

Archivos de Texto, 4 servers:

Tamaño (Mbytes)	Tiempo "Lineal"(Segs)	Tiempo "Paralelizado multiprocesador" (Segs)
1	13	8
3	39	24
5	65	43
7	95	59
9	126	77

Archivos Binarios, 4 servers:

Tamaño (Mbytes)	Tiempo "Lineal"(Segs)	Tiempo "Paralelizado multiprocesador" (Segs)
1	13	10
2	35	21
5	59	39
7	88	57
9	130	74
11	153	90
13	181	110

Tiempo en función de la cantidad de procesadores con archivo de 2 Mbytes :

Procesadores	Tiempo (Segs)	Hosts Utilizados
1	36	Algol
2	27	Algol + Ada
3	23	Algol + Ada + Ada2
4	21	Algol + Ada + Ada2 + Isis

Tiempo en función de la cantidad de procesadores con archivo de 7 Mbytes :

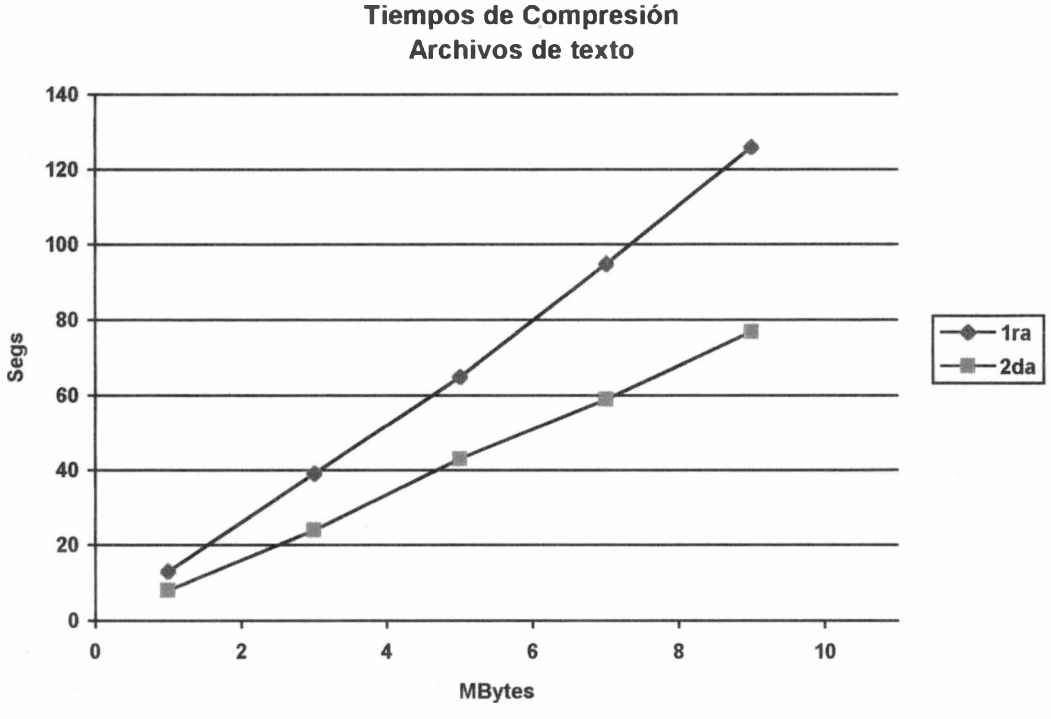
Procesadores	Tiempo (Segs)	Hosts Utilizados
1	98	Algol
2	72	Algol + Ada
3	63	Algol + Ada + Ada2
4	55	Algol + Ada + Ada2 + Isis

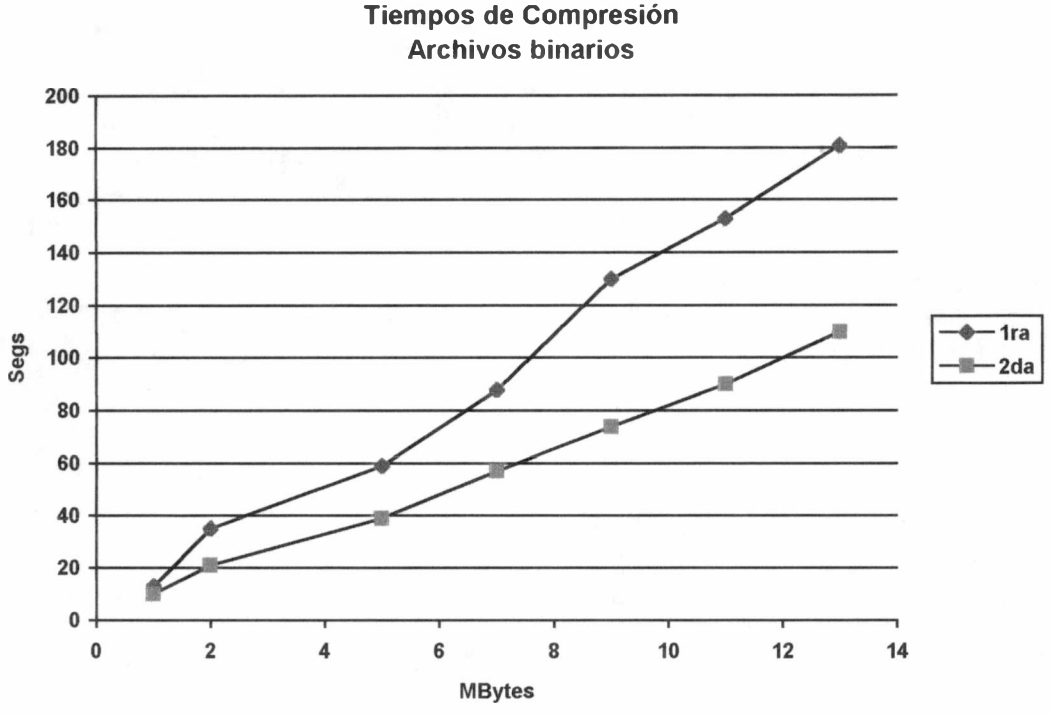
Observaciones:

La aplicacion cliente corría el el host algol.

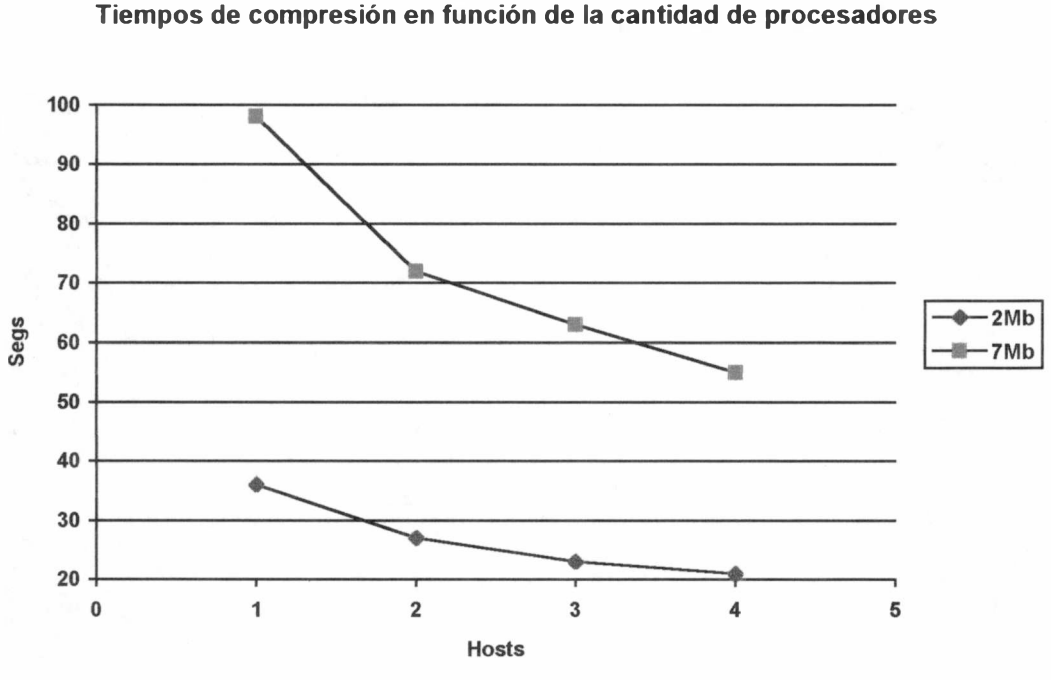
Las pruebas realizadas nos condujeron a disparar un servidor en el mismo host en que corría la aplicación cliente ya que esto mejoraba la performance.

Gráficos





Observación: En las versiones distribuidas se utilizaron 3 hosts (ALGOL,ADA e ISIS).



Conclusiones

Analizando las dos primeras tablas (sus gráficos asociados) puede verse que el algoritmo distribuido fué aproximadamente un 30% más rápido que el lineal.

Analizando las dos últimas (o el gráfico asociado) puede verse que si tenemos muchos procesadores disponibles y los datos a comprimir alcanzan varios Mbytes las mejoras son importantes. Es decir, que a medida que se agranda el número de hosts, los tiempos de compresión/descompresión, como era de esperarse, disminuyen.

El hecho de tener sólo cuatro procesadores diponibles nos imposibilitó analizar cual es el punto en donde la curva del último gráfico se estabiliza o comienza a crecer (si es que esto sucede). El problema que queríamos analizar es el siguiente. Supongamos que tenemos tantos procesadores como trabajos disponibles, si los servidores responden casi al mismo momento, se forma un cuello de botella en el cliente, ya que todos los procesos de comunicación están listos para trabajar. Por eso, en un caso como este habría que ver si lo que se gana distribuyendo totalmente los trabajos no se pierde en este cuello de botella. Quizás sea más eficiente disparar menos servidores con lo que la probabilidad de colisión entre los procesos que manejan la comunicación con los servidores sea baja.

Cabe aclarar que el algoritmo distribuido comprime un poco menos que el algoritmo lineal, ya que agrega unos pocos bytes por bloque para permitir el grabado en forma desordenada de los bloques comprimidos, pero esto no degrada demasiado los radios de compresión.

Capítulo 6

Huffman paralelo con múltiples diccionarios (paralelizado monoprocesador)

Introducción

Se realizó una segunda versión de los algoritmos paralelo-monoprocesador y paralelo-multiprocesador.

En este caso no se utilizó una codificación global sino que cada slot fué codificado en forma independiente de los demás. Es decir, se generó un código para cada slot de acuerdo a la estructura probabilística del mismo.

Una de las ventajas de esta implementación es que la compresión es más localizada por lo que los datos propiamente dichos serán mejor comprimidos. La desventaja es el espacio que se "desperdicia" con los headers ya que ahora se necesitará un header completo por cada slot. Nuestro trabajo, además de la implementación, es analizar cual de las estrategias es mejor o en que casos una es mejor que la otra.

En la compresión, otra ventaja de esta implementación con respecto a la descripta en el segundo capítulo es que este caso requiere menos control de procesos ya que con un solo lote de procesos cooperativos alcanza para realizar el trabajo, generar el histograma del bloque, comprimirlo y grabarlo. En la primera se disparaban dos lotes de procesos, uno para la generación del histograma y otro para la codificación. Notar que esto redujo la cantidad de accesos de lectura del archivo de entrada a la mitad.

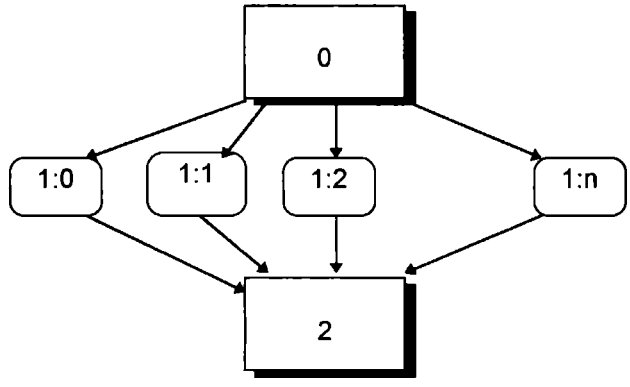
Otra ventaja es la reducción de datos compartidos necesarios, ya que los procesos no comparten el código para comprimir ni el árbol de decodificación para descomprimir. Asociadas a esta, están la reducción del número de semáforos necesarios, y más importante aún, la posibilidad de los servidores de atender a múltiples clientes simultáneamente.

Entre las desventajas, podemos mencionar que hay más procesamiento en el compresor y en el descompresor debido a que para cada bloque debe generarse el código (en la compresión) ó reconstruir el árbol de decodificación (en la descompresión) y que se graban más datos de control en el archivo comprimido.

El método de paralelización es el mismo que hemos estado utilizando en los casos anteriores.

Estructura

Compresor



0 (Main):

Setear el número de trabajos en función del tamaño del archivo y del tamaño del bloque de E/S.

Disparar los procesos compresores (procesos 1:i).

1: i (Compresores):

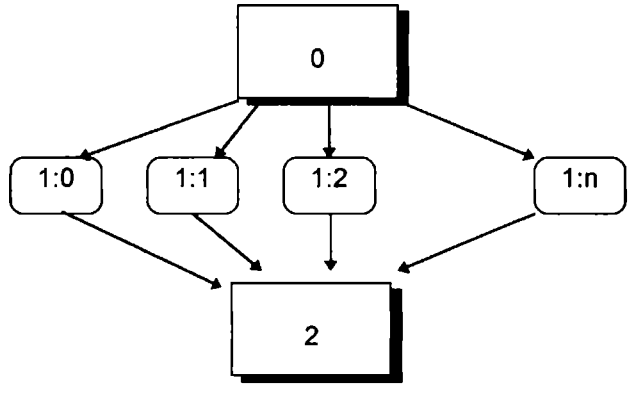
Mientras haya tareas pendientes

```
Leer_bloque(Archivo_de_entrada, &Bloque_entrada)
Crear_histograma(Bloque_entrada, &Histograma);
Armar_arbol(Histograma, &Arbol);
Armar_codigo_y_header(Arbol, &Header, &Codigo);
Grabar_header(&Bloque_salida, Header);
Codificar(Bloque_entrada, Codigo, &Bloque_salida);
Grabar(Archivo_de_salida, &Bloque_salida );
```

2 (Main):

Liberar recursos y finalizar.

Descompresor



0 (Main):

Setear el número de trabajos.

Disparar los procesos descompresores (procesos 1:i).

```
1: i (Descompresores):
        Mientras haya tareas pendientes
                Leer_bloque(Archivo_de_entrada,&Bloque_entrada);
                Rearmar_arbol(Bloque_entrada,&Arbol);
                Decodificar(Bloque_entrada, Arbol, &Bloque_salida);
                Grabar_en_posición (Archivo_de_salida,Bloque_entrada->offset_original,
        Bloque_salida);

2 (Main):
        Liberar recursos y finalizar.
```

Refinamiento

Compresor

```
        Codificar_en_paralelo(Archivo_de_entrada, Codigo, Archivo_de_salida);

Codificar_en_paralelo::
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_codificadores(Numero_de_procesos_codificadores);

Codificador[1..M] ::

        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        Tomar_desplazamiento_en_el_archivo_de_entrada(ifo);
                        Leer(Archivo_de_entrada, buffer_in, Tamano_de_slot);
                        V(s_shmem->input_file);
                        comprimir_slot( buffer_in , buffer_out, obs);
                        P(s_shmem->output_file);
                        write(_ofh, &ifo, sizeof(ifo));    /* offset del slot en el input file */
                        write(_ofh, &obs, sizeof(obs)); /* longitud en bits del slot comprimido */
                        write(_ofh, buffer_out, obs/8 + ((obs%8)+7)/8 );
                        V(s_shmem->output_file);
                }
        } while (!fin);

comprimir_slot ( buffer_in , buffer_out , obs )::
        Crear_histograma(buffer_in, &Histograma);
        Armar_arbol(Histograma, &Arbol);
        Armar_codigo_y_header(Arbol, &Header, &Codigo);
        Grabar_header(buffer_out, Header);
        Codificar(buffer_in, Codigo, buffer_out, & obs);
```

Descompresor

```
        Decodificar_en_paralelo(Archivo_de_entrada, Arbol, Archivo_de_salida);

Decodificar_en_paralelo::

        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_decodificadores(Numero_de_procesos_decodificadores);

Decodificador [1..M]::

        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        read(_ifh, &ofo, sizeof(ofo));
                        read(_ifh, &ibs, sizeof(ibs));
                        bytes_leidos = read(_ifh, buffer_in, (ibs/8) + ((ibs%8)+7)/8);
                        V(s_shmem->input_file);
                        descomprimir_slot( buffer_in , bytes_leidos, buffer_out, ibs, &obs);
                        P(s_shmem->output_file);
                        lseek(_ofh, ofo, SEEK_SET);
                        write(_ofh, buffer_out, obs);
                        V(s_shmem->output_file);
                }
        } while (!fin);

descomprimir_slot(buffer_in , bytes_leidos, buffer_out, ibs, &obs )::

        Leer_Header(buffer_in, &Header)
        Leer_Slot(buffer_in, &Slot)
        Rearmar_arbol(Header,&Arbol);
        Decodificar(Slot, Arbol, buffer_out , ibs , &obs );
```

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño de Slot
Slots Comprimidos

- **Estructura de los Slots Comprimidos:**

Tamaño del árbol
Arbol (Inorder del código)
Offset del slot descomprimido en el archivo original
Tamaño en bits de los datos comprimidos
Datos comprimidos

Capítulo 7

Huffman paralelo con múltiples diccionarios (paralelizado multiprocesador)

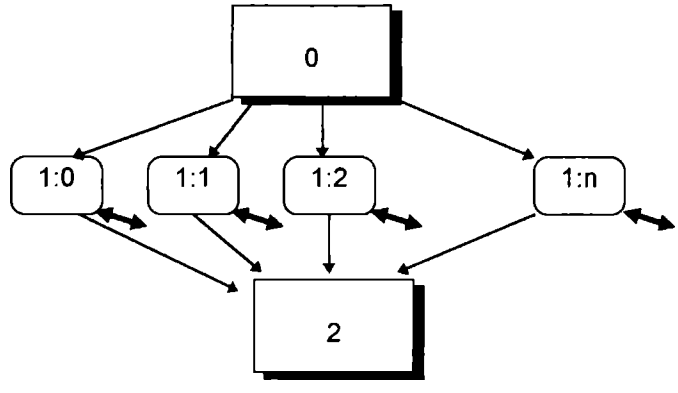
Introducción

La evolución de este nuevo algoritmo con respecto al del capítulo anterior es similar a la evolución del algoritmo del capítulo **4** (Huffman paralelo multiprocesador con diccionario único) respecto al algoritmo del capítulo 3 (Huffman paralelo monoprocesador con diciconario único), es decir se trasladaron los procesos codificadores y decodificadores a otros procesadores utilizando el paradigma cliente/servidor.

Las ventajas y desventajas entre este algoritmo y el del capítulo **4** son básicamente las mismas que mencionamos en el capitulo anterior. Una nueva ventaja que aparece en este caso es que el tráfico de datos a través de la red en el proceso de compresión se reduce aproximadamente a la mitad ya que la generación del código y la compresión se realiza en un solo paso. Otra ventaja es que ni en la compresión ni en la descompresión se debe distribuir el diccionario, con esto, además, se simplifica la estructura de los procesos servidores.

Estructura

Compresor



Notas:

——→ : fork/join

←→ : comunicación con servidor

0 (Main):

Setear número de tareas pendientes.
Disparar los procesos compresores (procesos 1:i).

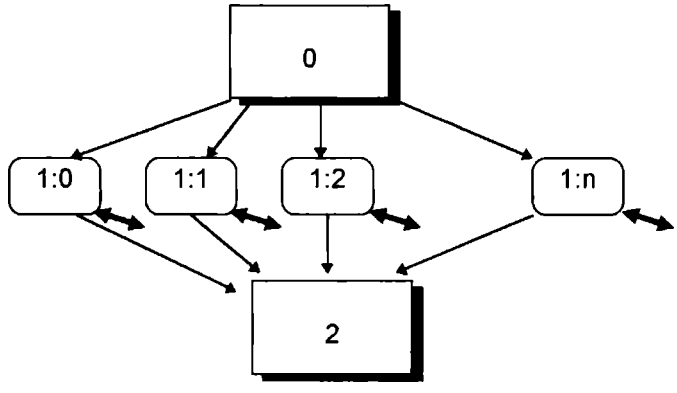
1: i (Compresores):

Mientras haya tareas pendientes
Leer slot.
Enviar slot al servidor asociado (compresor).
Recibir slot comprimido.
Grabar el slot comprimido en el archivo de salida.

2 (Main):

Liberar recursos y finalizar.

Descompresor



Notas:

⟶ : fork/join

⟷ : comunicación con servidor

0 (Main):

Setear el número de trabajos pendientes
Disparar los descompresores (procesos 1:i).

1: i (Descompresores):

Mientras haya tareas pendientes
Leer header de slot y slot comprimido.
Enviar slot comprimido al servidor asociado (descompresor).
Recibir slot descomprimido.
Grabar el slot descomprimido (en el offset original) en el archivo de salida.

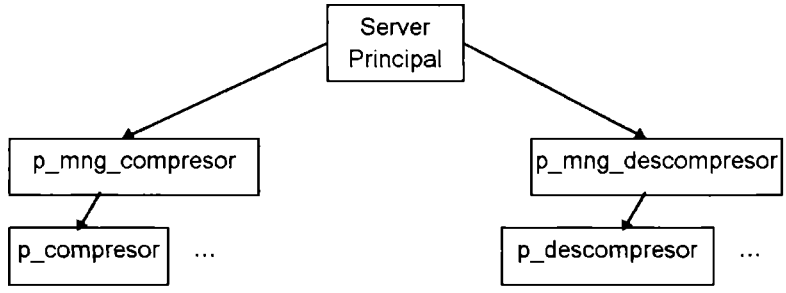
2 (Main):

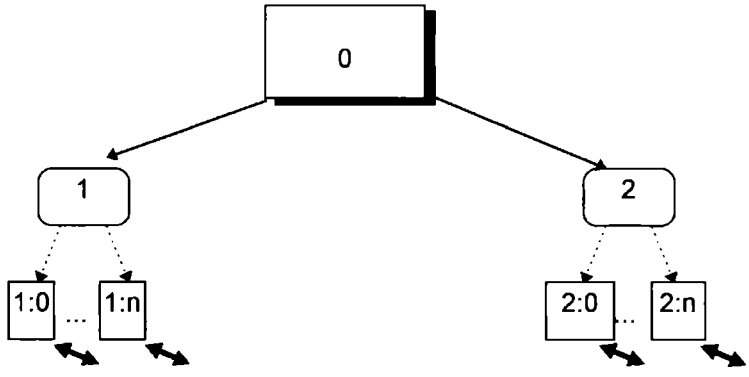
Liberar recursos y finalizar.

Servidor

Los servidores son los encargados de satisfacer los requerimientos de la aplicación principal. A diferencia del servidor descripto en el capítulo 3 ahora sólo se proveen dos clases de servicios (compresión y descompresión) lo que simplifica en gran medida la implementación del nuevo servidor y permite la atención de múltiples clientes simultáneamente.

El funcionamiento de un servidor es el siguiente. El proceso principal dispara dos procesos, el proceso p_mng_compresor el cual, cada vez que recibe un requerimiento COMPRESOR, dispara un proceso compresor (p_compresor) y establece la comunicación entre este y el cliente y el proceso p_mng_descompresor el cual, cada vez que recibe un requerimiento DESCOMPRESOR, dispara un proceso descompresor (p_descompresor) y establece la comunicación entre este y el cliente.





Notas:

——→ : fork/join

·····→ : fork/join + conección punto a punto con el cliente

←→ : comunicación con cliente

0 (Deamon principal):

Disparar procesos creadores de servidores (1..2).

2 (Servidor Creador de Compresores):

Recibir requerimiento.
Disparar Servidor Compresor y asociarlo al cliente.

2:i (Servidor Compresor):

Mientras no finalicen los requerimientos
Recibir un slot.
Generar el código de Huffman asociado a ese slot.
Comprimir slot.
Retornar el slot comprimido y el código utilizado al cliente.

3 (Servidor Creador de Descompresores):

Recibir requerimiento.
Disparar Servidor Descompresor y asociarlo al cliente.

3:i (Servidor Descompresor):

Mientras no finalicen los requerimientos
Recibir un slot.
Rearmar el árbol de decodificación.
Descomprimir slot.
Retornar el slot descomprimido al cliente.

Refinamiento

Compresor

```
        Codificar_en_paralelo(Archivo_de_entrada, Codigo, Archivo_de_salida);

Codificar_en_paralelo::
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_codificadores_com(Numero_de_procesos_codificadores_com);

Proceso_codificador_com[i:1..M] ::

        socket  = Abrir_socket(CODIFICADOR);
        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        Tomar_desplazamiento_en_el_archivo_de_entrada(ifo);
                        Leer(Archivo_de_entrada, buffer_in, Tamano_de_slot);
                        V(s_shmem->input_file);
                        comprimir_slot( socket, buffer_in , buffer_out, obs); /* Escribir y leer de un
socket */
                        P(s_shmem->output_file);
                        write(_ofh, &ifo, sizeof(ifo)); /* offset del bloque en el input file */
                        write(_ofh, &obs, sizeof(obs)); /* longitud en bytes del buffer */
                        write(_ofh, buffer_out, obs);
                        V(s_shmem->output_file);
                }
        } while(!fin);

comprimir_slot( socket, buffer_in , buffer_out, obs)
        writesocket(socket, buffer_in);
        readsocket(socket, buffer_out);
```

Descompresor

```
        Decodificar_en_paralelo(Archivo_de_entrada, Arbol, Archivo_de_salida);

Decodificar_en_paralelo::

        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_decodificadores_com(Numero_de_procesos_decodificadores_com);
```

```
Proceso_decodificador_com[i:1..M]::

        socket  = Abrir_socket( DECODIFICADOR );
        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        read(_ifh, &ofo, sizeof(ofo));
                        read(_ifh, &ibs, sizeof(ibs));
                        if (ibs) bytes_leidos = read(_ifh, buffer_in, ibs);
                        else Tratar_excepción;
                        V(s_shmem->input_file);
                        if (ibs) descomprimir_slot( socket, buffer_in , bytes_leidos, buffer_out, ibs,
&obs);
                        else Tratar_excepción;
                        P(s_shmem->output_file);
                        lseek(_ofh, ofo, SEEK_SET);
                        write(_ofh, buffer_out, obs);
                        V(s_shmem->output_file);
                }
        } while(!fin);

descomprimir_slot( socket, buffer_in , bytes_leidos, buffer_out, ibs, &obs)
        writesocket(socket, buffer_in);
        readsocket(socket, buffer_out);
```

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Slots Comprimidos

- **Estructura de los Slots:**

Offset del slot descomprimido en el archivo original
Tamaño en bytes del slot a distribuir
Slot a distribuir (sub-slot) o datos-excepción

Si ***Tamaño en bytes del slot a distribuir*** es cero (excepción: cuando el slot a comprimir tenía una o mas ocurrencias del mismo caracter) los campos son:

Repeticiones del caracter
Caracter

Si ***Tamaño en bytes del slot a distribuir*** es distinto de cero la estructura del slot a distribuir es:

Tamaño del arbol
Arbol (inorder del código)
Tamaño de los datos comprimidos
Datos comprimidos

Capítulo 8

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Datos recogidos de las pruebas

Tablas

Archivos Binarios, Tiempo de Compresión:

Tamaño (Mbytes)	Tiempo "Lineal"(Segs)	Tiempo "Paralelizado multiprocesador (Capítulo 4)" (Segs)	Tiempo "Paralelizado multiprocesador (Capítulo 7)" (Segs)
1	15	10	8
2	29	20	15
3	43	27	25
4	56	37	30
5	70	46	38
6	82	55	44
7	97	66	51
8	112	75	67
9	130	87	73
10	145	93	77

Archivos Binarios, Tiempo de Descompresión:

Tamaño (Mbytes)	Tiempo "Lineal"(Segs)	Tiempo "Paralelizado multiprocesador (Capítulo 4)" (Segs)	Tiempo "Paralelizado multiprocesador (Capítulo 7)" (Segs)
1	14	9	8
2	28	16	15
3	42	23	21
4	55	31	28
5	68	37	34
6	85	46	44
7	98	55	50
8	113	63	60
9	130	74	67
10	143	82	74

Archivos Binarios, Radios de compresión:

Tamaño (Mbytes)	"Lineal"	"Paralelizado multiprocesador (Capítulo 4)"	"Paralelizado multiprocesador (Capítulo 7)"
1	0.7710	0.7718	0.7169
2	0.7696	0.7704	0.7338
3	0.7703	0.7711	0.7478
4	0.7648	0.7656	0.7358
5	0.7606	0.7614	0.7286
6	0.7426	0.7434	0.7000
7	0.7410	0.7419	0.7048
8	0.7491	0.7499	0.7187
9	0.7557	0.7565	0.7281
10	0.7602	0.7610	0.7337

Observación: En las versiones ditribuidas se utilizaron los hosts ALGOL, ADA e ISIS.

Tiempo en función de la cantidad de procesadores con archivo de 5 Mbytes :

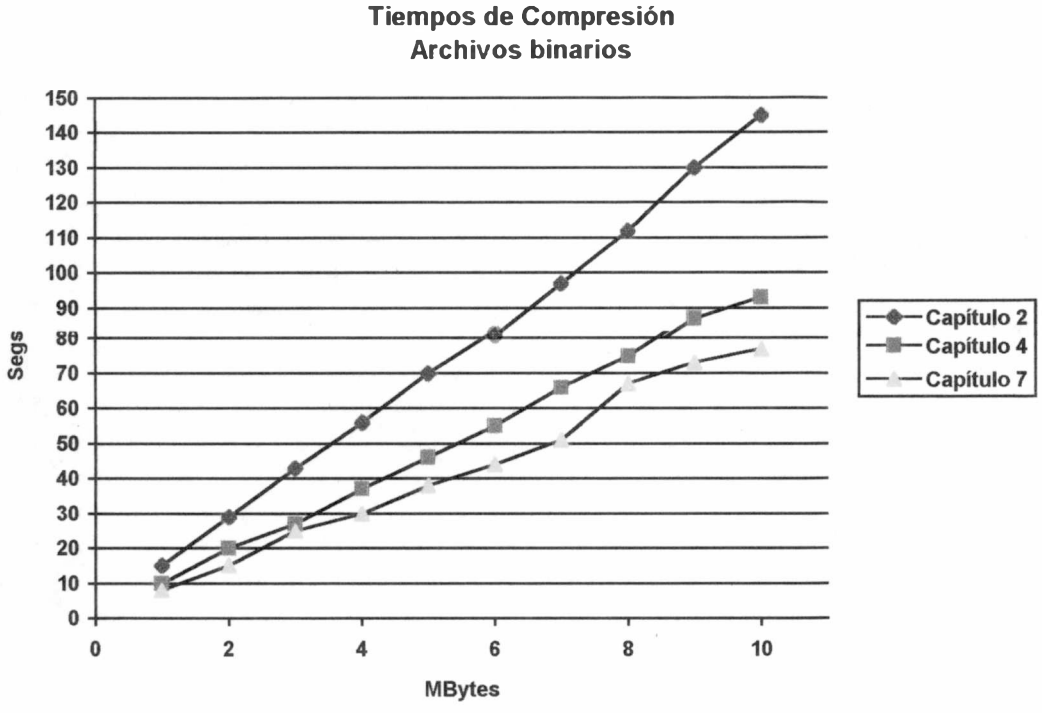
Procesadores	Tiempo de compresión (Segs)	Tiempo de descompresión (Segs)	Hosts Utilizados
1	74	63	Algol
2	48	44	Algol + Ada
3	38	34	Algol + Ada + Isis
4	31	29	Algol + Ada + Isis + Nahuel

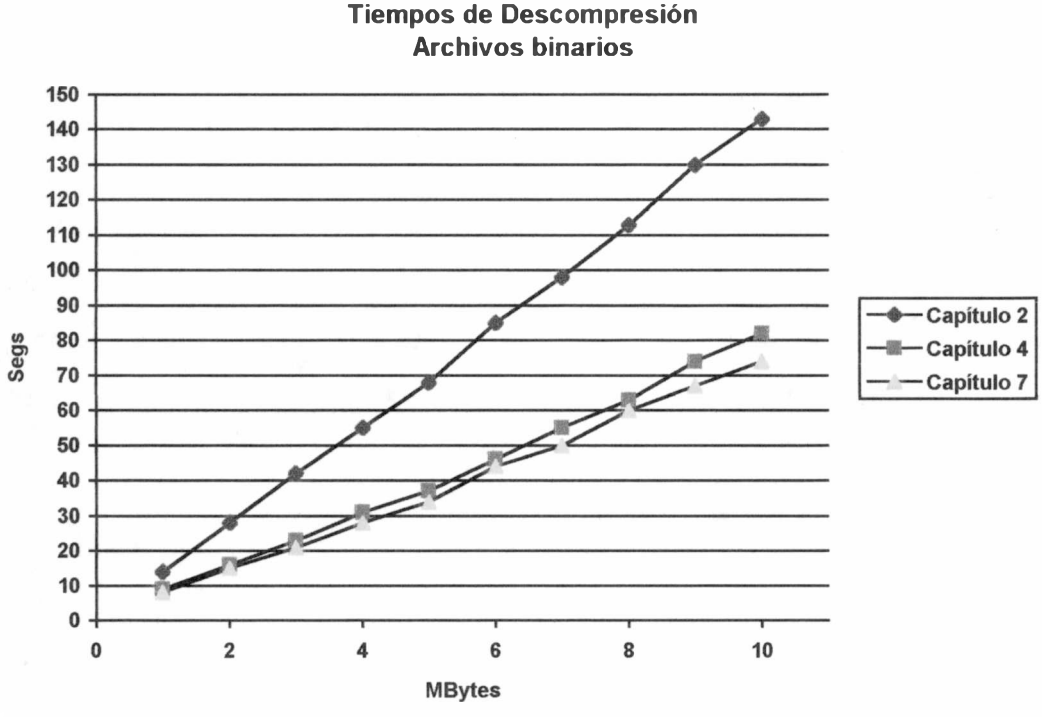
Observaciones:

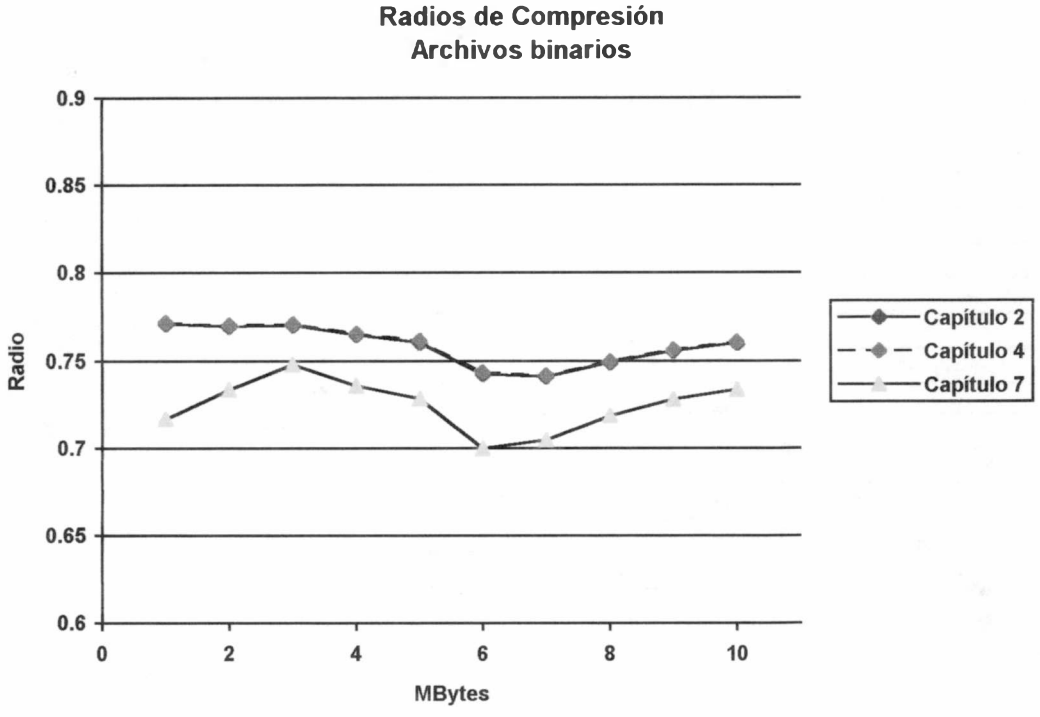
La aplicacion cliente corría en el host Algol.

Las pruebas realizadas nos conducieron a disparar un servidor en el mismo host que corría la aplicacion cliente ya que esto mejoraba la performance.

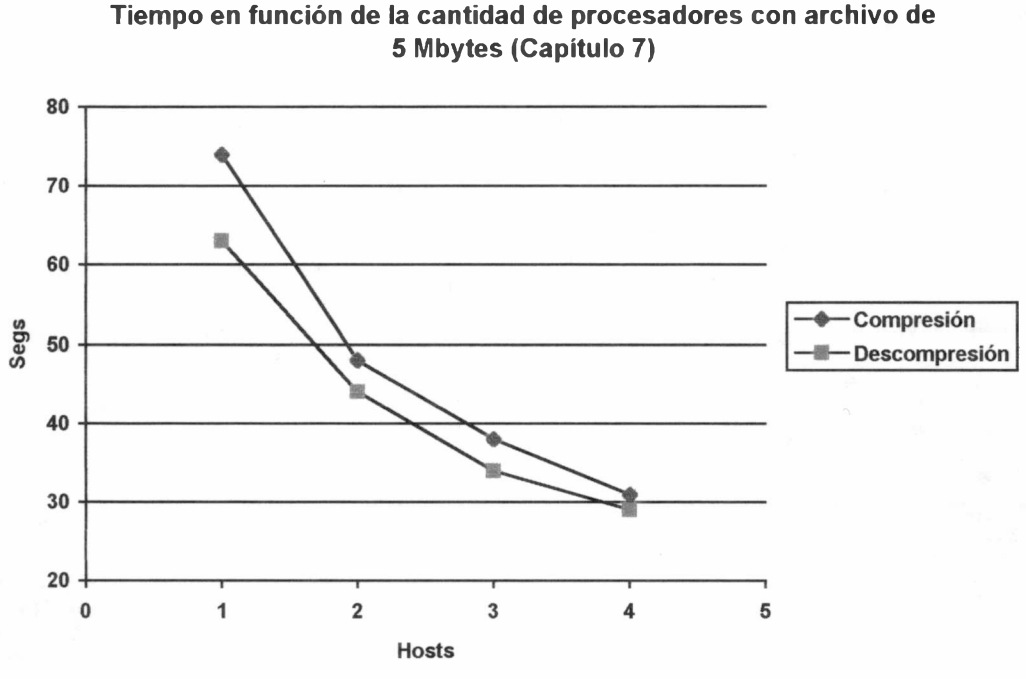
Gráficos


Tiempos de Compresión
Archivos binarios
Segs
MBytes
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
0
2
4
6
8
10
Capítulo 2
Capítulo 4
Capítulo 7


Tiempos de Descompresión
Archivos binarios
Segs
MBytes
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
0
2
4
6
8
10
Capítulo 2
Capítulo 4
Capítulo 7



Observación: En las versiones distribuidas se utilizaron 3 hosts (ALGOL, ADA e ISIS).



Conclusiones

Analizando la primer tabla o el gráfico asociado (ambos referidos a los tiempos de compresión) puede verse que el algoritmo distribuido descripto en el capítulo 4 fué aproximadamente un 30% más rápido que el lineal (correspondiente al capítulo 2). Mientras que la segunda versión distribuida (capítulo 7) fué, en promedio un 48% más rápida que la lineal.

Analizando la segunda tabla o el gráfico asociado (ambos referentes a los tiempos de descompresión) puede verse que el algoritmo distribuido descripto en el capítulo 4 fué aproximadamente un 44% más rápido que el lineal (descripto en el capítulo 2). Mientras que la versión descripta en el capítulo 7 fué, en promedio un 49% más rápida.

Analizando la tercer tabla o el gráfico asociado (ambos referentes a los radios de compresión alcanzados) puede verse que la primera versión del algoritmo distibuido comprime un poco menos que el algoritmo lineal, esto era de esperarse ya que utilizan el mismo código y que el distribuido agrega información para el manejo de los slots. También puede verse que la segunda versión comprime mas que el algoritmo lineal, en otras palabras, aunque al igual que la otra versión distribuida se agrega información para gestionar los slots, y aunque incrementa el tamaño de los headers de los slots debido a que debe almacenar un código por slot, esto es compensado por la localidad de la codificación.

En resumen, la segunda versión distribuida, descripta en el capítulo 7, comprime más que el algoritmo lineal y es más rápida que la primer versión distribuida.

Capítulo 9

Huffman distribuido con múltiples diccionarios (paralelizado multiprocesador utilizando PVM)

Introducción

Realizamos una implementación similar a la descripta en el capítulo 7 utilizando la herramienta de procesamiento paralelo en red PVM (Para más información referirse al apéndice H7). Elegimos dicha implementación por ser esta la más eficiente de todas las que habíamos realizado hasta el momento.

El objetivo de realizar esta nueva implementación era básicamente evaluar el desempeño de la herramienta.

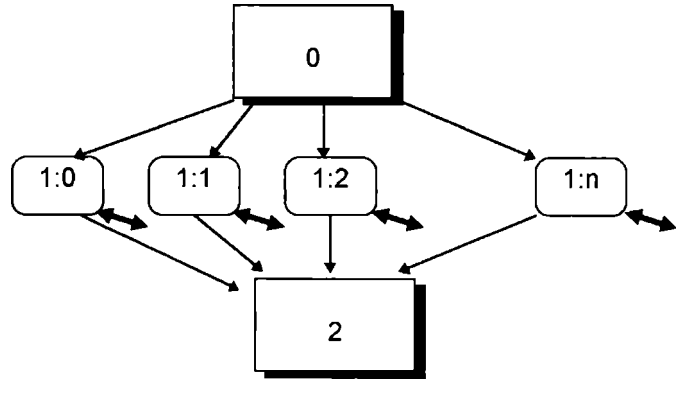
Una diferencia importante con la implementación del capítulo 7, es que en esta versión los servidores no son procesos independientes (equivalentes a los servidores usuales de UNIX) sino que son procesos esclavos disparados por el cliente cuando requiere sus servicios. Nacen cuando el cliente lo requiere y terminan una vez realizada la tarea por la cual fueron creados.

Otra diferencia, no independiente de la anterior, es que en esta versión, parte de la gestión de procesos, y la totalidad de la comunicación remota son provistas por la herramienta.

La implementación resultante fue más clara, corta y simple que la del capítulo 7.

Estructura

Compresor



Notas:

——→ : fork/join

←→ : comunicación con servidor

0 (Main):

Setear número de tareas pendientes.

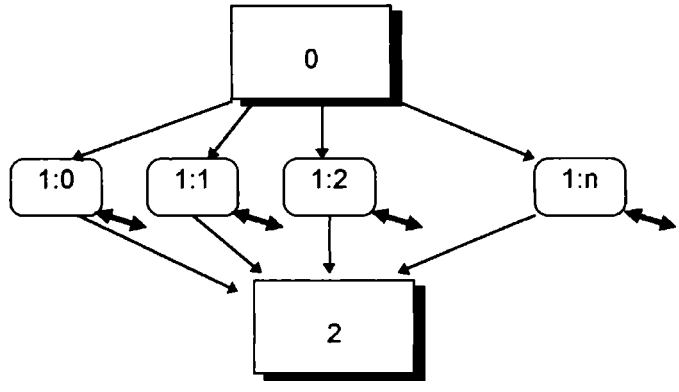
Disparar los procesos compresores (procesos 1:i).

1: i (Compresores):

Creación de un servidor de compresión (equivalente a los procesos compresores [p_compresor] de los servidores de las versiones distribuidas)
Mientras haya tareas pendientes
Leer slot.
Enviar slot al servidor asociado (compresor).
Recibir slot comprimido.
Grabar el slot comprimido en el archivo de salida.

2 (Main):
Liberar recursos y finalizar.

Descompresor



Notas:
: fork/join
: comunicación con servidor

0 (Main):
Setear el número de trabajos pendientes
Disparar los descompresores (procesos 1:i).

1: i (Descompresores):
Creación de un servidor de descompresión (equivalente a los procesos descompresores [p_descompresor] de los servidores de las versiones distribuidas)
Mientras haya tareas pendientes
Leer header de slot y slot comprimido.
Enviar slot comprimido al servidor asociado (descompresor).
Recibir slot descomprimido.
Grabar el slot descomprimido (en el offset original) en el archivo de salida.

2 (Main):
Liberar recursos y finalizar.

Servidor

En este caso los servicios son provistos directamente por los procesos compresores y descompresores de los servidores anteriores y la creación y la comunicación son temas solucionados transparentemente por el PVM. Cuando un cliente necesita los servicios de un servidor le pide al PVM que dispare un proceso servidor y que resuelva la comunicación entre ellos.

Refinamiento

Compresor

```
        Codificar_en_paralelo(Archivo_de_entrada, Codigo, Archivo_de_salida);

Codificar_en_paralelo::
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_codificadores_com(Numero_de_procesos_codificadores_com);

Proceso_codificador_com[i:1..M] ::

        sid = pvm_spawn(SERVIDOR_COMPRESION);
        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        Tomar_desplazamiento_en_el_archivo_de_entrada(ifo);
                        Leer(Archivo_de_entrada, buffer_in, Tamano_de_slot);
                        V(s_shmem->input_file);
                        comprimir_slot( sid, buffer_in , buffer_out, obs);
                        P(s_shmem->output_file);
                        write(_ofh, &ifo, sizeof(ifo)); /* offset del bloque en el input file */
                        write(_ofh, &obs, sizeof(obs)); /* longitud en bytes del buffer */
                        write(_ofh, buffer_out, obs);
                        V(s_shmem->output_file);
                }
        } while(!fin);

comprimir_slot( sid, buffer_in , buffer_out, obs)
        pvm_send(sid, buffer_in);
        pvm_recv(sid, buffer_out);
```

Descompresor

```
        Decodificar_en_paralelo(Archivo_de_entrada, Arbol, Archivo_de_salida);

Decodificar_en_paralelo::
```

```
        Setear_trabajos_pendientes_en_memoria_compartida(Tamano_del_archivo_de_entrada,
Tamano_de_slot);
        Disparar_procesos_decodificadores_com(Numero_de_procesos_decodificadores_com);

Proceso_decodificador_com[i:1..M]::

        sid = pvm_spawn(SERVIDOR_DESCOMPRESION);
        fin = FALSE;
        do {
                P(s_shmem->trabajos_pendientes);
                if (!Hay_trabajos_pendientes) fin = TRUE;
                else Decrementar_trabajos_pendientes;
                V(s_shmem->trabajos_pendientes);
                if (!fin) {
                        P(s_shmem->input_file);
                        read(_ifh, &ofo, sizeof(ofo));
                        read(_ifh, &ibs, sizeof(ibs));
                        if (ibs) bytes_leidos = read(_ifh, buffer_in, ibs);
                        else Tratar_excepción;
                        V(s_shmem->input_file);
                        if (ibs) descomprimir_slot( sid, buffer_in , bytes_leidos, buffer_out, ibs, &obs);
                        else Tratar_excepción;
                        P(s_shmem->output_file);
                        lseek(_ofh, ofo, SEEK_SET);
                        write(_ofh, buffer_out, obs);
                        V(s_shmem->output_file);
                }
        } while(!fin);

descomprimir_slot( sid, buffer_in , bytes_leidos, buffer_out, ibs, &obs)
        pvm_send(sid, buffer_in);
        pvm_recv(sid, buffer_out);
```

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Slots Comprimidos

- **Estructura de los Slots:**

Offset del slot descomprimido en el archivo original
Tamaño en bytes del slot a distribuir
Slot a distribuir (sub-slot) o datos-excepción

Si ***Tamaño en bytes del slot a distribuir*** es cero (excepción: cuando el slot a comprimir tenía una o mas ocurrencias de un único caracter) los campos son:

Repeticiones del caracter
Caracter

Si ***Tamaño en bytes del slot a distribuir*** es distinto de cero la estructura del slot a distribuir es:

Tamaño del arbol
Arbol (inorder del código)
Tamaño de los datos comprimidos
Datos comprimidos

Capítulo 10

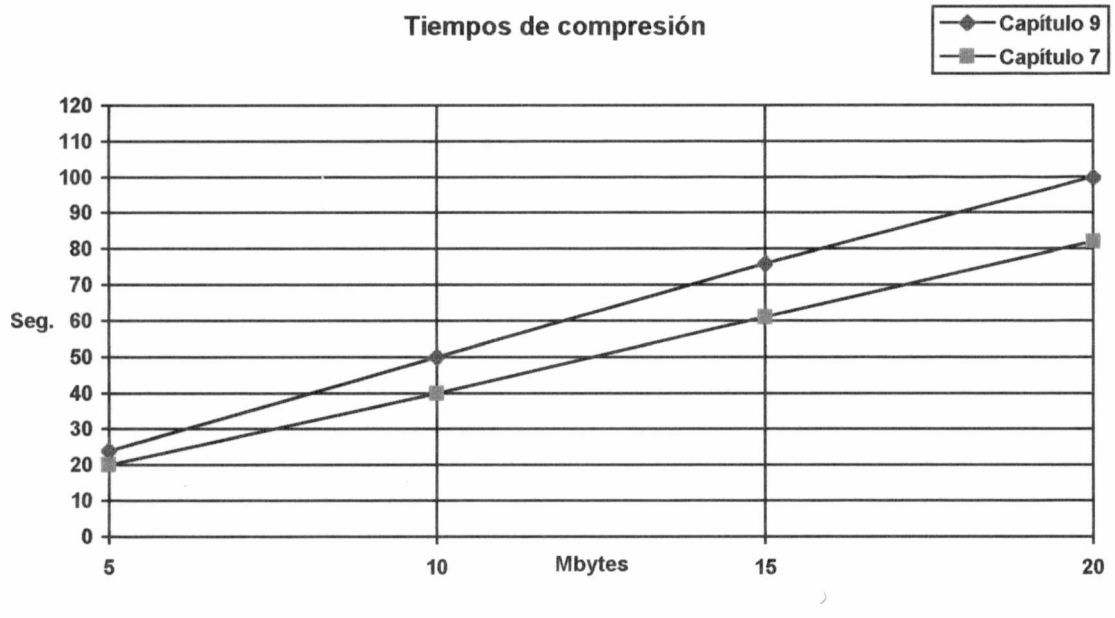
Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Datos recogidos de las pruebas

Tablas

SOCKETS				PVM			
Tamaños (bytes)		Tiempos (seg.)		Tamaños (bytes)		Tiempos (seg.)	
Original	*Comp.*	*Comp.*	*Descom.*	*Original*	*Comp.*	*Comp.*	*Descom.*
20971520	16014427	82	68	20971520	16014427	100	89
15728640	11981695	61	52	15728640	11981695	76	68
10485760	7873631	40	34	10485760	7873631	50	43
5242880	3914234	20	17	5242880	3914234	24	22

Gráficos



Observación:

Los resultados experimentales del algoritmo se obtuvieron utilizando las mismas computadoras con similar carga de trabajo y obviamente con las mismas entradas, estas PC's tienen todas la misma arquitectura (Intel).

Conclusiones

Como se puede apreciar a partir de la tabla o del gráfico, la implementación del capítulo 7 es más rápida. Además en el gráfico se puede apreciar fácilmente como se van separando las curvas a medida que se aumenta el tamaño del archivo a comprimir. Esto implica que el PVM no sólo introduce un retardo inicial debido a la creación de los procesos servidores subordinados, sino que existe un overhead en las primitivas de comunicación que se mantienen a lo largo del proceso.

Probablemente el overhead en la comunicación se deba a la portabilidad entre distintas arquitecturas que provee la herramienta.

Dejando de lado la velocidad de ejecución, con PVM tenemos mayor abstracción en relación a las comunicaciones entre los diferentes procesos y en relación a la distribución de los procesos en hosts. Esto permite concentrarse en el algoritmo a desarrollar (el problema a resolver), delegando el problema de la comunicación interprocesos y del control de procesos a la herramienta.

Capítulo 11

Huffman distribuido con múltiples diccionarios (paralelizado multiprocesador utilizando MPI)

Introducción

Nuestro objetivo era realizar una implementación similar a la descripta en el capítulo 7 utilizando una interface de pasaje de mensajes para una red de computadores posiblemente heterogéneos llamada MPI (Para más información referirse al apéndice H8). Elegimos dicha implementación por ser esta la más eficiente de todas las que habíamos realizado hasta el momento.

Al igual que en el caso del PVM, nuestra intención era básicamente evaluar el desempeño de la herramienta.

Cuando comenzamos a trabajar nos encontramos con que esta implementación de la herramienta no era thread safety como lo informaba la documentación que tenía asociada (nosotros trabajamos con wheigth process, no lo probamos con light process). Esto nos llevó a cambiar la estructura de los clientes en la que nos estábamos basando para adecuarla a las posibilidades de la herramienta.

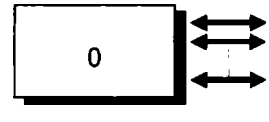
La diferencia es la siguiente. Los nuevos clientes no disparan procesos para resolver eficientemente la comunicación con los servidores, sino que, a través de una facilidad provista por la herramienta ellos mismos pueden atender selectivamente las respuestas de los servidores.

El modo de funcionamiento de los nuevos clientes es el siguiente. Reparten un trabajo para cada servidor secuencialmente y luego se quedan esperando a que alguno termine de procesar su bloque. Cuando uno termina, se graba el bloque devuelto y, si quedan bloques por procesar, se le envía uno al servidor libre. Si no quedan bloques por procesar se le avisa al servidor que puede terminar. Si quedan servidores trabajando, se espera por la finalización de alguno y se vuelve al paso anterior. Si no quedan servidores trabajando se finaliza el proceso.

La implementación resultante fue clara, corta y simple y utilizaba menos recursos en la máquina del cliente ya que N servidores eran "manejados" por un solo cliente (a diferencia de las implementaciones de los capítulos 7 y 9 en las cuales se utilizaban N+1 procesos).

Estructura

Compresor



Nota:

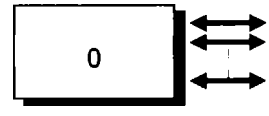
⟷ : comunicación con servidor

0 (Main):

Setear número de tareas pendientes.
Inicializar los servidores (enviarles un bloque).

Mientras haya servidores trabajando
Esperar la finalización de un servidor
Grabar el slot comprimido en el archivo de salida.
Si hay tareas pendientes
Leer slot.
Enviar slot al servidor que terminó.
Si no, liberar el servidor que finalizó (avisarle que puede terminar)
Liberar recursos y finalizar.

Descompresor



Nota:
⟷ : comunicación con servidor

0 (Main):
Setear número de tareas pendientes.
Inicializar los servidores (enviarles un bloque).
Mientras haya servidores trabajando
Esperar la finalización de un servidor
Grabar el slot descomprimido en el offset original en el archivo de salida.
Si hay tareas pendientes
Leer slot.
Enviar slot al servidor que terminó.
Si no, liberar el servidor que finalizó (avisarle que puede terminar)
Liberar recursos y finalizar.

Servidor

El funcionamiento de los servidores es similar al del caso PVM. Existen dos tipos de servidores (compresor y descompresor) que aceptan bloques del cliente, procesan el bloque y se lo retornan. Así hasta que el cliente no tenga más bloques para procesar.

Refinamiento

Compresor

```
servidores_trabajando = Asignar_tarea_inicial_a_los_servidores();
while (1) {
        Esperar_respuesta_de_algun_servidor(&servidor, &bloque);
        Grabar_el_bloque_recibido(bloque, archivo_de_salida);
        if (!eof(archivo_de_entrada)) Enviar_bloque_a_servidor(servidor,
archivo_de_entrada);
        else {
                Avisarle_al_servidor_que_puede_finalizar(servidor);
```

```
                    if (!--servidores_trabajando) break;
            }
        }
```

Descompresor

```
        servidores_trabajando = Asignar_tarea_inicial_a_los_servidores();
        while (1) {
            Esperar_respuesta_de_algun_servidor(&servidor, &bloque);
            Grabar_el_bloque_recibido(bloque, archivo_de_salida);
            if (!eof(archivo_de_entrada)) Enviar_bloque_a_servidor(servidor,
archivo_de_entrada);
            else {
                Avisarle_al_servidor_que_puede_finalizar(servidor);
                if (!--servidores_trabajando) break;
            }
        }
```

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Slots Comprimidos

- **Estructura de los Slots:**

Offset del slot descomprimido en el archivo original
Tamaño en bytes del slot a distribuir
Slot a distribuir (sub-slot) o datos-excepción

Si ***Tamaño en bytes del slot a distribuir*** es cero (excepción: cuando el slot a comprimir tenía una o mas ocurrencias de un único caracter) los campos son:

Repeticiones del caracter
Caracter

Si ***Tamaño en bytes del slot a distribuir*** es distinto de cero la estructura del slot a distribuir es:

Tamaño del arbol
Arbol (inorder del código)
Tamaño de los datos comprimidos
Datos comprimidos

Capítulo 12

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

En este capítulo vamos a presentar los resultados del análisis de la performance de las implementaciones correspondientes a los capítulos 2 (Huffman lineal), 7 (Huffman distribuido (multiprocesador) utilizando C aumentado con las librerías estandar), 9 (Huffman distribuido multiprocesador utilizando PVM) y 11 (Huffman distribuido multiprocesador utilizando MPI).

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

Datos recogidos de las pruebas

Tablas

Lineal

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	691706	12.84	11.89
2	2097152	1397560	23.05	23.79
3	3145728	2107439	37.37	37.73
4	4194304	2803582	51.41	50.16
5	5542880	3509536	66.16	62.66
6	6291456	4197618	79.02	74.76
7	7340032	4926113	94.45	87.28
8	8388608	5650498	106.43	101.49
9	9437184	6357718	119.98	113.38
10	10485760	7067018	132.83	127.64

Tabla 1

Distribuida (C estandar)

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	5.03	6.50
2	2097152	1430653	12.19	11.04
3	3145728	2154364	16.2	17.36
4	4194304	2856867	22.70	21.56
5	5542880	3445067	29.17	29.57
6	6291456	4032455	34.47	35.58
7	7340032	4775894	40.56	41.43
8	8388608	5517596	46.84	46.60
9	9437184	6264067	52	53.22
10	10485760	6999398	58.87	59.15

Tabla 2

Distribuida (PVM)

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	12.99	11.92
2	2097152	1430653	25.40	23.47
3	3145728	2154364	36.59	35.10
4	4194304	2856867	49.43	46.32
5	5542880	3445067	60.24	58.3
6	6291456	4032455	71.29	70.97
7	7340032	4775894	82.97	81.31
8	8388608	5517596	95.16	93.28
9	9437184	6264067	105.98	104.1
10	10485760	6999398	119.61	118.73

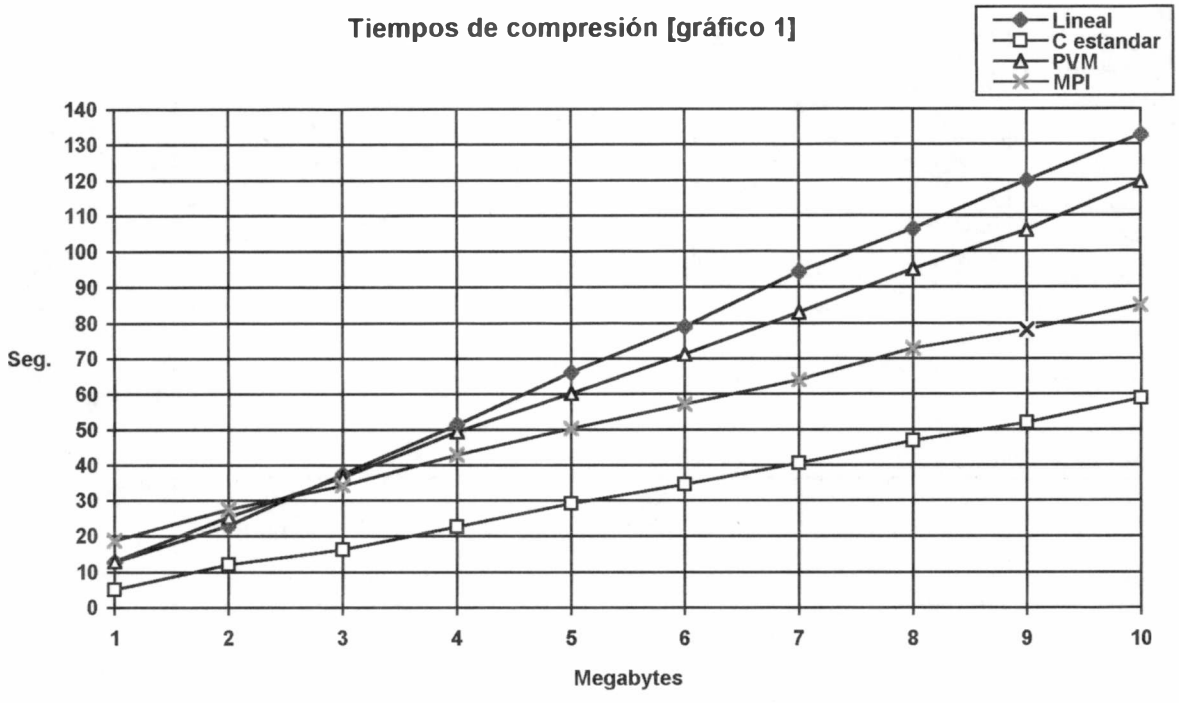
Tabla 3

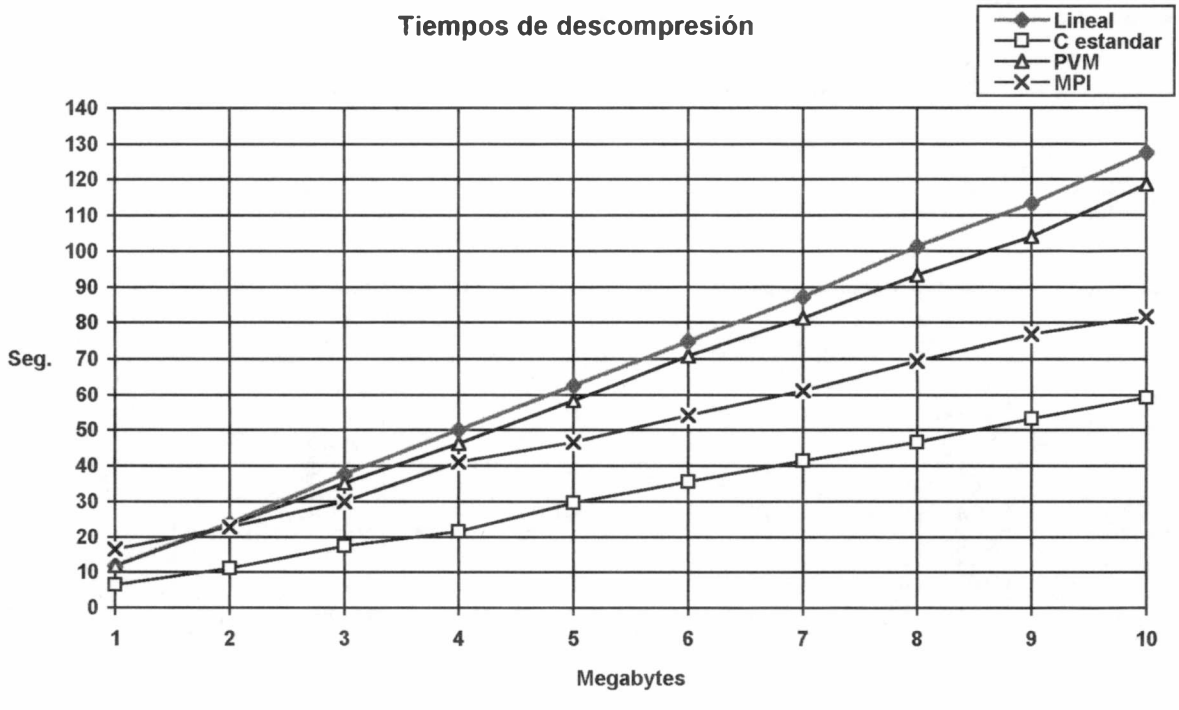
Distribuida (MPI)

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	18.75	16.5
2	2097152	1430653	27.43	22.84
3	3145728	2154364	34.25	29.95
4	4194304	2856867	42.95	41.12
5	5542880	3445067	50.28	46.62
6	6291456	4032455	57.20	54.22
7	7340032	4775894	64.07	61.28
8	8388608	5517596	72.97	69.47
9	9437184	6264067	78.22	76.82
10	10485760	6999398	85.03	81.86

Tabla 4

Gráficos


Tiempos de compresión [gráfico 1]
Lineal
C estandar
PVM
MPI
Seg.
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1
2
3
4
5
6
7
8
9
10
Megabytes


Tiempos de descompresión
Lineal
C estandar
PVM
MPI
Seg.
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1
2
3
4
5
6
7
8
9
10
Megabytes

Conclusiones

Analizando las tablas y los gráficos puede apreciarse que la solución más rápida fué la correspondiente a la descripta en el capítulo 7. Que le sigue la solución correspondiente al capítulo 11 (MPI), luego la del capítulo 9 (PVM), y por último, la solución lineal.

Cabe destacar que la diferencia de las pendientes de las curvas correspondientes a las soluciones distribuidas se debe básicamente al retardo introducido por las implementaciones particulares de la comunicación entre los procesos. Este retardo se debe básicamente a la portabilidadad entre distintas plataformas que proveen el MPI y el PVM (conversión a un formato estandar y recreación al formato adecuado a la arquitectura del anfitrión).

También se debería notar (aunque no es visible directamente en los gráficos ni en las tablas) que la creación de los procesos probablemente no tarde lo mismo en las diferentes implementaciones distribuidas, esto generaría una diferencia inicial constante entre las curvas.

También puede apreciarse que la pendiente de la curva correspondiente a la solución MPI es aproximadamente igual a la pendiente de la solución introducida en el capítulo 7. Esto significa que la eficiencia de la comunicación de la solución MPI es casi equivalente a la de la implementación del capítulo 7.

Esto nos permite concluir que el MPI resuelve muy bien la comunicación pero tiene un gran retardo inicial (probablemente debido a la creación de los procesos y a la inicialización de la comunicación).

Otra cosa que puede verse es que la pendiente de la curva correspondiente a la solución PVM es aproximadamente igual a la pendiente de la solución lineal. Esto significa que la eficiencia de la comunicación de la solución PVM es bastante pobre y necesita un gran volumen de datos para despegarse de la solución lineal.

Capítulo 13

Huffman distribuido con múltiples diccionarios (paralelizado multiprocesador utilizando espera selectiva)

Introducción

A partir de la implementación realizada en el capítulo 11 surgió una variante interesante de los clientes. Utilizar espera selectiva en vez de múltiples procesos para resolver la comunicación con los servidores.

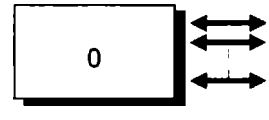
A pesar de que la implementación del capitulo 11 fué menos eficiente que la del 7 la estructura de los clientes nos pareció que podía llegar a ser más eficiente ya que, a diferencia de la otra, con un sólo proceso atendía selectivamente las respuestas de los servidores. Debido a esto reescribimos la solución sin utilizar la herrramienta (utilizando las librerías estandar). Esto nos permitiría comparar el nuevo esquema del cliente contra la implementación del capítulo 7 (que hasta este momento era la mejor alternativa). Esta nueva implementación era necesaria para un correcto análisis ya que en las comparaciones del capítulo 12 influía en los resultados el overhead que introducía la herramienta y esto nos impedía realizar comparaciones puramente estructurales.

La ventaja de esta implementación con respecto a la del capítulo 7 es la utilización de un sólo proceso para realizar el trabajo. Esto evitó las tareas relacionadas con la creación, comunicación y sincronización de procesos. Lo cual simplificó el código y permitió la no utilización de recursos tales como los semáforos y los segmentos de memoria compartida.

Nota:
Para resolver el problema de la espera selectiva utilizamos la system call select.

Estructura

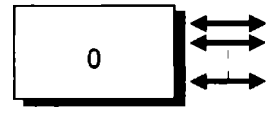
Compresor



Nota:
⟷ : comunicación con servidor

0 (Main):
Setear número de tareas pendientes.
Inicializar los servidores (enviarles un bloque).
Mientras haya servidores trabajando
Esperar la finalización de un servidor
Grabar el slot comprimido en el archivo de salida.
Si hay tareas pendientes
Leer slot.
Enviar slot al servidor que terminó.
Si no, liberar el servidor que finalizó (avisarle que puede terminar)
Liberar recursos y finalizar.

Descompresor



Nota:

 : comunicación con servidor

0 (Main):

```
Setear número de tareas pendientes.
Inicializar los servidores (enviarles un bloque).
Mientras haya servidores trabajando
        Esperar la finalización de un servidor
        Grabar el slot descomprimido en el offset original en el archivo de salida.
        Si hay tareas pendientes
                Leer slot.
                Enviar slot al servidor que terminó.
        Si no, liberar el servidor que finalizó (avisarle que puede terminar)
Liberar recursos y finalizar.
```

Servidor

Se utilizó el servidor descripto en el capítulo 7.

Refinamiento

Compresor

```
servidores_trabajando = Asignar_tarea_inicial_a_los_servidores();
while (1) {
        Esperar_respuesta_de_algun_servidor(&servidor, &bloque);
        Grabar_el_bloque_recibido(bloque, archivo_de_salida);
        if (!eof(archivo_de_entrada)) Enviar_bloque_a_servidor(servidor,
archivo_de_entrada);
        else {
                Avisarle_al_servidor_que_puede_finalizar(servidor);
                if (!--servidores_trabajando) break;
        }
}
```

Descompresor

```
servidores_trabajando = Asignar_tarea_inicial_a_los_servidores();
while (1) {
        Esperar_respuesta_de_algun_servidor(&servidor, &bloque);
```

```
                Grabar_el_bloque_recibido(bloque, archivo_de_salida);
                if (!eof(archivo_de_entrada)) Enviar_bloque_a_servidor(servidor,
archivo_de_entrada);
                else {
                        Avisarle_al_servidor_que_puede_finalizar(servidor);
                        if (!--servidores_trabajando) break;
                }
        }
```

Estructura del archivo comprimido

- **Estructura del archivo comprimido:**

Tamaño del Archivo Original
Tamaño del Slot
Slots Comprimidos

- **Estructura de los Slots:**

Offset del slot descomprimido en el archivo original
Tamaño en bytes del slot a distribuir
Slot a distribuir (sub-slot) o datos-excepción

Si ***Tamaño en bytes del slot a distribuir*** es cero (excepción: cuando el slot a comprimir tenía una o mas ocurrencias de un único caracter) los campos son:

Repeticiones del caracter
Caracter

Si ***Tamaño en bytes del slot a distribuir*** es distinto de cero la estructura del slot a distribuir es:

Tamaño del arbol
Arbol (inorder del código)
Tamaño de los datos comprimidos
Datos comprimidos

Capítulo 14

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

En este capítulo se analizó el desempeño de tres algoritmos, el algoritmo descripto en el capítulo 7 (implementado en C utilizando librerías estandar y multiprocesamiento en el cliente), el descripto en el capítulo 11 (implementado en C utilizando MPI y espera selectiva en el cliente) y el descripto en el capítulo anterior (implementado en C utilizando las librerías estandar y clientes con espera selectiva).

El realizar una implementación en C utilizando las librerías estandar cuya estructura sea similar a la de la implementación MPI nos iba a permitir atacar dos puntos que a nuestro criterio habían quedado pendientes, por un lado comparar la eficiencia de la espera selectiva frente al multiprocesamiento (Capítulo 13 vs. Capítulo 7) objetivamente (sin overheads introducidos por herramientas no estandar, léase MPI) y por otro lado comparar la version MPI con una versión C estandar de estructura similar (Capítulo 11 vs. Capítulo 13) lo que nos permitiría evaluar con más precisión el desempeño de la herramienta.

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

Datos recogidos de las pruebas

Tablas

Distribuida (Capítulo 7)

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	5.03	6.50
2	2097152	1430653	12.19	11.04
3	3145728	2154364	16.2	17.36
4	4194304	2856867	22.70	21.56
5	5542880	3445067	29.17	29.57
6	6291456	4032455	34.47	35.58
7	7340032	4775894	40.56	41.43
8	8388608	5517596	46.84	46.60
9	9437184	6264067	52	53.22
10	10485760	6999398	58.87	59.15

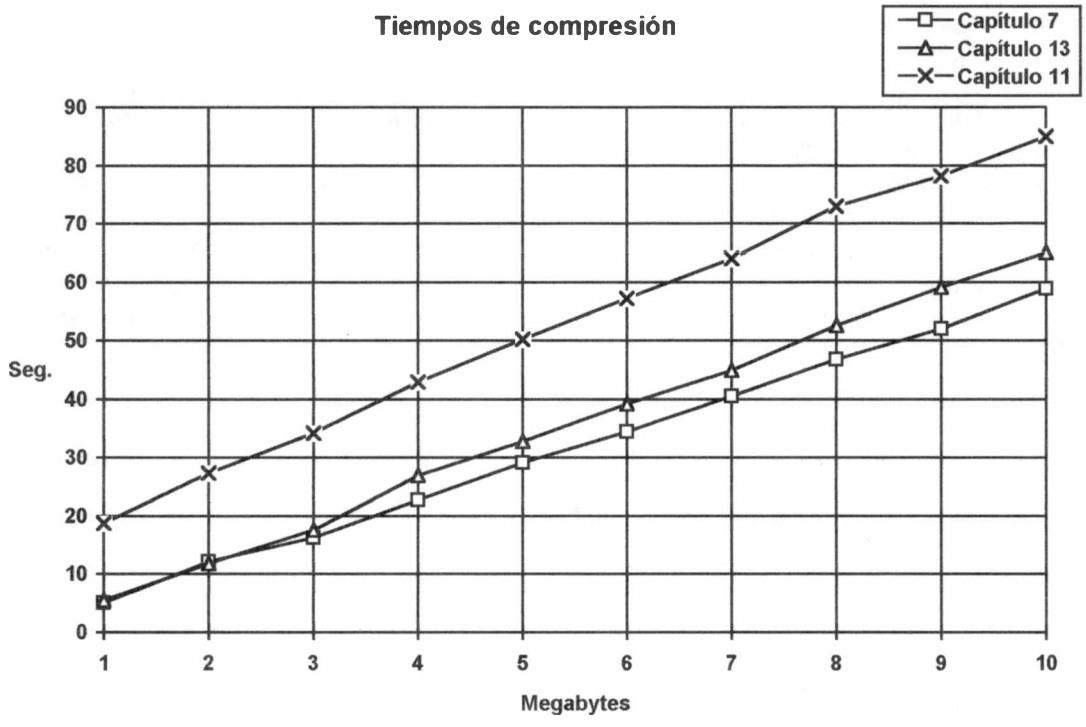
Distribuida (Capítulo 13)

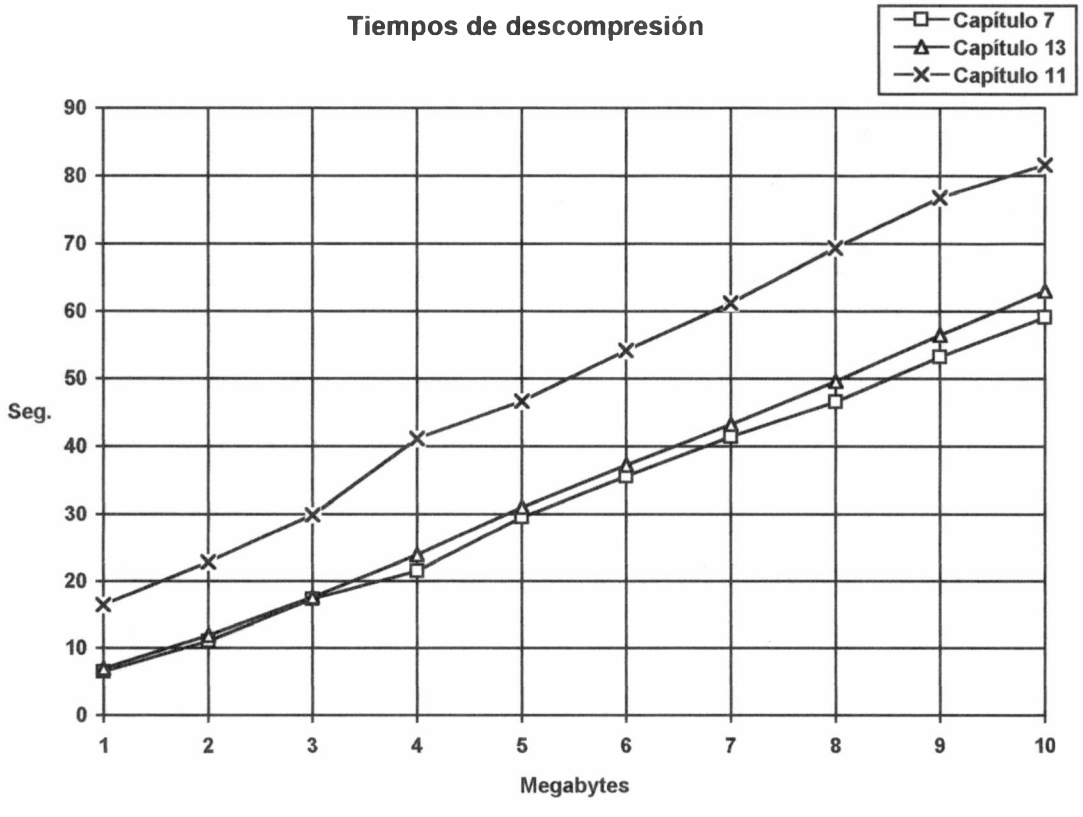
Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	5.56	7
2	2097152	1430653	11.81	11.86
3	3145728	2154364	17.59	17.62
4	4194304	2856867	26.90	23.92
5	5542880	3445067	32.85	31.04
6	6291456	4032455	39.19	37.28
7	7340032	4775894	44.94	43.28
8	8388608	5517596	52.53	49.59
9	9437184	6264067	59.10	56.51
10	10485760	6999398	65.05	63.09

Distribuida (MPI)

Tamaño (Megabytes)	Tamaño (Bytes)	Tamaño Comprimido (Bytes)	Tiempo Compresión (Segundos)	Tiempo Descompresión (Segundos)
1	1048576	710940	18.75	16.5
2	2097152	1430653	27.43	22.84
3	3145728	2154364	34.25	29.95
4	4194304	2856867	42.95	41.12
5	5542880	3445067	50.28	46.62
6	6291456	4032455	57.20	54.22
7	7340032	4775894	64.07	61.28
8	8388608	5517596	72.97	69.47
9	9437184	6264067	78.22	76.82
10	10485760	6999398	85.03	81.86

Gráficos


Tiempos de compresión
Capítulo 7
Capítulo 13
Capítulo 11
Seg.
0
10
20
30
40
50
60
70
80
90
1
2
3
4
5
6
7
8
9
10
Megabytes


Tiempos de descompresión
Capítulo 7
Capítulo 13
Capítulo 11
Seg.
0
10
20
30
40
50
60
70
80
90
1
2
3
4
5
6
7
8
9
10
Megabytes

Conclusiones

Analizando las tablas y los gráficos puede apreciarse que la solución más rápida fué la correspondiente a la descripta en el capítulo 7. Que le sigue la solución correspondiente al capítulo 13 (C estandar y espera selectiva) y, por último, la solución MPI.

Por un lado podemos ver que el solapamiento del procesamiento con la E/S resultó ser más eficiente que la espera selectiva. En otras palabras, el tiempo que se pierde en la creación/control de procesos es menor que el que se pierde en la espera de la realización de una E/S.

Por otro lado, comparando las curvas MPI y C con espera selectiva llegamos a la misma conclusión que habíamos alcanzado en la evaluación de la herramienta realizada en el capítulo 12. El MPI resuelve muy bien la comunicación pero tiene un gran retardo inicial (probablemente debido a la creación de los procesos y a la inicialización de la comunicación).

Capítulo 15

Conclusiones

En este capítulo presentamos las conclusiones generales mas importantes junto con un cuadro sinóptico que resume las características relevantes de las distintas implementaciones distruibuidas.

Una conclusión importante fue el hecho que conviene utilizar múltiples procesos para manejar E/S local y remota en vez de utilizar espera selectiva. Otra fue que, aunque la velocidad de las redes sea baja en proporción a la de acceso a memoria, el aumento del poder de cómputo, si el procesamiento a realizar es intensivo y se debe hacer sobre un conjunto grande de datos, incrementa la velocidad de ejecución en gran medida.

Según los datos recogidos de las pruebas, un compresor distribuido ya tardaba menos tiempo en procesar 1 Mbyte de datos que el compresor lineal. Además, como las pendientes de las curvas que representan los tiempos de ejecución son distintas, a medida que se agranda el tamaño de datos a procesar, aumenta la diferencia en las velocidades de ejecución.

En este punto vale hacer un alto y analizar brevemente algunos temas. Para ganar en velocidad de ejecución, el tiempo que se gana en distribuir el procesamiento y en paralelizar la E/S debe ser mayor que el tiempo que se pierde en la creación/manejo de procesos y en la transferencia de datos a través de la red. Esto limita la utlización práctica del compresor dependiendo del sistema operativo, la plataforma, la red utilizada, la performance de los servidores, etc. a ciertos tamaños de archivos. Por ejemplo, en la configuración particular que utilizamos el límite está bajo el Mbyte ya que los tamaños de archivos que utilizamos en las pruebas fueron de 1 a 10 Mbytes y siempre obtuvimos mejoras.

También cabe mencionar que existen factores que degradan la performance de un compresor distribuido y no de un compresor local. Por ejemplo, el tráfico de la red, la carga de trabajo en los servidores, factores dependientes de los routers, etc.

Además, un compresor distribuído debería incluir algún método de tolerancia a fallos.

A continuación presentamos un cuadro sinóptico que resume las características relevantes de las distintas implementaciones distribuidas.

	Capítulo 2	Capítulo 4	Capítulo 7	Capítulo 9	Capítulo 11	Capítulo 13
Utilización de Memoria Compartida	No	En el cliente y en el servidor	En el cliente	En el cliente	No	No
Semáforos	No	En el cliente	En el cliente	En el cliente	No	No
Comunicación	No	Vía sockets	Vía sockets	Solucionado en forma transparente por la herramienta	Solucionado en forma transparente por la herramienta	Vía sockets
Velocidad de ejecución	5°		1°	4°	3°	2°
Espera Selectiva	No	No	No	No	Si, solucionado en forma transparente por la herramienta	Si, solucionado utilizando la system call select
Control de procesos	No	Si, utilizando las system calls fork y waitpid	Si, utilizando las system calls fork y waitpid	Si, utilizando las system calls fork y waitpid para el manejo de los procesos clientes y utilizando facilidades provistas por la herramienta para la creación de los servidores	Si, solucionado en forma transparente por la herramienta	Si, utilizando las system calls fork y waitpid
Facilidad de Implemenación	Usual	Compleja	Compleja (pero menos compleja que la anterior)	Buena para un programa distribuido	Buena para un programa distribuido	Buena para un programa distribuido
Atención de múltiples clientes simultáneos		No	Si	Si	Si	Si

Parte II: Una herramienta para la distribución del procesamiento en una red de procesadores

Capítulo 16

Introducción

En esta segunda parte presentamos una herramienta para la distribución del procesamiento en una red de procesadores.

La herramienta se basa en una arquitectura cliente/servidor similar a la utilizada en las versiones distribuidas del algoritmo de Huffman descriptas en la parte anterior.

La estructura del cliente es básicamente la misma.

La diferencia fundamental es que en este caso utilizamos el inetd para disparar nuestros servidores (para más datos consultar el apéndice H6) en vez de codificar un servidor completo como lo hicimos en la parte anterior.

Esta herramienta permite agregar o quitar servicios sin tener que recodificar ni el cliente ni el servidor.

Analicemos el funcionamiento del cliente. El cliente recibe un archivo de entrada, un archivo de salida, un modo de funcionamiento, un servicio a utilizar, un tamaño de bloque y una lista con los hosts a utilizar como servidores. Luego distribuye el procesamiento con el mismo criterio utilizado en las versiones distribuidas del algortimo de Huffman descriptas en la parte anterior.

Un ejemplo de la utilización de esta herramienta y en particular la forma en que la utilizamos es el siguiente: definimos servicios compresores y descompresores subordinados al inted, esto nos permitió comprimir/descomprimir archivos (utilizando distintos métodos) en forma distribuida utilizando el mismo cliente.

También utilizamos la herramienta para filtrar, utilizando como filtro el comando cat de UNIX.

La herramienta también podría extenderse para el procesamiento de imágenes y/o sonido (filtrado, realces, transformadas, ...), encriptamiento de datos, etc, todo resuelto en forma distribuida por el mismo cliente.

A continuación describimos brevemente la organización en capítulos de esta parte y los contenidos los mismos.

En el Capítulo 17 se describe la estructura de la herramienta y el funcionamiento de sus componentes.

En el Capítulo 18 se describe el compresor/descompresor basado en la transformada BWT.

En el Capítulo 19 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para las versiones lineal y distribuida del compresor/descompresor BWT junto con algunas conclusiones relevantes.

En el Capítulo 20 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para analizar el desempeño de la herramienta realizando un filtrado distribuido junto con algunas conclusiones relevantes.

En el Capítulo 21 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para las versiones lineal y distribuidas del compresor/descompresor compress/uncompress junto con algunas conclusiones relevantes.

En el Capítulo 22 se presentan tablas y gráficos basados en los resultados obtenidos a partir de la ejecución de lotes de prueba para las versiones lineal y distribuidas del compresor/descompresor gzip/gunzip junto con algunas conclusiones relevantes.

En el Capítulo 23 se presentan algunas conclusiones generales asociadas a esta parte del libro.

Capítulo 17

Descripción de la herramienta

Introducción

La herramienta está compuesta por un cliente y uno o varios servidores.

El cliente distribuye el procesamiento utilizando el mismo esquema que los clientes de los capítulos 4, 7, y 9. Es decir, utiliza la fórmula que habíamos determinado como mejor en la primer parte del libro.

El cliente acepta los siguientes parámetros en la línea de comandos: archivo de entrada (archivo a partir del cual realizar el procesamiento), archivo de salida (archivo donde dejar los resultados del procesamiento), tamaño de bloque a utilizar en la distribución del procesamiento, número de port asociado al servicio requerido, y modo de funcionamiento (actualmente compresor, descompresor o filtrador).

En el modo de compresión el cliente graba el tamaño del archivo de entrada en el archivo de salida y luego dispara los procesos que se encargan de distribuir la entrada y de recoger la salida. Estos procesos leen bloques del archivo de entrada (siguiendo la política de coordinación descripta en la primer parte del libro), utilizan el servicio especificado del servidor que tienen asociado, y graban el bloque comprimido en el archivo de salida agregando un pequeño header de bloque.

En el modo de compresión el cliente lee el tamaño del archivo de salida del archivo de entrada y luego dispara los procesos que se encargan de distribuir la entrada y de recoger la salida. Estos procesos leen bloques (y headers) del archivo de entrada (siguiendo la política de coordinación descripta en la primer parte del libro), utilizan el servicio especificado del servidor que tienen asociado, y graban el bloque descomprimido en el archivo de salida en el offset que ocupaba originalmente.

En el modo de filtrado el cliente dispara los procesos que se encargan de distribuir la entrada y de recoger la salida. Estos procesos leen bloques del archivo de entrada (siguiendo la política de coordinación descripta en la primer parte del libro), utilizan el servicio especificado del servidor que tienen asociado, y graban el bloque filtrado en el archivo de salida en el offset que ocupaba originalmente.

El servidor es básicamente el inetd (al menos en esta implementación). Un servidor completo está compuesto por, el inetd, un wrapper de servicios nuestro y el ejecutable del servicio propiamente dicho.

Para información acerca del inetd referirse al apéndice H6.

El wrapper se incluye debido a que los ejecutables de los servicios no necesariamente saben que están escribiendo/leyendo de un socket (aún más, generalmente no lo sabrán, es decir serán programas realizados para trabajar sobre archivos) por lo que no tendrán en cuenta algunas particularidades referidas a estos (xEj un read puede retornar menos datos que los pedidos y puede ser que no se halla alcanzado el fin del stream). El wrapper por un lado resuelve la comunicación de ida y vuelta con el cliente y por otro genera archivos temporales de entrada y de salida para el ejecutable del servicio de manera que este no tenga que lidiar directamente con los sockets.

El funcionamiento del wrapper es el siguiente, lee los datos que le envía el cliente y genera un archivo temporal con ellos. Luego genera un archivo temporal vacío para que el ejecutable del

servicio vuelque sus resultados. Luego dispara el ejecutable del servicio (fork/execve) redireccionando la entrada estandar al archivo temporal que contiene los datos que envió el cliente, la salida estandar al archivo temporal vacío y el error estandar a /dev/null (podría redireccionarse a algún archivo de logeo de errores asociado a la herrramienta). El tema del redireccionamiento del error estandar es importante ya que de otro modo todo lo que salga por el error estandar iría al error estandar del wrapper (el socket !!!).

Por último, el ejecutable del servicio es cualquier compresor, descompresor o filtro lineal que lea de su entrada estandar y escriba en su salida estandar.

En los próximos capítulos se presentan varias instanciaciones de la herramienta, una de ellas utiliza un compresor/descompresor basado en la transformada BWT y el resto utilizan un grupo de ejecutables típicos de UNIX (cat, gzip/gunzip, compress/uncompress).

El agregar servicios es tan simple como agregar el ejecutable del servicio al archivo inetd.conf y asociarle un port a este nuevo servicio en el archivo services.

Por ejemplo, para distribuir el cat agregamos la siguiente entrada al archivo /etc/inetd.conf

```
cat    stream      tcp    nowait      root   /home/andir/bin/wrapper /bin/cat
```

y la siguiente entrada al archivo /etc/services

```
cat    10000/tcp
```

en todos los hosts listados en el archivo .cmprc (el archivo que lista los hosts a utilizar como servidores).

Esto es todo lo que hizo falta modificar para que nuestro cliente distribuya el cat. La invocación del cliente es como sigue:

```
cliente -f -i archivo-de-entrada -o archivo-de-salida -s 10000
```

El parámetro -f le indica al cliente que se va a realizar un filtrado, el -i le indica el nombre del archivo de entrada, el -o el archivo de salida y el -s el port del servicio a utilizar.

Capítulo 18

Compresor/Descompresor basado en la transformada BWT

Introducción

En este capítulo describiremos la estructura de un compresor/descompresor basado en la transformada BWT y explicaremos el porqué de la misma.

Descripción

El compresor está implementado a través del siguiente pipe:

rle | bwt | mtf | rle | ari

rle: Implementa un codificador Run-Length. Este algoritmo explota la ocurrencia de corridas de caracteres iguales. Además, con su primera ocurrencia en el pipe, evita que se degrade la performance de la transformada a causa de comparaciones de strings que comparten el mismo prefijo. En su segunda ocurrencia aprovecha la estructura de la salida del MTF.

bwt: Transformada BWT (para más detalles referirse al apéndice C3).

mtf: Implementa un codificador Move-To-Front. Explota localidades de datos (para cada localidad genera un conjunto de enteros pequeños).

ari: Realiza una codificación aritmética de orden cero similar a la codificación de Huffman.

El descompresor, como era de esperarse, está dado por el siguiente pipe

unari | unrle | unmtf | unbwt | unrle

Capítulo 19

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

Utilizando la herramienta descripta en el capítulo 17 y el compresor/descompresor del capítulo anterior realizamos un compresor/descompresor BWT distribuido. El hacerlo sólo requirió una implementación lineal de los mismos y la actualización de las tablas correspondientes del inetd en los hosts servidores.

Debido a restricciones del formato de las tablas del inetd debimos utilizar este pipe dentro de un shell script que a su vez era invocado por el wrapper descripto en el capítulo 17.

El hecho de que el servicio sea provisto por un pipe encerrado en un shell scrip debe ser tenido en cuenta en el análisis de performance.

En este capítulo se analizan los resultados obtenidos de las pruebas realizadas utilizando la herramienta de distribución del procesamiento descripta en el capítulo 17 y la versión lineal del algoritmo de compresión basado en la transformada BWT.

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

Datos recogidos de las pruebas

Tablas

Tiempos de compresión en segundos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	91,66	21,29	15,18	80,45
2	169,78	40,98	27,36	78,98
3	305,41	61,39	40,53	78,60
4	380,77	83,07	57,29	77,97
5	512,83	104,76	77,51	104,52
6	586,46	123,99	84,76	123,87
7	653,73	143,98	96,46	149,80
8	718,18	165,15	109,92	148,58
9	783,59	185,53	120,47	159,88
10	850,26	206,63	136,76	162,72

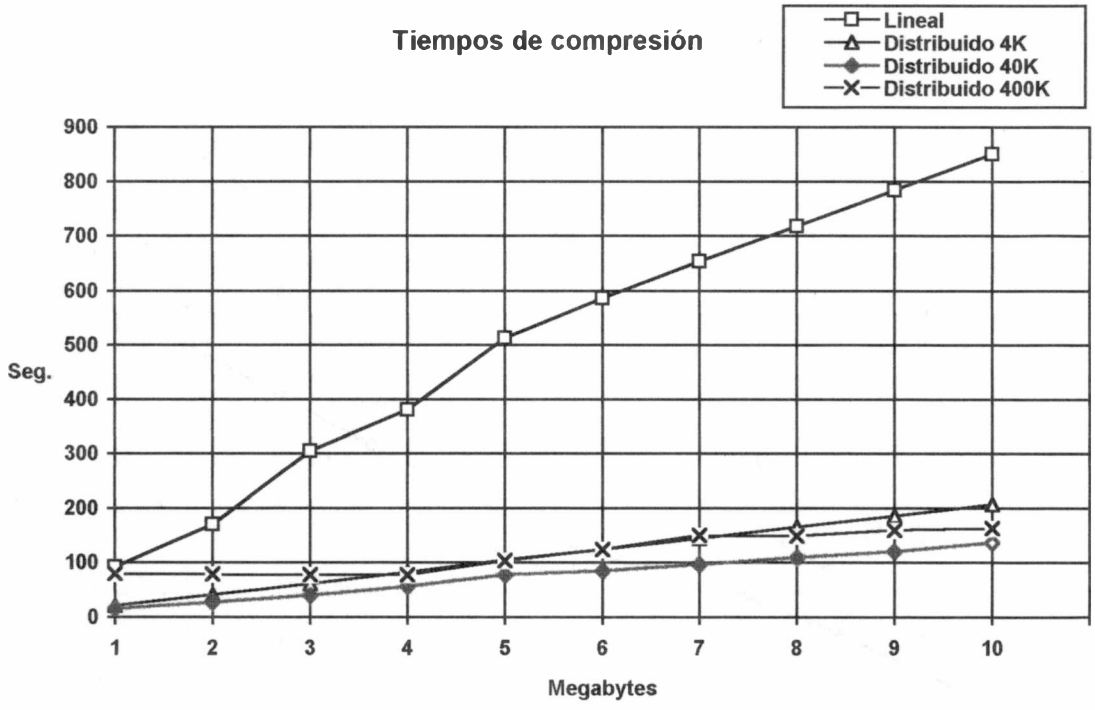
Tiempos de descompresión en segundos

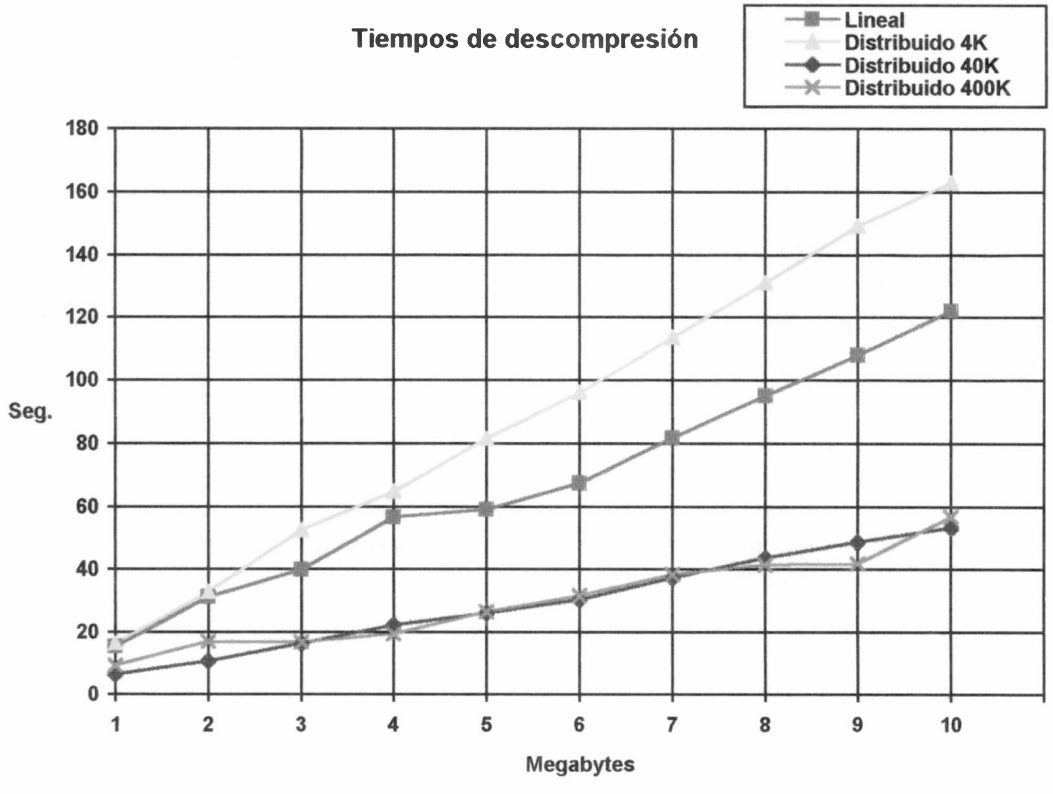
Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	15,40	16,32	6,31	9,28
2	31,35	32,94	10.61	16,94
3	39,87	52,49	16,33	16,81
4	56,61	64,75	22,36	19,53
5	59,09	81,68	26,22	26,58
6	67,59	96,08	30,57	31,73
7	81,72	113,64	37,37	38,47
8	95,09	131,05	43,79	41,59
9	108,06	149,30	48,86	41,85
10	122,06	162,94	53,34	56,81

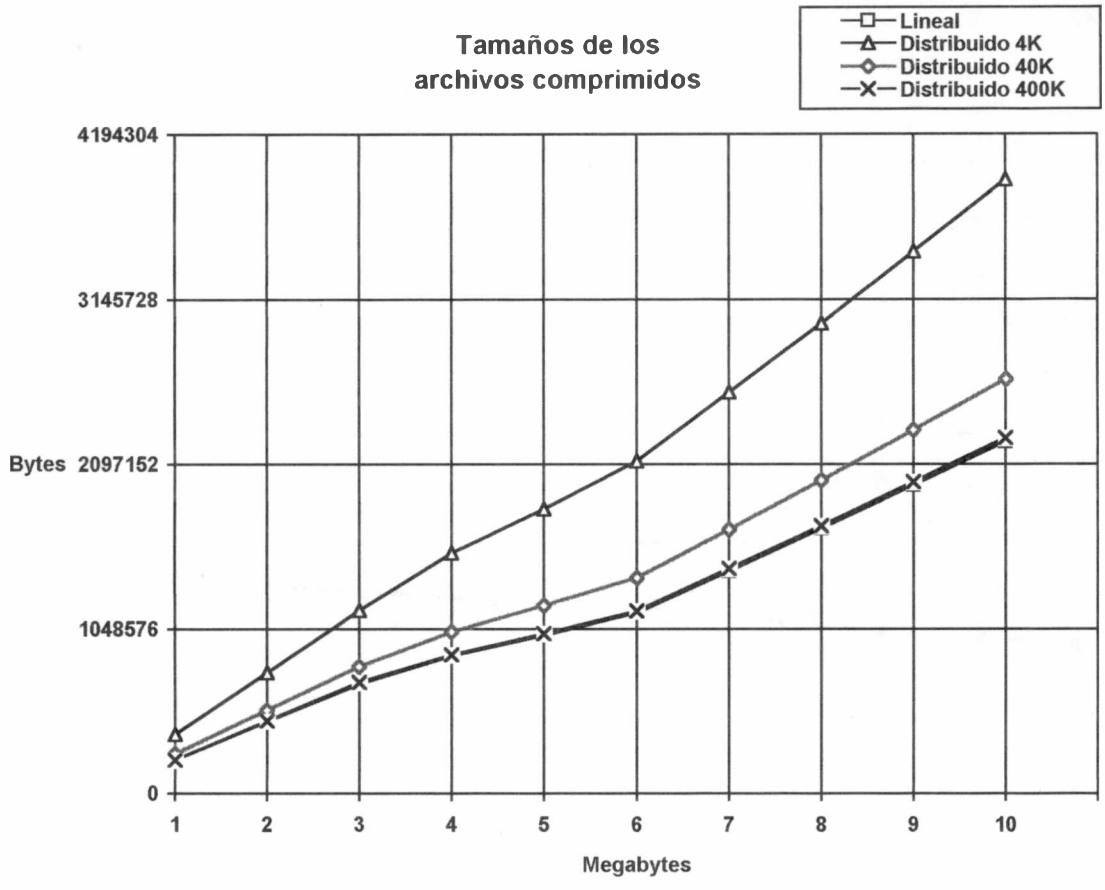
Tamaño de los archivos comprimidos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	212150	375755	252996	212724
2	459080	769757	534630	463276
3	698349	1166655	812757	707888
4	877525	1532466	1033228	887446
5	1007811	1813548	1200161	1018966
6	1152081	2115574	1376767	1164797
7	1420010	2554322	1685113	1433162
8	1689563	2993764	1995062	1706726
9	1962935	3452726	2317308	1982726
10	2240589	3905903	2639915	2262138

Gráficos


Tiempos de compresión
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
900
800
700
600
500
400
300
200
100
0
Seg.
1
2
3
4
5
6
7
8
9
10
Megabytes


Tiempos de descompresión
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
180
160
140
120
100
80
60
40
20
0
Seg.
1
2
3
4
5
6
7
8
9
10
Megabytes



Conclusiones

Las versiones distribuidas del compresor fueron mucho más rápidas que la versión lineal.

La versión distribuida que utiliza bloques de 4K degrada demasiado los radios de compresión por lo que quedaría descartada su utilización. La de 40K los degrada pero se mantiene cerca de los radios de la lineal. Y la que utiliza bloques de 400K los mantiene casi idénticos.

Por otro lado, el descompresor lineal es más rápido que el distribuido de 4K pero es más lento que las otras dos versiones distribuidas. Entre estas dos últimas versiones distribuidas casi no existen diferencias. Nuevamente, nos encontramos con que el proceso de descompresión requiere menos procesamiento que el de compresión lo cual le permitió al descompresor lineal acomodarse dentro de los tiempos de las versiones distribuidas.

También puede verse que varían los radios de compresión y la velocidad de ejecución de las versiones distribuidas. Como el cliente acepta la especificación del tamaño de bloque se puede optimizar respecto de la compresión, de la velocidad, o de ambas.

Vale resaltar que, a diferencia de este caso (pipe y script), normalmente el servicio será proveído por un único ejecutable binario. Esto mejorará los tiempos de ejecución de las versiones distribuidas.

Como un último comentario vale aclarar que la diferencia de velocidad entre las versiones distribuidas del compresor y la versión lineal no sólo se debe a la distribución del procesamiento sino

al tamaño de bloque utilizado por la implementación lineal del compresor. Una prueba que arrojaría resultados más ajustados sería realizar 3 compresores lineales, uno que utilice bloques de 4K, otro bloques de 40K y el último bloques de 400K, y comparar cada caso con su equivalente distribuido.

Capítulo 20

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

En este capítulo se analizan los resultados obtenidos de las pruebas realizadas utilizando la herramienta de distribución del procesamiento descripta en el capítulo 17 con el comando cat. El comando fue usado como un filtro, en particular el filtro más trivial (simplemente copiar la entrada en la salida).

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

El script que utilizamos para automatizar las corridas limpiaba la cache de disco antes de ejecutar el comando a testear.

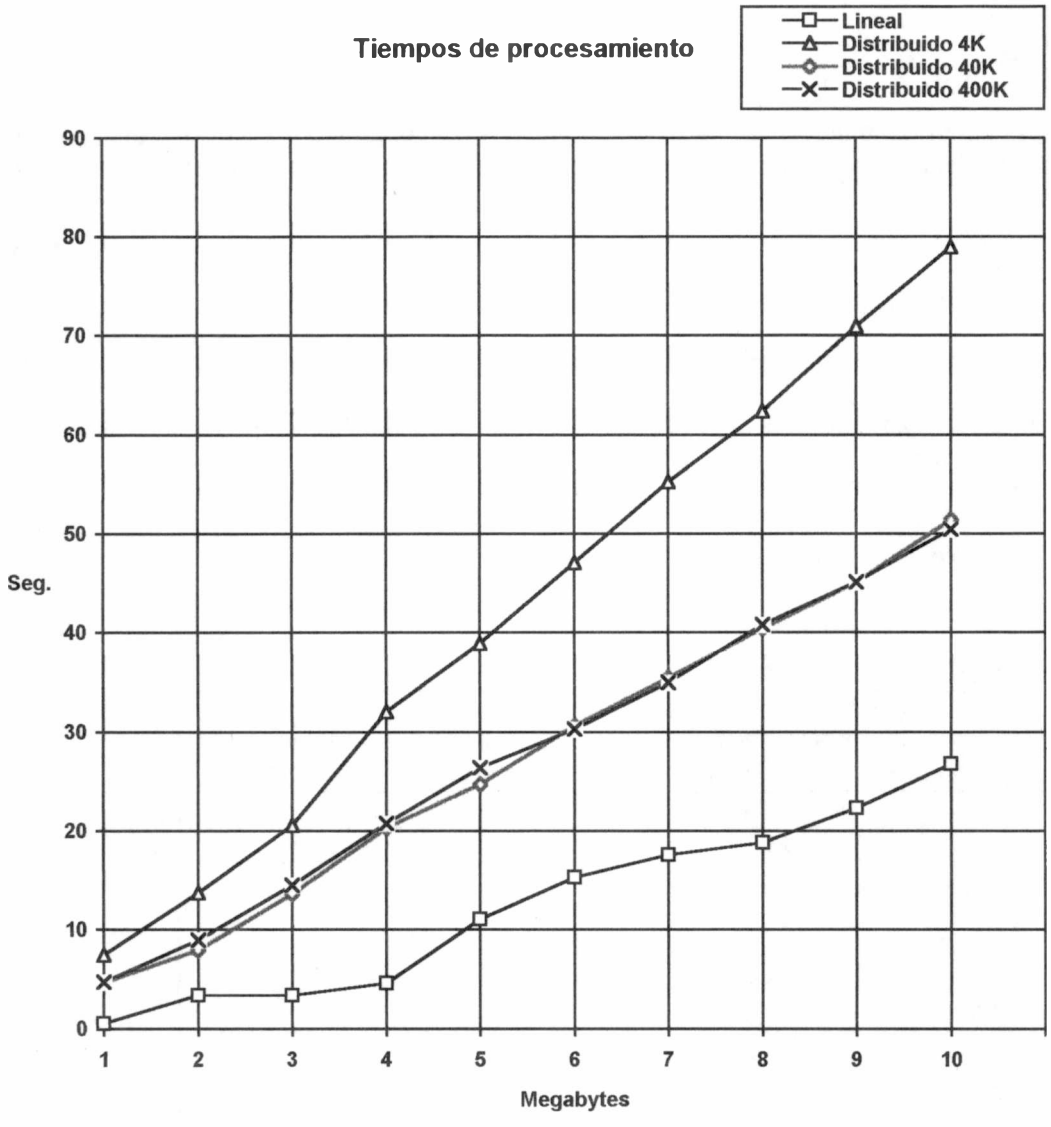
Datos recogidos de las pruebas

Tablas

Tiempos de procesamiento en segundos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	0,54	7,51	4,67	4,73
2	3,41	13,71	7,93	8,98
3	3,40	20,58	13,60	14,48
4	4,63	32,07	20,33	20,73
5	11,07	38,89	24,75	26,41
6	15,31	47,03	30,66	30,35
7	17,55	55,25	35,48	35,04
8	19,81	62,43	40,42	40,80
9	22,34	70,95	45,10	45,13
10	26,80	78,91	51,43	50,46

Gráficos



Conclusiones

Las versiones distribuidas fueron más lentas que la versión lineal. Con esto podemos concluir que este esquema de distribución necesita que el procesamiento sea intensivo.

Otra cosa a notar es la variación de los tiempos de las versiones distribuidas según el tamaño de bloque utilizado. Como la herramienta permite la especificación del tamaño del bloque a utilizar en la línea de comandos es posible buscar para cada servicio y cada configuración el tamaño ideal de bloque.

Capítulo 21

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

En este capítulo se analizan los resultados obtenidos de las pruebas realizadas utilizando la herramienta de distribución del procesamiento descripta en el capítulo 17 con los comandos compress/uncompress.

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

El script que utilizamos para automatizar la corrida limpiaba la cache de disco antes de ejecutar el comando a testear.

Datos recogidos de las pruebas

Tablas

Tiempos de compresión en segundos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	4,94	8,56	4,11	5,82
2	9,88	13,79	6,85	8,35
3	13,80	20,81	10,27	10,15
4	17,95	30,32	15,66	16,06
5	25,42	38,28	20,03	19
6	31,86	46,11	24,66	21,97
7	37,71	53,36	27,50	27,72
8	43,98	61,52	33,11	28,54
9	47,66	68,75	35,32	37,72
10	53,03	75,39	40,49	38,32

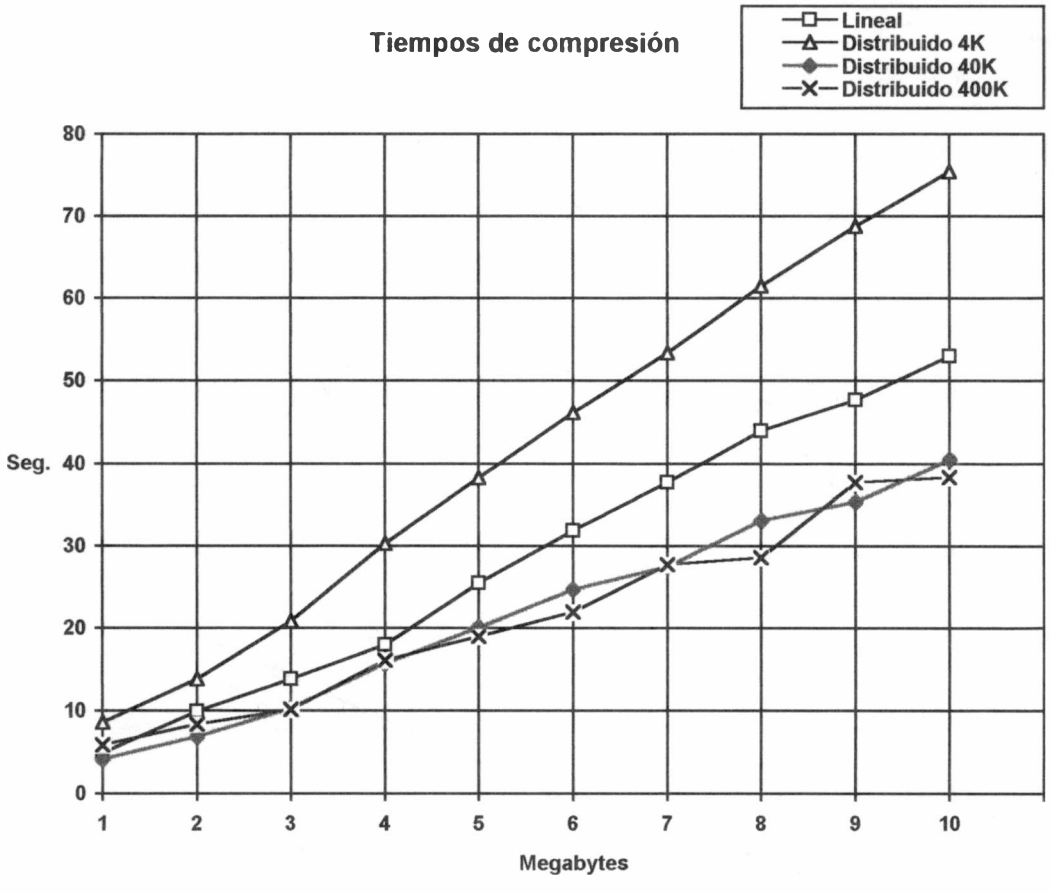
Tiempos de descompresión en segundos

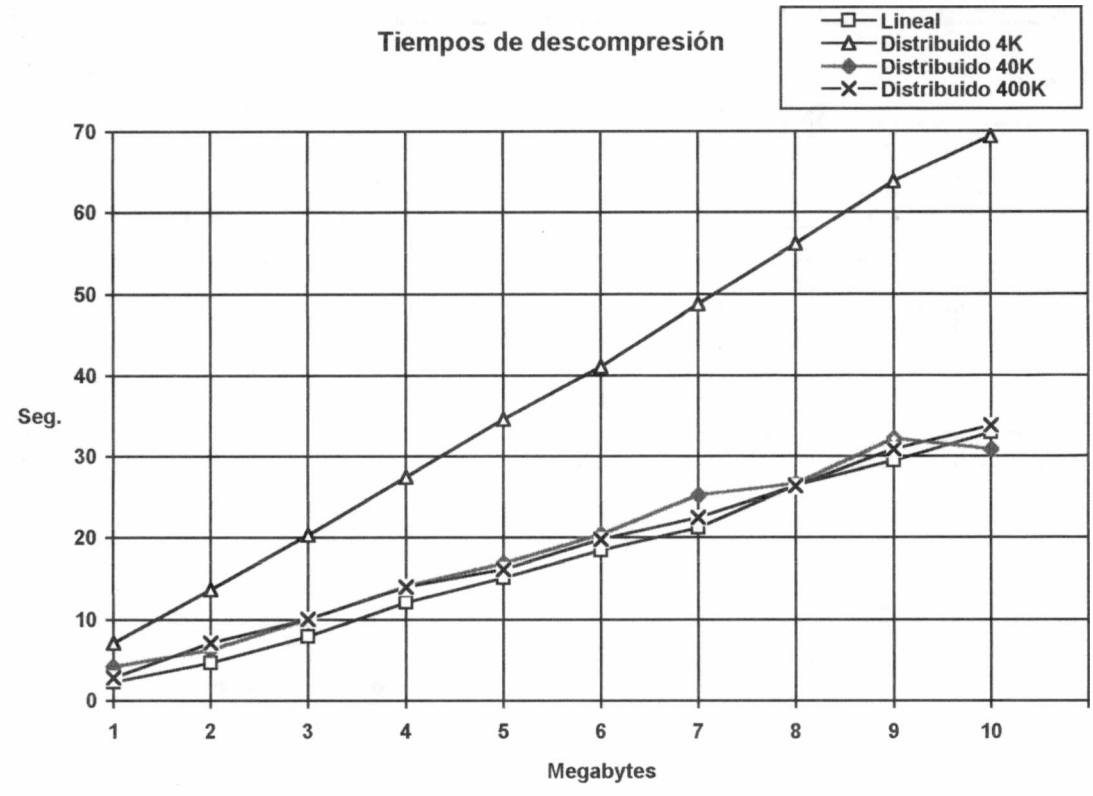
Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	2,32	7,09	4,23	2,86
2	4,69	13,72	6,22	7,13
3	7,95	20,35	10,07	10,07
4	12,14	27,47	14,07	14,02
5	15,11	34,59	16,84	16,13
6	18,44	41,06	20,45	19,77
7	21,26	48,79	25,25	22,45
8	26,41	56,15	26,67	26,40
9	29,49	63,79	32,21	30,87
10	32,95	69,38	35,82	33,85

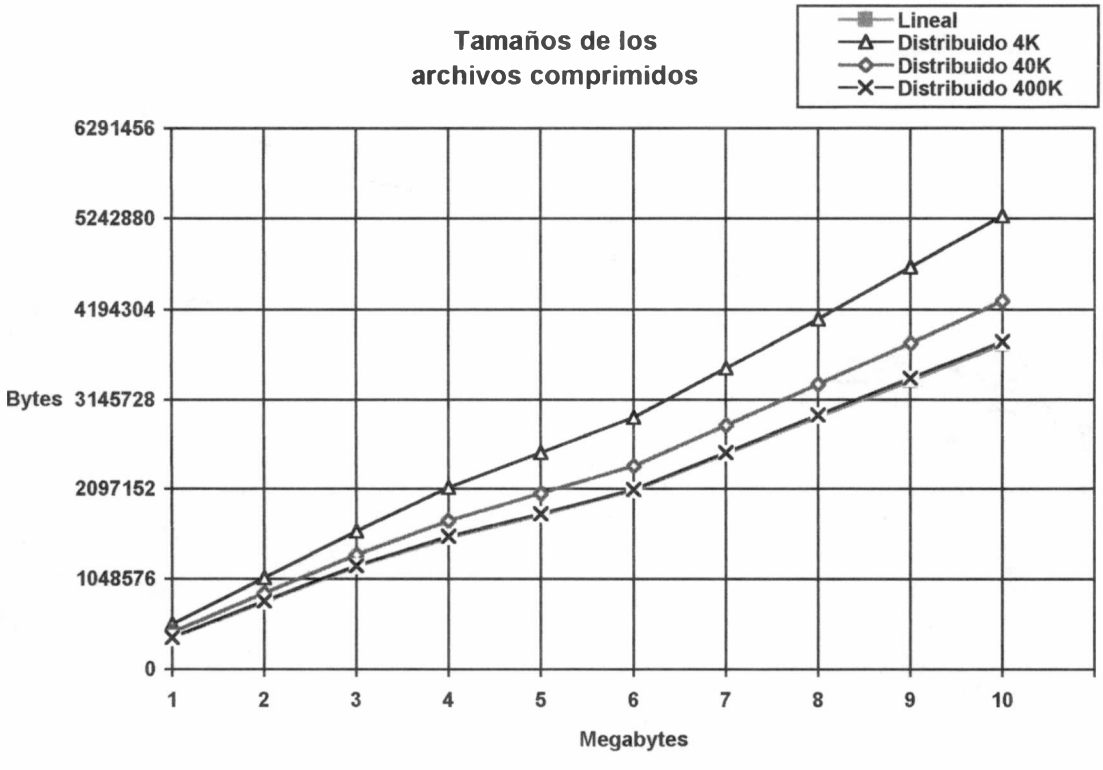
Tamaño de los archivos comprimidos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	373629	525562	428254	375402
2	785350	1064584	883014	792948
3	1196249	1604737	1334073	1203613
4	1528165	2110921	1729438	1548680
5	1796677	2524154	2046384	1810832
6	2085570	2937114	2369507	2095393
7	2512599	3514118	2846211	2525914
8	2937717	4091029	3324567	2964640
9	3366985	4684361	3812845	3397319
10	3796059	5269366	4295152	3830117

Gráficos




Tiempos de descompresión
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
70
60
50
40
30
20
10
0
Seg.
1
2
3
4
5
6
7
8
9
10
Megabytes


Tamaños de los
archivos comprimidos
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
6291456
5242880
4194304
3145728
2097152
1048576
0
Bytes
1
2
3
4
5
6
7
8
9
10
Megabytes

Conclusiones

En la compresión la versión lineal fue más rápida que la versión distribuida que utilizaba bloques de 4K pero fue más lenta que las otras dos versiones distribuidas.

En la descompresión ninguna de las versiones distribuidas logró igualar la performance de la versión lineal. Debido a que generalmente la descompresión implica mucho menos procesamiento que la compresión.

Otra cosa que se puede ver es que los radios de compresión casi no fueron degradados en la versión distribuida que utilizaba bloques de 400K.

Al igual que en los capítulos anterior la variación de los tiempos de las versiones distribuidas dependían en parte del tamaño de bloque utilizado. En este caso se puede optimizar el tamaño de bloque para ganar velocidad o ganar compresión o buscar un punto medio conveniente.

Capítulo 22

Estadísticas, datos recogidos de las pruebas, análisis y conclusiones

Introducción

En este capítulo se analizan los resultados obtenidos de las pruebas realizadas utilizando la herramienta de distribución del procesamiento descripta en el capítulo 17 con los comandos gzip/gunzip.

Realizamos las pruebas utilizando 4 hosts de la red del LIDI con la misma carga de trabajo en los hosts y el mismo tráfico en la red.

Utilizamos el comando time para medir los tiempos y la utilización de recursos.

El script que utilizamos para automatizar la corrida limpiaba la cache de disco antes de ejecutar el comando a testear.

Datos recogidos de las pruebas

Tablas

Tiempos de compresión en segundos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	9,42	8,60	5,03	9,76
2	17,54	13,51	7,3	9,93
3	25,21	20,23	9,55	10,41
4	32,57	29,42	13,15	15,95
5	42,21	37,19	18,55	18,99
6	51,14	44,28	23,17	21,64
7	62,70	51,56	26,27	27,05
8	72,66	56,87	30,70	31,29
9	84,45	65,12	33,88	33,94
10	94,80	72,89	37,12	41,65

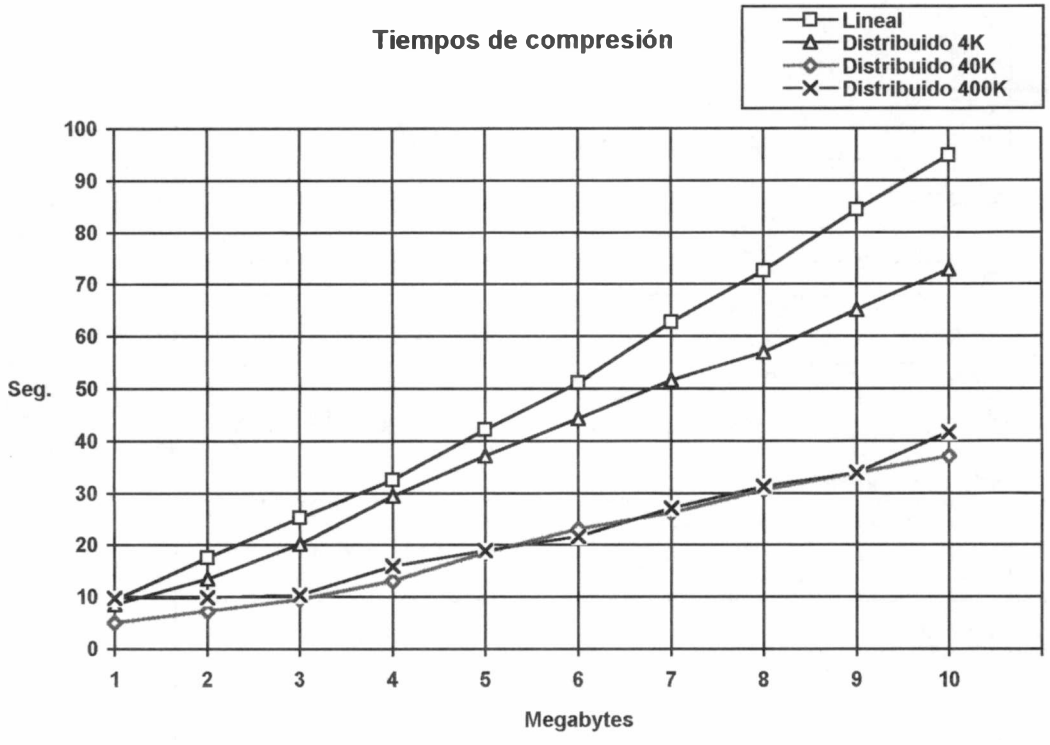
Tiempos de descompresión en segundos

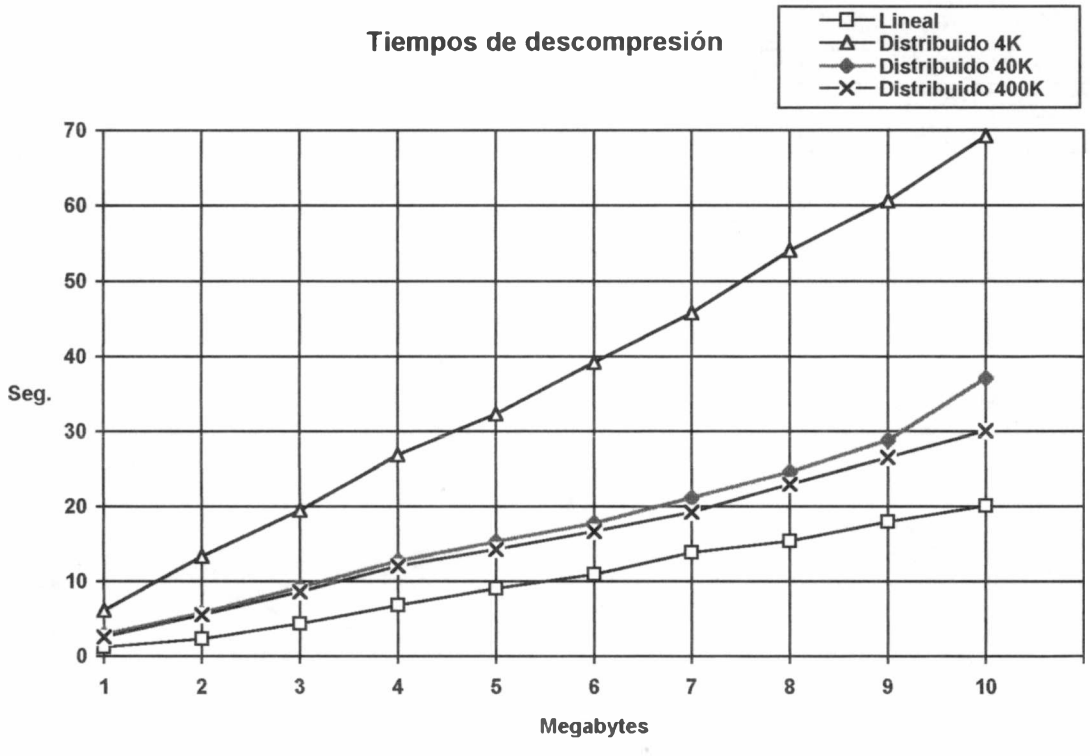
Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	1,19	6,13	2,88	2,55
2	2,29	13,34	5,78	5,54
3	4,38	19,50	9,21	8,67
4	6,83	26,95	12,77	12,02
5	9,03	32,35	15,33	14,25
6	10,95	39,21	17,74	16,68
7	13,81	45,81	21,18	19,25
8	15,38	54,08	24,69	23,02
9	17,97	60,61	28,88	26,56
10	20,10	69,26	37,17	30,10

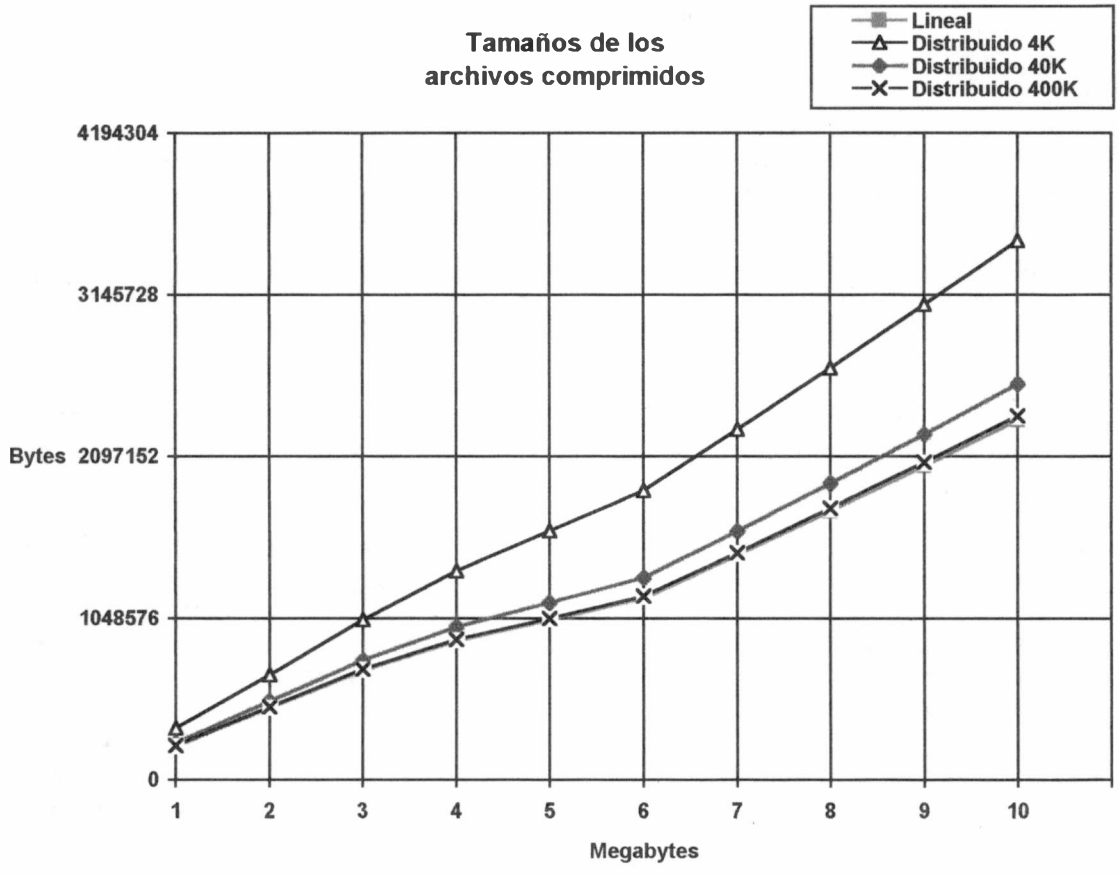
Tamaño de los archivos comprimidos

Tamaño (Megabytes)	Lineal	Distribuido (Bloques de 4K)	Distribuido (Bloques de 40K)	Distribuido (Bloques de 400K)
1	218232	333181	239509	219502
2	467638	682168	507894	470908
3	711752	1042037	779070	717382
4	902715	1358723	993392	911191
5	1039491	1615050	1150957	1049694
6	1182831	1877630	1315293	1194934
7	1462109	2272163	1618765	1475810
8	1745391	2669637	1926070	1762779
9	2036666	3083217	2245356	2058452
10	2336157	3497779	2570050	2359518

Gráficos


Tiempos de compresión
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
Seg.
0
10
20
30
40
50
60
70
80
90
100
1
2
3
4
5
6
7
8
9
10
Megabytes


Tiempos de descompresión
Lineal
Distribuido 4K
Distribuido 40K
Distribuido 400K
Seg.
0
10
20
30
40
50
60
70
1
2
3
4
5
6
7
8
9
10
Megabytes



Conclusiones

En la compresión la versión lineal fue más lenta que todas las versiones distribuidas. La versión distribuida que utilizaba bloques de 4K fue mas lenta que las otras dos versiones ditribuidas. Las otras dos versiones distribuidas tuvieron una performance similar.

En la descompresión ninguna de las versiones distribuidas logró igualar la performance de la versión lineal. Debido a que generalmente la descompresión implica mucho menos procesamiento que la compresión, para analizar el porque de la afirmación anterior con un ejemplo real referirse al apéndice que describe la transformada BWT y su utilización para la compresión de datos (C3). Por otro lado, nuevamente la performance de las versiones distribuidas con tamaños de bloque de 40K y 400K fueron similares.

Otra cosa que se puede ver es que los radios de compresión casi no fueron degrados en la versión distribuida que utilizaba bloques de 400K y que la de 40K no se alejó demasiado de los radios originales.

Al igual que en el capítulo anterior la variación de los tiempos de las versiones distribuidas dependían en parte del tamaño de bloque utilizado. En este caso se puede optimizar el tamaño de bloque para ganar velocidad o ganar compresión o buscar un punto medio conveniente.

Capítulo 23

Conclusiones

A partir de los capítulos referidos a los resultados obtenidos con los compresores distribuidos podemos concluir que si el procesamiento es intensivo se puede incrementar en gran medida la velocidad de ejecución. Por otro lado, si el procesamiento no es intensivo, para que distribuirlo.

Esto abre las puertas a métodos de compresión que habían sido descartados por su alto tiempo de procesamiento (la transformada BWT es un ejemplo).

El filtrado distribuido no dió buenos resultados ya que el filtro utilizado no realizaba ningún tipo de procesamiento sobre los datos, directamente copiaba su entrada estandar a su salida estandar. Este ejemplo muestra claramente que la distribución del procesamiento no es gratuita y que debe existir un mínimo de procesamiento que justifique su realización.

Entre las posibles aplicaciones de este tipo de distribución podríamos destacar el backup comprimido y el procesamiento de señales en general (imágenes, sonido, ...).

Analizando los resultados se puede concluir que uno puede incrementar su capacidad de procesamiento en gran medida distribuyendo el cómputo en una red de ordenadores comunes sin necesidad de hardware extraordinario.

La aplicación de esta estrategia tiene sus limitaciones ya que debe existir un procesamiento intensivo para obtener buenos resultados pero generalmente es en esos casos en los que uno puede estar interesado en ganar performance.

Trabajos futuros

Parte I

Implementación del compresor/descompresor del capítulo 7 con threads (utilizar light weigth process).

Ampliar las pruebas para hallar el número de procesadores (si es que lo hay) en el cual la ganancia se estabiliza o comienza decaer.

Parte II

Optimización del compresor basado en BWT

Mejorar la agrupación de las localidades cambiando el orden de los caracteres o unificando localidades entre bloques.

Unificación del pipe en un sólo proceso.

Extender la funcionalidad de la herramienta de distribución

Procesamiento de imágenes

Encriptado de datos

Backups

Filtros que modifiquen el tamaño de su entrada

Mejorar la interface de la herramienta de distribución

Configuración a través de tablas y variables de ambiente, definición de valores por default, asociación de extensiones con servicios, asociacion de nombres con servicios, ...

Optimización de la herramienta de distribución

Codificar un servidor dedicado (unificar la funcionalidad del inetd y del wrapper en un servidor especializado).

Optimizar a nivel de comunicación (analizar la performance utilizando diferentes clases de sockets [udp, sockets unix, ...]).

Incluir un mecanismo de tolerancia a fallos.

Ampliar las pruebas

Hallar el tamaño de bloque ideal para las distintas instanciaciones de la herramienta.

Realizar pruebas con filtrados más costosos.

Parte III: Apéndices

Compresión de Datos

C1 Introducción a la Compresión de Datos

A través del desarrollo de la computación, el volumen de datos transferidos debido a la computación remota, o el ocupado por los sistemas de almacenamiento y recuperación de información ha tenido un gran crecimiento.

Un problema puntual es, por ejemplo, el excesivo crecimiento del tamaño de las bases de datos. Este problema puede ser abordado agregando espacio de almacenamiento (discos, cintas, ...).

Acompañando el crecimiento del tamaño de las bases de datos también creció el número de usuarios y la duración de sus sesiones remotas. Estos factores determinaron que una tremenda cantidad de datos sea transferida entre computadoras y terminales remotas. En un intento de hacer viable la transmisión de la cantidad de datos requerida, las líneas de comunicación y los dispositivos auxiliares, han ido evolucionando hasta alcanzar grandes capacidades de transferencia de datos.

Como ya lo mencionamos, una solución obvia a este problema es la instalación de dispositivos de almacenamiento adicional y la expansión de las facilidades de comunicación. Esto implica un incremento del equipamiento de la organización con el costo que esto tiene asociado. Un método alternativo es intentar codificar los datos en forma más eficiente. Si uno examina una base de datos de una organización o monitorea una línea de transmisión hay una excelente chance de que la secuencia de caracteres que conforman los recursos anteriores pueda ser codificada más eficientemente.

Un ejemplo simple de compresión de datos es una base datos de PERSONAL con un campo OCUPACION de longitud fija (30 caracteres alfanuméricos). Supongamos que un valor válido de este campo es OBRERO. Los registros que contengan este valor de campo estarían desperdiciando 24 caracteres. La existencia de muchos obreros desembocaría en un gran desperdicio del almacenamiento. Suponiendo que existen 256 ocupaciones distintas es posible indicar cada ocupación con un sólo caracter. Esto permite ahorrar 29 caracteres por registro sólo observando ese campo.

Otro ejemplo es la codificación de un campo FECHA. Supongamos que inicialmente se dedicaban 25 caracteres para permitir representar fechas con el siguiente formato: 24 de Agosto de 1997. Una primera aproximación podría ser dedicar un caracter para el día un caracter para el mes y un caracter para el año (3 caracteres). Otra más interesante, se basa en la representación binaria. Y es dedicar el número de bits necesario para cubrir el rango de valores de cada campo (5 para el día, 4 para el mes, y 7 para el año [1900-2027]), esto permitiría representar una fecha utilizando sólo 2 caracteres (asumiendo que los caracteres son de 8 bits).

El primer ejemplo es un caso típico de los que se presentan en el momento del diseño de la estructura de una base de datos. El segundo, está relacionado con el comportamiento del manejador de la base de datos. Esto muestra que la compresión de datos puede aplicarse en varios niveles, generalmente en forma independiente y por distintos agentes. Algunos autores llaman a estas formas de compresión, compresión lógica. Y llaman compresión física al proceso de reducir la cantidad de datos antes de entrar a un método de almacenamiento/transferencia y al de la posterior expansión de los datos a su formato original. La compresión física se aplica en un nivel inferior a los anteriormente nombrados. La compresión lógica intenta representar los datos en forma eficiente sin perder de vista su significado (organizar la información de una manera alternativa). La compresión física trata de recodificar la información de acuerdo a características matemáticas y probabilísticas de los datos ignorando su significado y, probablemente, trabajando en crudo sobre la corriente de caracteres. Por ejemplo, codificando los caracteres mas frecuentes con menos bits y los menos frecuentes con mas bits. Otro puede ser el reemplazo de cadenas de caracteres iguales por un par (cantidad, caracter).

Los beneficios de la compresión de datos no sólo tienen que ver con la reducción de los requerimientos de almacenamiento sino que en algunos casos pueden reducir el tiempo de ejecucción de los programas. Ya que si bien se agregan instrucciones para la compresión/descompresión de los datos, se reduce la cantidad necesaria de accesos a los dispositivos (discos, cintas, red, ..), los cuales generalmente tienen tiempos de acceso que degradan en gran medida la ejecución de los programas. Otro beneficio, en el caso de la transmisión de datos, es la disminución de los costos asociados, por ejemplo, el relacionado con el tiempo de utilización de una línea telefónica. Además, al transmitirse menos caracteres también se reduce la probabilidad de error. Otro beneficio, es el encriptamiento intrínseco de los datos, ya que esto agrega un nivel de seguridad adicional ante intrusos o espías.

La compresión de datos puede estar implementada en hardware o en software. Generalmente, la implementación se realiza inicialmente en software, a medida que los algoritmos se optimizan, ganan aceptación y se estandarizan, se van migrando a componentes de hardware (modems, placas de video, ...).

C2 Códigos de Huffman

Dadas las probabilidades de ocurrencia de los diferentes símbolos de una fuente vamos a describir un método de construcción de un código instantáneo (el algoritmo de Huffman) y vamos a demostrar que el código resultante pertenece al conjunto de códigos eficientes asociado al conjunto de probabilidades dado. Cuando decimos que un código es eficiente queremos significar que no existe un código instanáneo asociado al mismo conjunto de probabilidades cuya longitud promedio sea menor.

El algoritmo para la construcción del código a partir del conjunto de probabilidades es como sigue. Lo que haremos es ir agrupando en cada paso las dos probabilidades mas pequeñas en una única probabilidad correspondiente a la suma de las dos probabilidades seleccionadas. Ahora vamos a generar un código para el nuevo conjunto de probabilidades. Este código tendrá un símbolo menos que el del paso anterior. Seguimos de esta manera hasta que sólo queden dos probabilidades que codificar. Aquí, la forma de codificar es única (asignar un 1 a una probabilidad y un 0 a la otra) y cualquiera de los dos códigos resultantes es eficiente. Ahora vamos hacia atrás armando las palabras de código de los códigos cuya generación nos fue quedando pendiente. Para cada probabilidad del conjunto corriente hacemos lo siguiente, si fue compuesta la explotamos en las dos probabilidades componentes y asociamos a una de estas el símbolo que tenía asociado la probabilidad compuesta con un 1 al final y a la otra el mismo símbolo pero con un 0 al final. Así hasta que el conjunto corriente de probabilidades sea el conjunto de probabilidades originales.

En este momento tenemos un código instantáneo, basado en las probabilidades de ocurrencia de los símbolos, que pertenece al conjunto de códigos eficientes asociado al conjunto de probabilidades dado.

Para demostrar la proposición anterior vamos a utilizar otras proposiciones que pasaremos a enunciar y demostrar.

Proposición 1: Una condición necesaria para que un código sea eficiente es la siguiente. Si ordenamos las probabilidades de mayor a menor y listamos bajo cada probabilidad la longitud de la palabra de código asociada a ella, dichas longitudes deben quedar ordenadas de menor a mayor.

Demostración: Denotemos las probabilidades de aparición como p_i y las palabras de código como s_i. Supongamos que hay q palabras. Supongamos que l_i representa la longitud en bits de la palabra s_i. Supongamos que L representa la longitud promedio del código ($L = \sum_{i=1}^{q} p_i l_i$). Supongamos que ordenamos las p_i de mayor a menor. Vamos a demostrar que si las l_i asociadas no quedaron ordenadas de menor a mayor, el código puede ser mejorado reasignando las palabras de código.

Supongamos que p_m es mayor que p_n y que l_m es mayor que l_n, entonces $L = p_m l_m + p_n l_n + \sum_{i \in \{1, ..., q\} - \{m,n\}} p_i l_i$. Sea L' la longitud promedio del código resultante al intercambiar las palabras de código ($L' = p_m l_n + p_n l_m + \sum_{i \in \{1, ..., q\} - \{m,n\}} p_i l_i$). Es fácil demostrar que $L' < L$ (o equivalentemente, $p_m l_n + p_n l_m < p_m l_m + p_n l_n$). Por lo tanto el código original no es eficiente.

Proposición 2: Una condición necesaria para que un código instantáneo sea eficiente es la siguiente. Sea l la longitud de la/s palabra/s de código más larga/s. Al menos dos símbolos deben estar codificados con palabras de esa longitud y además los dos símbolos menos probables deben pertenecer al conjunto de símbolos codificados con palabras de esa longitud.

Demostración: Supongamos que el símbolo de mayor longitud tiene longitud l. Si es único, como el código es instantáneo, si descartamos el último bit el código obtenido sigue siendo instantáneo y tiene menor longitud promedio que el código original. Por lo tanto el código original no es eficiente.

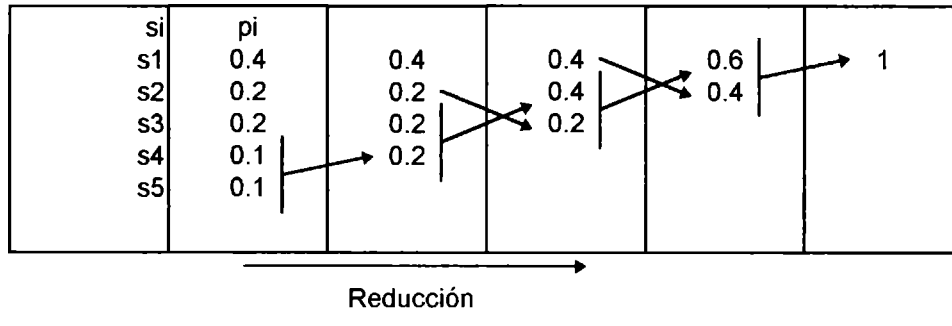
Proposición 3: Esta proposición generaliza en cierta forma a la proposición anterior. Una condición necesaria para que un código instantáneo sea eficiente es la siguiente. No deben existir en el árbol de decodificación nodos que tengan un sólo hijo.

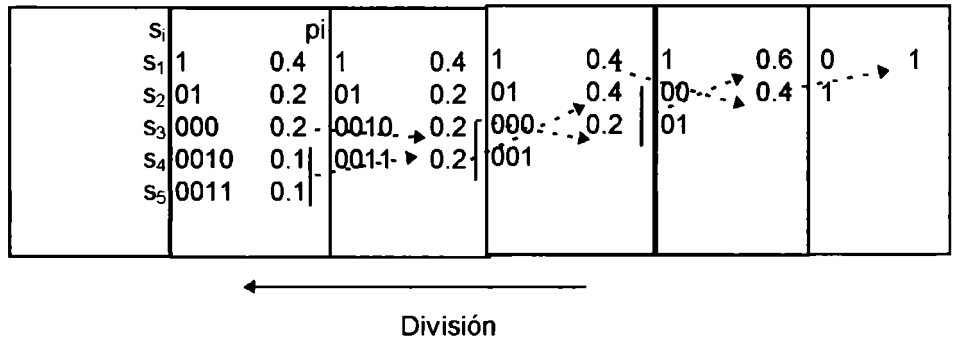
Demostración: Supongamos que existen. Si eliminamos uno de esos nodos y enganchamos a su hijo de su padre el código resultante sigue siendo instantáneo y su longitud promedio es menor a la del código original.

Ahora estamos en condiciones de probar que el código que generamos con el método descripto al comienzo del apéndice (la codificación de Huffman) es eficiente. Supongamos que no lo es. Es decir, que existe un código instantáneo asociado al conjunto de probabilidades dado cuya longitud promedio es menor a la longitud promedio de nuestro código.

Comparemos los árboles de decodificación de ambos códigos. Busquemos las dos hojas que representan el último bit de los dos símbolos menos probables. Si hay más de un par que cumpla esta condición tomamos cualquiera. Sabemos que debe existir al menos un par por la discusión sobre el largo de los símbolos menos probables. Ahora en el cálculo de la longitud promedio de nuestro código tenemos $l * (p_n + p_m)$ y en el del otro tenemos $l' * (p_n + p_m)$ donde l es la longitud de palabra de código máxima de nuestro código y l' es la longitud de palabra de código máxima del otro y donde p_n y p_m son las probabilidades de ocurrencia asociadas a los símbolos seleccionados. Juntemos estos dos símbolos en uno en ambos árboles y veamos que sucede en los nuevos códigos. En nuestro nuevo código esa rama contribuye $(l-1)(p_n + p_m)$ y en el otro nuevo código $(l'-1)(p_n + p_m)$. O sea que en ambos casos, si juntamos esas probabilidades en una generando un nuevo código con un símbolo menos, la longitud promedio de los dos códigos originales se acorta en $(p_n + p_m)$. Como la cantidad en que se acorta la longitud promedio es la misma para los dos códigos la diferencia que existía entre ambas se sigue manteniendo para las nuevas longitudes promedio. Si seguimos este procedimiento hasta que sólo tengamos dos símbolos, la longitud promedio de nuestro código es 1 mientras que la del otro debe ser menor que uno. Absurdo. Por lo tanto, el código de Huffman es un código instantáneo eficiente para un conjunto dado de probabilidades.

Vamos a describir mediante un ejemplo un método particular de construcción de un código de Huffman asociado a un conjunto de probabilidades. Existen varios métodos, algunos orientados a las personas (como el que vamos describir) y otros a la computadora. Todos son variaciones del método descripto al principio del Apéndice.





Notar que el algoritmo enunciado al principio de este apéndice es no determinístico. Ya que por un lado el conjunto de probabilidades mínimas puede llegar a contener mas de dos elementos lo que generaría mas de una codificación posible. Por otro lado, la asignación de 0s y 1s a los pares de probabilidades unidos es arbitraria (esto puede pensarse gráficamente como el intercambio de los hijos de un nodo del árbol de decodificación).

Notar que estos códigos son equivalentes teniendo en cuenta sólo la longitud promedio. Pero distintos métodos de construcción pueden llegar a generar códigos con distintas propiedades. Por ejemplo, si en el método del ejemplo cada vez que se unen dos probabilidades se hubiesen ubicado lo más arriba posible en la tabla, se genera un código de varianza mínima. Es decir, que la variabilidad en la longitud de las palabras del código es mínima entre todos los códigos equivalentes. También tiene la propiedad, no independiente de la anterior, de que la longitud de palabra máxima es mínima comparada con la longitud de palabra máxima de los demás códigos equivalentes.

En este punto vemos que todos los códigos eficientes generados con otros métodos son variantes de la codificación de Huffman.

Por último, vamos a dar una clasificación de los códigos de Huffman. Esta clasificación esta hecha con la intención de encuadrar el algoritmo de codificación que utilizamos (Huffman Semi-estático).

Estático: El histograma está calculado previamente a partir de lotes de entrenemiento (Por ejemplo, las variantes de Huffman utilizadas por los modems).

Semi estático: El histograma es calculado cada vez que se ejecuta el compresor.

Dinámico: En este método no se calcula el histograma ya que el código se construye dinámicamente a medida que van apareciendo los símbolos.

Existen otras variaciones del algoritmo de Huffman, por ejemplo, Huffman truncado o modificado, pero no las incluímos en nuestra clasificación porque nuestra idea no es hacer un clasificación exhaustiva sino una que permita ubicar al lector en la variación del algoritmo sobre la cual basamos parte de nuestro trabajo.

C3 La Transformada Burrows Wheeler (BWT)

En matemática, problemas difíciles a menudo pueden ser simplificados realizando un transformación en el conjunto de datos. Por ejemplo, programas de procesamiento digital de señales utilizan la transformada de Fourier para convertir datos de muestras de audio a conjuntos equivalentes de datos de frecuencia. Realizar un filtrado pasa bajos ahora sólo es cuestion de multiplicar los puntos por un factor de escala. Realizamos la transformación inversa y obtenemos la nueva forma de onda de audio correspondiente al filtrado.

Michel Borrow y David Wheeler publicaron los detalles de un nueva transformada que abre las puertas a nuevas y revolucionarias técnicas de compresión. La transformada Borrow Wheeler (BWT) reorganiza un bloque de datos en una forma que está muy bien caracterizada para ser comprimida.

Michel Borrow y David Wheeler publicaron un reporte en 1994 comentando el trabajo que habían estado haciendo en el Centro de Investigación de Sistemas Digital. Su paper, "A Block-sorting Lossless Data Compression Algorithm", presentó un algoritmo de compresión de datos basado en una transformada nunca publicada previamente descubierta por Wheeler en 1983.

Mientras que este paper discute un conjunto completo de algoritmos para la compresión y descompresión, el corazón del paper es la transformada BWT.

El algoritmo BWT reordena un bloque de datos usando un algoritmo de ordenación. El resultado es un bloque de datos que contiene exactamente los mismos datos que el bloque original, con la única diferencia en el orden de los datos. La transformación es reversible, significa que el orden inicial de los elementos puede ser restaurado sin pérdida de fidelidad.

La transformada BWT trabaja sobre bloques de datos a diferencia de los algortimos usuales que operan sobre uno o unos pocos bytes a la vez. La compresión que permite alcanzar esta transformada mejora a medida que aumenta el tamaño de bloque, cosa que generalmente no sucede con los algoritmos basados en diccionarios por la perdida de localidad del diccionario.

Notamos que el algoritmo en sí no comprime, solo reordena los datos de una manera que permite que otro algoritmo de compresión obtenga un alto grado de compresión. Obviamente muy superior al que obtendría trabajando sobre el bloque original.

Para ilustrar el funcionamiento del algoritmo utilizaremos el siguiente conjunto de datos:

TRABAJO

Figura 1: Nuestros datos de entrada

El algoritmo toma un bloque de datos y realiza una matriz de NxN (N tamaño del bloque), donde la fila 0 es el bloque original, la fila 1 es la rotación de la fila 0 hacia la izquierda en un caracter y así sucesivamente.

S0	TRABAJO
S1	RABAJOT
S2	ABAJOTR
S3	BAJOTRA
S4	AJOTRAB
S5	JOTRABA
S6	OTRABAJ

Figura 2: Matriz de strings rotados

Luego se ordenan lexicográficamente las filas.

	F L
S2	ABAJOTR
S4	AJOTRAB
S3	BAJOTRA
S5	JOTRABA
S6	OTRABAJ
S1	RABAJOT
S0	TRABAJO

Figura 3: Matriz de strings rotados ordenados

En este momento hay dos puntos que conviene resaltar. Primero, los strings han sido ordenados pero hemos llevado el rastro de la posición de cada string en la matriz original. Por lo tanto el String 0 (el string original) ha bajado hasta la fila 6 de la matriz. Segundo, hemos marcado la primera y última columna de la matríz con las letras F (First) y L (Last). La columna F contiene los caracteres del string original en forma ordenada; por lo tanto el string original TRABAJO está representado en F por AABJORT.

Los caracteres en la columna L, aparentemente están completamente desordenados, pero en realidad ellos respetan una interesenta propiedad: cada caracter en L es el prefijo del string que comienza en la misma fila en la columna F.

La salida de la BWT está compuesta por dos elementos: Una copia de la columna L y un entero indicando que fila de L contiene el primer caracter del string original(Indice primario). Por lo tanto la salida de BWT aplicada a nuestro string es RBAAJTO, y el indice primario 5.

El indice primario 5 facil de encontrar ya que el primer caracter del buffer se encontrará en la columna L en la fila de la matríz que contenga a S1. Por lo tanto ubicar la fila de L en la que está el primer caracter del buffer equivale a ubicar la fila de la matriz que contiene S1.

En este momento surgen dos preguntas. Primero, no parece que esta sea una transformación reversible. Generalmente, una función sort() no viene con un unsort() asociado que restaure el orden original. Segundo, que es lo bueno que esta extraña transformación provee.

Para obtener el buffer original a partir de la columna L se requiere el uso de un vector de transformación, este es un arreglo que define el orden en que los strings rotados fueron reordenados a través de las filas de la matriz.

El vector de transformación, digamos T, es un arreglo con un índice para cada fila. Para una fila dada j, T[j] se define como la fila donde se encuentra S_{i+1} suponiendo que S_i se encuentra en la fila j. En la Figura 3, la fila 6 contiene S_0, el string original de entrada, y la fila 5 contiene S_1, el string

original rotado un caracter a la izquierda. Por lo tanto, T[6] contiene el valor 5. S_2 es encontrado en la fila 0, por lo tanto T[5] contiene el valor 0. Para nuestro ejemplo, el vector de transformación es [2, 3, 1, 4, 6, 0, 5].

La Figura 4 muestra como es usado el vector de transformación para recorrer las distintas filas. Para cualquier fila que contiene S_i, el vector provee el valor de la fila donde se encuentra S_{i+1}.

El vector de transformación da la clave para restaurar L a su orden original. Dado L y el índice primario, se puede restaurar el S_0 original. Dada una copia de L, usted puede calcular el vector de transformación para la matriz original de entrada. Y consecuentemente, dado el índice primario, usted puede recrear S_0.

La clave para recrear el vector de transformación es que usted puede hacerlo si conoce L y F. Además, si usted tiene L, tiene F (L ordenada lexicográficamente). Por lo tanto, usted puede generar el vector de transformación a partir de L.

Calculemos T para nuestro conjunto de trabajo

	LF
	R
	B
	A
	A
	J
	T
	O

Figura 4: Estado inicial en la reconstrucción de T dado L

Como vemos en la Figura 4, dada una copia de L no conocemos mucho acerca de la matriz original (movimos la L a la primera columna con propósitos de ilustración). Podemos obtener F ordenando los caracteres en L.

	LF
	RA
	BA
	AB
	AJ
	JO
	TR
	OT

Figura 5: Reconstrucción de T dado L

Ahora comenzamos el cálculo de T. El caracter R en la fila 0 de L se mueve a la fila 5 de la columna F lo que significa que el valor de T[5] es 0. El caracter B en la fila 1 se mueve a la fila 2 por lo que T[2] es 1. Pero que pasa con la letra A en la fila 2 ? La A aparece más de una vez en ambas columnas (en particular en la columna F).

La elección no es ambigua. A pesar de que el proceso de decisión no es intuitivo. Por definición, la columna F está ordenada y todos los strings comenzando con A en la columna L también están ordenados. Por que ? Todos los strings que comienzan con el mismo caracter están ordenados según su segundo caracter (más precisamente, por el todo el substring restante).

Ahora, cuando dichos caracteres aparezcan en L, estarán ordenados de acuerdo al mismo substring antes mencionado.

Este análisis se puede generalizar para cualquier caracter x:

...
xSS1
xSS2
xSS3
...
SS1x
...
SS2x
...
SS3x

Siguiendo este proceso obtenemos el siguiente vector de transformación: [2, 3, 1, 4, 6, 0, 5]. Les parece familiar este vector ?

Una vez que esta dificultad ha sido esclarecida recuperemos S_0 utilizando el siguiente algoritmo:

```
int T[] = { 2, 3, 1, 4, 6, 0, 5 };
char L[] = "RBAAJTO";
int indice_primario = 5;

void decode()
{
        int index = indice_primario;

        for(int i = 0; i < strlen(L); i++) {
                printf("%c", L[index]);
                index = T[index];
        }
}
```

La salida de este programa es TRABAJO.

Hemos contestado la primera de las dos preguntas que nos formulamos arriba. Ahora pasaremos a contestar la segunda que se refería a los beneficios que esta transformación puede ofrecernos. Para hacerlo veamos el siguiente cuadro, el cual corresponde a la aplicación de la transformada a este documento:

La tabla tiene 2 columnas:
- L: salida de la transformada (truncada)
- S_i: string asociado al caracter de salida (L es el último caracter del string S_i)

S_i	L
avid Wheeler.<13><10><13><10>El algoritmo	D
bajo "Compresi<243>n de datos que	**a**
bamos con algunos ejemplos.<13><10><13>	**o**
blicada en el paper "A Block-s	**u**
blicado por Michel Borrow y Da	**u**
bloque de datos que contiene e	
bloque de datos usando un algo	
bloque original, con la unica	
bre una red virtual de procesa	**o**
buffer, scatter , gatter,etc.	
buffereados, sin buffer, scatt	
buida. Una comparaci<243>n con Soc	**i**
c, asinc, buffereados, sin buf	n
c, buffereados, sin buffer, sc	n

Concentremos nuesta atención en la sección de la salida correspondiente a todos los string que comienzan con el caracter b. No es sorprendente que el prefijo de los que comienzan con bloque * sea un blanco. O que el prefijo de los que comienzan con blicad* sea una u. O sea que la salida de la transformada tiene el siguiente formato:

prefijos del SR_0, prefijos de SR_1, ...

donde SR_i denota el string correspondiente a la fila i de la matriz ordenada.

Notar que son prefijos de un string y no de un caracter en particular. Por ejemplo, no es lo mismo trabajar sobre los prefijos de r que sobre los prefijos de "rabajo de grado". Obviamente, el segundo conjunto está mucho más acotado.

La idea en este momento es utilizar compresores que exploten el formato de la salida de la BWT. Por ejemplo, los radios de compresión resultantes de aplicar un Huffman Semi-estático al archivo original y al archivo BWT-transformado correspondiente son los mimos ya que el histograma asociado a ambos es el mismo por lo que no tendría sentido aplicar la transformada.

Este es un punto bastante poco explotado, pero un ejemplo de uno de esos algoritmos es el que sugieren los autores, el MTF (Move To Front), seguido de un codificador de la entropía (Xej Huffman).

El codificador MTF es otra transformación reversible, y tampoco comprime por sí mismo sino que, al igual que la BWT deja los datos en un formato muy apropiado para una posterior compresión. Este codificador mantiene todos, los 256, posibles caracteres en una lista. Cada vez que uno de ellos está por ser "enviado", se envía la posición de ese caracter en la lista y luego este es puesto al frente de la lista. Si la entrada del MTF es la salida de la BWT la salida del MTF será un conjunto de enteros pequeños, algunas corridas de 0 y algún que otro entero grande correspondiente, probablemente a la primera aparición de un caracter.

Como ejemplo, la salida MTF correspondiente a la sección de L en negrita de nuestro ejemplo ("aouu o i") es, [97, 111, 117, 0, 32+3, 0, 0, 2, 1, 0, 105+2]. Notemos que no se ha ganado demasiado debido al tamaño de la entrada. Estas transformadas mejoran a medida que aumenta el tamaño de la entrada. Como ejemplo, piense que pasaría si aplicaramos estas transformadas a un libro.

Herramientas

H1 UNIX

En 1965 se formó un grupo entre los laboratorios Bell, la compañía General Electric y el (Massachusett Institute of Technology) para desarrollar un nuevo sistema operativo llamado Multics. Los objetivos del sistema operativo Multics fueron proveer acceso simultáneo a una gran comunidad de usuarios, para ampliar el poder de computación y de almacenamientop de datos y para permitir a los usuarios compartir fácilmente sus datos, si lo deseaban. Mucha gente de los laboratorios Bell que luego formó parte del desarrollo inicial de UNIX participó en el proyecto Multics. A pesar de que una primitiva versión del sistema Multics corría en una computadora GE 645 por el año 1969, esta no alcanzaba los objetivos para los cuales había sido creada por lo tanto los laboratorios Bell abandonaron el proyecto.

Con la finalización del trabajo en el proyecto Multics miembros de los laboratorios Bell se quedaron sin un servicio computacional interactivo conveniente. En un intento de mejorar su ambiente de programación, Kent Thompson, Dennis Ritchie, y otros realizaron un borrador de un paper de un diseño de un sistema de archivos que mas tarde evolucionó en una temprana versión del sistema de archivos de UNIX. Thompson escribió programas que simulaban el comportamiento del sistema de archivos propuestos y de programas en un ambiente con demanda de páginas, y además codificó un kernel simple para la computadora GE 645. Al mismo tiempo, el escribió un juego, "Space Travel", en Fortran para un sistema GECOS pero el programa era insatisfactorio porque era difícil controlar la nave espacial y su ejecución era muy costosa. Thompson más tarde encontró una computadora usada PDP-7 que proveía buenos gráficos y bajo costo de ejecución. La programación del juego le permitió a Thompson aprender sobre esta máquina, pero su entorno de desarrollo era precario ya que debía realizar su trabajo sobre la GECOS y llevar el código generado a la PDP-7. Para crear un ambiente de desarrollo mejor, Thompson y Ritchie implementaron su diseño del sistema sobre la PDP-7. Esto incluía una primitiva versión del sistema de archivos de UNIX, el subsistema de control de procesos, y un conjunto reducido de programas utilitarios. Eventualmente, el nuevo sistema no necesitó mas el entorno de desarrollo GECOS y se soportaba a sí mismo. Al nuevo sistema se le dió el nombre de UNIX, una variación de la palabra Multics realizada por otro miembro de los laboratorios Bell, Brian Kernighan.

A pesar de que esta temprana versión del sistema UNIX era muy prometedora, no se pudo explotar su potencial hasta que fue usada en un proyecto real. En 1971, el UNIX fue migrado a una _PDP-11 para proveer un sistema de procesamiento de texto para el departamento de patentes de los laboratorios Bell. El sistema se caracterizaba por su pequeño tamaño: 16K del sistema, 8K para programas de usuario, un disco de 512K con un límite de 64K por archivo. Luego de este éxito inicial, Thompson se dedicó a implementar un compilador Fortran para el nuevo sistema, pero en cambio terminó desarrollando un intérprete para el lenguaje B, influenciado por BPCL. B era un lenguaje interpretado con todas las desventajas de performance que esto implica, por lo tanto, Ritchie desarrolló un lenguaje llamado C que permitía la generación de código de máquina, declaración de tipos de datos y definición de estructuras de datos. En 1973, el sistema operativo fue reescrito en C, algo nunca antes realizado, que tuvo un gran impacto. El número de instalaciones en los laboratorios Bell creció hasta 25, y se creó el Grupo de Sistema UNIX para proveer soporte interno.

Por ese entonces, AT&T no podía comercializar productos de computación debido a un acuerdo que había firmado con el gobierno federal en los años 1956, pero proveía el sistema UNIX a universidades que lo requerían con propósitos educativos. AT&T nunca anunció, ni hizo propoganda, ni realizó soporte del sistema en adhesión al acuerdo que había firmado. De cualquier manera, la popularidad del sistema creció rápidamente. En 1974 Thompson y Ritchie publicaron un paper describiendo el sistema UNIX, dando un mayor impulso a su aceptación. Por 1977, el número de sistemas UNIX había crecido hasta 500 de los cuales 125 eran en universidades. El sistema UNIX se volvió popular en las compañias telefónicas proveyendo un buen entorno para el desarrollo de programas, servicios de transacciones en red y servicios en tiempo real. Fueron dadas licencias para

el sistema a instituciones comerciales y universidades. En 1977 Interactive System Corporation se convirtió en el primer vendedor de valor agregado del sistema UNIX, mejorándolo para el uso en entornos de automatización de oficinas. 1977 también marcó el año en que el sistema operativo UNIX fue por primera vez portado a máquinas no PDP (esto es, se hizo correr en otras máquinas con poco o nigún cambio).

Con el crecimiento de popularidad de los microprocesadores, otras compañias portaron el sistema a nuevas máquinas pero su simplicidad y claridad tentó a varios desarrolladores a extenderlo según su modo de ver lo que generó muchas variantes del sistema básico. En el período de 1977 a 1982 los laboratorios Bell combinaron varias variantes AT&T en un sistema simple conocido comercialmente como UNIX System III. Los laboratorios Bell, luego agregaron varias características al System III y lo llamaron System V, y AT&T anunció soporte oficial para este nuevo sistema en Enero de 1983. A pesar de que gente en la universidad de Berkeley había desarrollado una variante del sistema UNIX, cuya más reciente versión llamada 4.3 BSD para máquinas VAX, proveían nuevas e interesantes características.

En los comienzos de 1984, existían alrededor de 100000 instalaciones del sistema en el mundo, corriendo sobre gran variedad de máquinas que iba desde microprocesadores a mainframes de diferentes fabricantes. Diversas razones para la popularida y el éxito del sistema UNIX han sido sugeridas:

- El sistema está escrito en un lenguaje de alto nivel, haciéndolo fácil de leer, entender, modificar, y portar a otras máquinas, Ritchie estimó que el primer sistema en C era de un 20 a un 40% más grande y lento debido a que no estaba escrito en assembler, pero la ventaja de usar un lenguaje de alto nivel opacó estas desventajas.

- Tenía una simple interfase de usuario la cual tenía el poder de proveer los servicios que los usuarios deseaban.

- Proveía primitivas que permitían construir complejos programas a partir de programas más simples.

- Usaba un sistema de archivos jerárquico que permitía un fácil mantenimiento y una eficiente implementación.

- Usaba un formato de archivo consistente, el byte stream, haciendo fácil la escritura de aplicaciones.

- Proveía una interfase simple y consistente hacia dispositivos periféricos.

- Es un sistema multiusuario y multiproceso; cada usuario puede ejecutar varios procesos simultáneamente.

- Oculta la arquitectura de la máquina al usuario, facilitando la escritura de programas que corran sobre diferente hardware.

La filosofía del sistema es simplicidad y consistencia.

H2 LINUX

Linux es una implementación de la especificación POSIX del sistema operativo UNIX, con extensiones SYSV y BSD, que ha sido escrita integramente. Linux es libremente distribuible bajo las condiciones de la licencia pública GNU. Inicialmente trabaja sobre PCs compatibles IBM con procesador 386 o mayor.

Además, trabaja sobre otras plataformas (Amiga, Atari, PowerPc y otras).

El kernel de Linux es escrito por Linus Torvalds y por otros voluntarios. Muchos de los programas que corren sobre Linux son freeware, la mayoría de ellos pertenecientes al proyecto GNU.

Tiene todas las características que uno esperaría de un moderno y flexible UNIX, incluyendo verdadero multitasking, memoria virtual, librerías compartidas, carga bajo demanda, ejecutables compartidos copy-on-write, manejo de memoria y soporte de red TCP/IP entre otras cosas.

H3 El Lenguaje C

C es un lenguaje de programación de propósito general que ha sido estrechamente asociado con el sistema UNIX en donde fue desarrollado puesto que tanto el sistema como los programas que corren en él están escritos en lenguaje C. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina, y aunque se le llama "lenguaje de programación de sistemas" debido a su utilidad para escribir compiladores y sistemas operativos, se utiliza con igual eficacia para escribir importantes programas en diversas disciplinas.

Muchas de las ideas importantes de C provienen del lenguaje BCPL , desarrollado por Martin Richards. La influencia de BCPL sobre C se continuó indirectamente a través del lenguaje B, el cual fue escrito por Ken Thompson en 1970 para el primer sistema UNIX de la DEC PDP - 7.

BCPL y B son lenguajes "carentes de tipos". En contraste, C proporciona una variedad de tipos de datos. Los tipos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños. Además, existe una jerarquía de tipos de datos derivados, creados con apuntadores, arreglos estructuras y uniones. Las expresiones se forman a partir de operadores y operandos; cualquier expresión, incluyendo una asignación o una llamada a función , puede ser una proposición. Los apuntadores proporcionan una aritmética de direcciones independientes de la máquina.

C proporciona las construcciones fundamentales de control de flujo que se requieren en programas bien estructurados: agrupación de proposiciones, toma de decisiones (if-else), selección de un caso entre un conjunto de ellos (switch), iteración con la condición de paro en la parte superior (while, for) o en la parte inferior (do), y terminación prematura de ciclos (break).

Las funciones pueden regresar valores de tipos básicos, estructuras, uniones o apuntadores. Cualquier función puede ser llamada recursivamente. Las variables locales son normalmente "automáticas", o creadas de nuevo con cada invocación. La definición de una función no puede estar anidada , pero las variables pueden estar declaradas en una modalidad estructurada por bloques. Las funciones de un programa en C pueden existir en archivos fuente separados, que se compilan de manera separada. Las variables pueden ser internas a una función, externas pero conocidas sólo dentro de un archivo fuente, o visibles al programa completo.

Un paso de preprocesamiento realiza substitución de macros en el texto del programa, inclusión de otros archivos fuente y compilación condicional.

C es un lenguaje de relativo "bajo nivel" . Esta caracterización no es peyorativa, simplemente significa que C trata con el mismo tipo de objetos que la mayoría de las computadoras, llámense caracteres, números y direcciones. Estos pueden ser combinados y cambiados de sitio con los operadores aritméticos y lógicos implantados por máquinas reales.

C no proporciona operaciones para tratar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas o arreglos. No existen operaciones que manipulen un arreglo o una cadena completa, aunque las estructuras pueden copiarse como una unidad. El lenguaje no define ninguna facilidad para asignación de almacenamiento que no sea la de definición estática y la disciplina de pilas provista por las variables locales de funciones; no emplea *heap* ni recolector de basura. Finalmente , C en sí mismo no proporciona capacidades de entrada/salida; no hay proposiciones READ o WRITE, ni métodos propios de acceso a archivos. Todos esos mecanismos de alto nivel deben ser proporcionados por funciones llamadas explícitamente.

De manera semejante, C solamente ofrece un control de flujo franco, y lineal: condiciones, ciclos, agrupamientos y subprogramas, pero no multiprogramación, operaciones paralelas, sincronización ni co-rutinas.

Aunque la ausencia de alguna de esas capacidades puede parecer como una grave deficiencia ("significa que se tiene que llamar a una función para comparar dos cadenas de caracteres?") , el mantener al lenguaje de un tamaño modesto tiene beneficios reales. Puesto que C es relativamente pequeño, se puede describir en un pequeño espacio y aprenderse con rapidez. Un programador puede razonablemente esperar conocer, entender y utilizar en verdad la totalidad del lenguaje.

Por muchos años, la definición de C fue el manual de referencia de la primera edición de *El lenguaje de programación* C. En 1983, el American National Standards Institute (ANSI) estableció un comité para proporcionar una moderna y comprensible definición de C. La definición resultante, el estándar ANSI o "ANSI C" , se esperaba fuera aprobada a fines de 1988. La mayoría de las características del estándar ya se encuentran soportadas por compiladores modernos.

El estándar está basado en el manual de referencia original. El lenguaje ha cambiado relativamente poco; uno de los propósitos del estándar fue asegurar que la mayoría de los programas existentes pudiesen permanecer válidos o, al menos, que los compiladores pudieran producir mensajes de advertencia acerca del nuevo comportamiento.

Para la mayoría de los programadores, el cambio más importante es una nueva sintaxis para declarar y definir funciones. Una declaración de función ahora puede incluir una descripción de los argumentos de la función; la sintaxis de la definición cambia para coincidir. Esta información extra permite que los compiladores detecten más fácilmente los errores causados por argumentos que no coinciden; de acuerdo con nuestra experiencia, es una adición muy útil al lenguaje.

Existen otros cambios de menor escala en el lenguaje. La asignación de estructuras y enumeraciones, que ha estado ampliamente disponible, es ahora parte oficial del lenguaje. Los cálculos de punto flotante pueden ahora realizarse con precisión sencilla. Las propiedades de la aritmética, especialmente para tipos sin signo, están esclarecidas. El procesador es más elaborado. La mayor parte de esos cambios sólo tendrán efectos secundarios para la mayoría de los programadores.

Una segunda contribución significativa del estándar es la definición de una biblioteca que acompañe a C. Esta especifica funciones para tener acceso al sistema operativo (por ejemplo, leer de archivos y escribir en ellos), entrada y salida con formato, asignación de memoria, manipulación de cadenas y otras actividades semejantes. Una colección de encabezadores (headers) estándar proporcionan un acceso uniforme a las declaraciones de funciones y tipos de datos. Los programas que utilizan esta biblioteca para interactuar con un sistema anfitrión están asegurados de un comportamiento compatible. La mayor parte de la biblioteca está estrechamente modelada con base en la "biblioteca E/S estándar" del sistema UNIX .

Debido a que los tipos de datos y estructuras de control provistas por C son manejadas directamente por la mayoría de las computadoras, la biblioteca de ejecución (run- time) requerida para implantar programas autocontenidos es pequeña. Las funciones de la biblioteca estándar únicamente se llaman en forma explícita, de manera que se pueden evitar cuando no se necesitan. La mayor parte puede escribirse en C, y excepto por detalles ocultos del sistema operativo, ellas mismas son portátiles.

Aunque C coincide con las capacidades de muchas computadoras, es independiente de cualquier arquitectura. Con un poco de cuidado es fácil escribir programas portátiles , esto es, programas que pueden correr sin cambios en una variedad de máquinas. El estándar explica los problemas de la transportabilidad, y prescribe un conjunto de constantes que caracterizan a la máquina en la que se ejecuta el programa.

C no es un lenguaje fuertemente tipificado, sino que, al evolucionar, su verificación de tipos ha sido reforzada. La definición original de C desaprobó, pero permitió, el intercambio de apuntadores y enteros; esto se ha eliminado y el estándar ahora requiere la adecuada declaración y la conversión explícita que ya ha sido obligada por los buenos compiladores. La nueva declaración de funciones es otro paso en esta dirección. Los compiladores advertirán de la mayoría de los errores de tipo, y no hay conversión automática de tipo de datos incompatibles. Sin embargo, C mantiene la filosofía básica de que los programadores saben lo que están haciendo; sólo requiere que establezcan sus intenciones en forma explícita.

Como cualquier otro lenguaje, C tiene sus defectos. Algunos de los operadores tienen la precedencia equivocada; algunos elementos de la sintaxis pueden ser mejores. A pesar de todo, C a probado ser un lenguaje extremadamente efectivo y expresivo para una amplia variedad de programas de aplicación.

H4 Linux Interprocess Communications: System V

Introducción

Las facilidades IPC (Inter-process communication) de Linux proveen un método para que múltiples procesos se comuniquen unos con otros. Existen diversos métodos de IPC disponibles para los programadores C en Linux:

* Half-duplex UNIX pipes
* FIFOs (named pipes)
* SYSV style message queues
* SYSV style semaphore sets
* SYSV style shared memory segments
* Networking sockets (Berkeley style)

Estas facilidades, si son usadas eficientemente, proveen un ambiente de trabajo adecuado para el desarrollo de soluciones cliente/servidor en cualquier sistema UNIX (incluyendo Linux).

Half-duplex UNIX Pipes

Conceptos Básicos

Simplemente expresado, un pipe es un método para conectar la salida estandar de un proceso a la entrada estandar de otro. Los pipes son la más vieja de las herramientas IPC, apareciendo desde las más tempranas implementaciones del sistema operativo UNIX. Ellas proveen un método de comunicación uni-direccional (de aquí half-duplex) entre procesos.

Este servicio es ampliamente usado, aun en línea de comandos de UNIX (en el shell).

```
ls | sort | less
```

Lo de arriba setea un pipeline, tomando la salida del **ls** como la entrada de **sort** y la salida de **sort** como la entrada de **less**. Los datos corren a través de un half-duplex pipe, viajando (visualmente) de izquierda a derecha a lo largo del pipeline.

Aunque la mayoría de los usuarios de UNIX usan pipes (generalmente en la línea de comandos o en scrips), a menudo lo hacen sin reparar en lo que ocurre a nivel del kernel.

Cuando un proceso crea un pipe, el kernel aloca dos descriptores de archivos para el uso del pipe. Un descriptor es usado para entrar datos al pipe (write) y otro para sacar datos del pipe (read). En este punto, el pipe tiene poco uso práctico, ya que el proceso creador puede usar el pipe sólo para comunicarse con sí mismo. Sin embargo, existe un objetivo mucho más interesante que el simple hecho descripto arriba. Mientras que un pipe inicialmente conecta a un proceso con sí mismo, los datos que viajan por el pipe se mueven a través del kernel. Bajo Linux, en particular, los pipes son representados internamente con un inodo válido. Por supuesto, este inodo está dentro del kernel, y no pertenece a ningún file system físico. Este punto en particular abrirá algunas puertas interesantes de I/O para nosotros, como veremos más adelante.

En este momento el pipe tiene poco uso. Después de todo, porqué tomarse el trabajo de crear un pipe si sólo vamos a hablar con nosotros mismos ? En la práctica, el proceso creador típicamente forkea un proceso hijo. Ya que un proceso hijo heredará todos los descriptores de archivos abiertos

del padre, ahora tenemos las bases para la comunicación interprocesos (entre el padre y el hijo). En este momento, ambos procesos tienen acceso a los descriptores de archivo que constituyen el pipeline. Es ahora que debe tomarse una desición crítica. En qué dirección queremos que viajen los datos ? El proceso hijo le manda información al padre o al revés ? Los dos procesos se ponen de acuerdo, y proceden a cerrar el extremo del pipe que no les concierne.

La construcción del pipeline esta ahora completa ! La única cosa que resta por hacer es utilizarla. Para acceder un pipe directamente, las mismas sytem calls que son usadas para low-level file I/O pueden ser utilizadas (recuérdese que los pipes son representados internamente como un inodo válido).

Para enviar datos al pipe, usamos la llamada al sistema write(). Recordar, las llamadas al sistema para I/O sobre archivos a bajo nivel trabajan con descriptores de archivo ! Sin embrago, tener en mente que ciertas llamadas al sistema, tal como lseek(), no trabajan con descriptores de archivos asociados a pipes.

Creando Pipes en C

Para crear un pipe en C, utilizamos la llamada al sistema pipe(). La cual toma un sólo argumento, el cual es un arreglo de dos enteros, y si la llamada es exitosa, el arreglo contendrá dos nuevos descriptores de archivo a ser usados por el pipeline. Luego de crear un pipe, el proceso tipicamente dispara un nuevo proceso (recuerde que el hijo hereda los descriptores de los archivos abiertos). Para disparar un nuevo proceso se utilizan las llamadas al sistema fork y probablemente exec*.

El primer entero en el arreglo (elemento 0) es abierto para lectura, mientras que el segundo entero (elemento 1) es abierto para escritura. Visualmente hablando, la salida de fd[1] es la entrada de fd[0]. Una vez más, todos los datos que viajan a través del pipe se mueven a través del kernel.

Si el padre desea recibir datos del hijo, el debe cerrar fd[1], y el hijo debe cerrar fd[0]. Si el padre quiere enviarle datos al hijo, debe cerrar fd[0] y el hijo debe cerrar fd[1]. Ya que estos descriptores son compartidos entre el padre y el hijo, debemos siempre aseguramos de cerrar el extremo del pipe que no nos concierne. Como una nota técnica, el EOF nunca será retornado si los extremos innecesarios del pipe no son cerrados.

Como mencionamos previamente, una vez que el pipeline fue establecido, los descriptores de archivo pueden ser tratados como descriptores a archivos normales (con algunas restricciones).

A menudo, los descriptores en el hijo son duplicados (dup() y dup2()) en entrada estandar o salida estandar. El hijo luego puede exec() otro programa, el cual hereda los streams estandars.

Pipes, la Forma Fácil !

Si los pasos descriptos arriba le parecieron una forma demasiado complicada de crear y utilizar pipes, existe una alternativa: popen().

Esta función de la librería estandar crea un pipeline half-duplex llamando a pipe internamente. Luego forkea un proceso hijo, execs el Bourne shell, y ejecuta el comando que se le pasó como primer argumento en el shell. La dirección del flujo de datos es determinada por el segundo argumento. El puede ser "r"o "w", por "read" o "write". No puede ser ambos !

Mientras que esta función realiza gran parte del trabajo sucio para usted, usted pierde el fino control que una vez tuvo utilizando la llamada al sistema pipe(), y manejando el par fork/exec usted mismo.

Una nota interesante es que, ya que el Bourne shell es usado directamente, se realiza la expansión de metacaracteres (por ejemplo wildcards) del shell usual sobre el primer parámetro.

Pipes creadas con popen() deben ser cerradas con pclose(). La función pclose() realiza un wait4() sobre el proceso forkeado por popen(). Cuando retorna, destruye el pipe y el file stream pasado como parámetro.

Notas sobre Half-duplex Pipes

* Pipes bi-direccionales pueden ser creadas utilizando dos pipes, y asignando apropiadamente los descriptores de archivos.

* La llamada al sistema pipe() debe ser hecha ANTES de la llamada fork(), o los descriptores no serán heredados por el hijo !.

* Con half-duplex pipes, cualquier par de procesos conectados deben compartir un ancestro. Esto se puede evitar utilizando named pipes (FIFOs).

Named Pipes (FIFOs - First In First Out)

Conceptos Básicos

Un named pipe trabaja muy parecido a un pipe regular, pero tiene algunas diferencias notables.

* Los named pipes existen como un archivo especial de dispositivo en el file system.

* Procesos "sin un ancestro común" pueden compartir datos a través de un named pipe.

* Cuando toda la I/O es terminada por los procesos que lo comparten, el named pipe permanece en el file system para ser usado en el futuro.

Creando un FIFO

Hay diversas formas de crear un named pipe. Las primeras dos pueden ser hechas directamente desde el shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

Los dos comandos de arriba realizan operaciones idénticas, con una excepción. El comando mkfifo permite alterar los permisos sobre el archivo FIFO directamente luego de la creación. Con mknod, una llamada al comando chmod será necesaria.

Los archivos FIFO pueden ser rápidamente identificados en un file system físico por el indicador "p" que se muestra en un listado de directorio largo:

```
$ ls -l MYFIFO
prw-r--r-- 1 root root 0 Dec 14 22:15 MYIFFO |
```

También nótese la barra vertical ("pipe sign") ubicada inmediatamente después del nombre de archivo.

Para crear un FIFO en C, podemos hacer uso de la llamada al sistema mknod().

Operaciones sobre archivos FIFO

Las operaciones de I/O sobre un FIFO son escencialmente las mismas que sobre pipes normales, con una excepción. Se debe usar una llamada al sistema o función de librería open para abrir físicamente un canal al pipe. Con half-duplex pipes esto no es necesario, ya que el pipe reside en el kernel y no en un file system físico. También se puede tratar al pipe como un stream, abriéndolo con fopen() y cerrándolo con fclose().

System V IPC

Conceptos Fundamentales

Con el System V, AT&T introdujo tres formas de IPC (colas de mensajes, semáforos y memoria compartida). Mientras que el comité POSIX aun no ha completado la estandarización de estas facilidades, muchas implementaciones las soportan. Por otro lado, Berkeley (BSD) usa sockets como su principal forma de IPC, en vez de los elementos del System V. Linux permite utilizar ambos métodos de IPC (BSD y System V). En este Apéndice nos referiremos a las tres facilidades System V. Para leer información sobre sockets referirse al Apéndice BSD IPC.

Identificadores IPC

Cada objeto IPC tiene un único identificador IPC asociado. Cuando decimos objeto IPC, estamos hablando de una única cola de mensajes, un conjunto de semáforos, o un segmento de memoria compartida. Este identificador es utilizado dentro del kernel para identificar univocamente un objeto IPC. Por ejemplo, para acceder a un segmento de memoria compartida en particular, el único item que usted necesita es el valor único que ha sido asignado a ese segmento.

La unicidad de un identificador es relevante para el tipo de objeto en cuestión. Para ilustrar esto, asuma un identificador numérico de 12345. Mientras que nunca habrá dos colas de mensajes con este mismo identificador, existe la posibilidad de que una cola de mensajes y, digamos, un segmento de memoria compartida, tengan el mismo identificador numérico.

Llaves IPC

Para obtener un ID único, una llave debe ser usada. La llave debe ser mutuamente aceptada por ambos, el proceso cliente y el proceso servidor.

La llave puede ser el mismo valor todo el tiempo, mediante el hardcoding del valor de la llave dentro de la aplicación. Esto tiene la desventaja de que el valor de la llave puede estar siendo usado por otra aplicación. A menudo, la función ftok() es usada para generar valores de llaves para ambos,

el cliente y el servidor. Con esto, una aplicación puede chequear por colisiones y reintentar la generación del valor de la llave.

El uso de la función ftok() no es indispensable. El algoritmo de generación de valores de llave usado queda a total discreción del programador de la aplicación.

El valor de la llave, como quiera que sea obtenido, es usado en las llamadas al sistema IPC subsecuentes para crear o acceder a objetos IPC.

El comando ipcs

El comando ipcs puede ser usado para obtener el estatus de todos los objetos IPC System V.

El comando ipcs es una herramienta muy poderosa que provee una abertura para espiar dentro de los mecanismos de almacenamiento que tiene el kernel para los objetos IPC.

El comando ipcrm

El comando ipcrm puede ser usado para eliminar un objeto IPC del kernel. Mientras que los objetos IPC pueden ser eliminados vía llamadas al sistema en el código del usuario, la necesidad de eliminar objetos IPC manualmente a menudo aparece, especialmente bajo ambientes de desarrollo.

Simplemente se debe especificar si el objeto a eliminar es una cola de mensajes, un conjunto de semáforos, o un segmento de memoria compartida. Luego se debe especificar el IPC ID del objeto, el cual se puede obtener mediante el comando ipcs. Uno debe especificar el tipo de objeto ya que los identificadores son únicos sólo sobre objetos de un tipo.

Colas de Mensajes

Conceptos Básicos

Las colas de mensajes pueden ser descriptas como una lista encadenada dentro del espacio de direccionamiento del kernel. Los mensajes pueden ser enviados a la cola en orden y ser retirados de ella de diversas maneras. Cada cola de mensaje es unívocamente identificada por un identificador IPC.

Para crear una nueva cola de mensajes, o para ganar acceso a una cola ya existente, se usa la llamada al sistema msgget().

Una vez que tenemos el identificador de la cola, podemos comenzar a realizar operaciones sobre ella. Para enviar un mensaje a una cola, usted usa la llamada al sistema msgsnd().

Para retirar un mensaje de la cola, usted usa la llamada al sistema msgrcv().

Para realizar operaciones de control sobre una cola de mensajes usted puede usar la llamada al sistema msgctl(). Esta llamada al sistema sólo debe ser usada cuando sea indispensable porque un error puede romper todo el subsistema System V IPC.

Semáforos

Conceptos Básicos

Los semáforos pueden ser descriptos como contadores usados para controlar el acceso a recursos compartidos por múltiples procesos. Ellos son usados a menudo como un mecanismo de lockeo para prevenir que procesos accedan un recurso mientras otro proceso está realizando operaciones sobre él.

Los semáforos están implementados como conjuntos, más que como entidades simples. Por supuesto, un conjunto de semáforos puede contener sólo un semáforo.

Para crear un nuevo conjunto de semáforos, o para ganar acceso a uno ya existente, se usa la llamada al sistema semget().

Una vez que tenemos el identificador del conjunto, podemos comenzar a realizar operaciones sobre él. Para esto usted usa la llamada al sistema semop().

Para realizar operaciones de control sobre un conjunto de semáforos usted puede usar la llamada al sistema semctl().

Memoria Compartida

Conceptos Básicos

La memoria compartida puede ser descripta como un área (segmento) de memoria que puede ser direccionada y compartida por más de un proceso. Esta es, por lejos, la forma de IPC más rápida, porque no hay intermediación. Un segmento puede ser creado por un proceso y subsecuentemente escrito y leido por cualquier número de procesos.

Para crear un nuevo segmento de memoria compartida, o para ganar acceso a uno ya existente, se usa la llamada al sistema shmget().

Una vez que un proceso tiene un identificador IPC válido para un segmento dado, el próximo paso es attachear o mapear el segmento dentro de su propio espacio de direccionamiento. Para esto usted usa la llamada al sistema shmat().

Una vez que el segmento ha sido propiamente attacheado, y el proceso tiene un puntero al comienzo de ese segmento, escribir y leer sobre el segmento es tan fácil como referenciar o desreferenciar el puntero (lo usual para un puntero).

Para realizar operaciones de control sobre un segmento de memoria compartida usted puede usar la llamada al sistema shmctl().

Para desattachear un segmento un proceso utiliza la llamada al sistema shmdt

H5 Linux Interprocess Communications: BSD

Introducción

El bloque de contrucción básico para la comunicación en un sistema operativo tipo UNIX es el socket. Un socket es un punto final de comunicación. Un socket en uso tiene un tipo y uno o varios procesos asociados. Sockets existen dentro de un *Dominio de Comunicación*. Un *Dominio de Comunicación* es una abstracción para agrupar las propiedades comunes de procesos comunicándose a través de sockets. Los sockets generalmente intercambian datos con sockets en el mismo dominio aunque puede haber excepciones. Las actuales versiones de UNIX soportan 3 tipos de dominios: El dominio UNIX para comunicación inter procesos, el dominio Internet que es usado por procesos que se comunican usando el protocolo DARPA (TCP/IP) y el dominio NS que utiliza el protocolo standard de XEROX.

Tipos de Sockets

Existen 4 tipos de sockets disponibles para el usuario:
***Stream sockets*:** proveen un flujo de datos bidireccional, seguro, secuenciado, no formateados y no-duplicado.
***Sockets Datagrama*:** proveen un flujo de datos bidireccional sin *asegurar* secuencialidad, seguridad, secuencialidad y no-duplicidad.
Raw sockets: proveen a los usuarios accesos a los protocolos subyacentes. está orientado a Datagramas y es para ser utilizado por desarrolladores de nuevos protocolos.
Secuencia de Paquetes de Socket: similares a los stream sockets con la salvedad que son formateados en paquetes.

Uso de los sockets

Creación de un sockets

Para crear un socket se utiliza la llamada al sistema **socket**.

```
s = socket( dominio, tipo, protocolo )
```

donde dominio es AF_UNIX, IF_INET ó AF_NS , tipo es SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, ó SOCK_SEQPACKET. Generalmente el valor del protocolo es 0 lo cual indica utilizar el protocolo por default.

Ligando un nombre local

Un socket es creado sin nombre inicialmente. Hasta que un nombre no es asociado al socket, los procesos no pueden referenciarlo y por lo tanto no se pueden recibir mensajes en el. Los procesos comunicados están ligados por una asociación, en el dominio INET y NS, la asociación está dada por *direccion local + port* y *dirección remota + port*, y esta asociación deberá ser única; mientras que en el dominio UNIX la asociación está dada por *nombre de archivo local* y *nombre de archivo remoto* y esta deberá ser única.

La llamada al sistema **bind** se utiliza para realizar *media* tarea de asociación (dirección local + port local ó nombre de archivo local) mientras que las primitivas **connect** y **accept** se utilizan para terminar la asociación en el otro lado.

La llamada dependiendo del tipo de dominio es similar a

```
...
struct sockaddr_{in,ns,un} sin;
...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Establecimiento de la conexión

El establecimiento de la conexión es generalmente asimétrico, con un proceso "cliente" y otro proceso "server". El server cuando ofrece un servicio, liga un socket a un port conocido asociado al servicio y pasivamente escucha ("listens") sobre ese socket. De esta manera es posible establecer un mecanismo de rendezvous con el server. En el lado del cliente, el llamado a **connect** es usado para iniciar la conexión.

```
struct sockaddr_{in,ns,un} server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

Para que el server reciba la conexión del cliente, deberá realizar dos pasos previos antes de ligar su socket. El primero es indicar su disposición a escuchar requerimientos de conexiones entrantes:

```
listen(s, 5);
```

El segundo parámetro especifica la cantidad máxima de conexiones que puede haber encoladas esperando ser atendidas por el proceso server.

Luego de indicar al socket como "escuchando", un proceso servidor puede aceptar una conexión:

```
struct sockaddr_{in,ns,un} from;
...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

Un nuevo socket es retornado, también es posible conocer la identidad del proceso cliente si se le especifican el 2° y 3° parámetro.

Accept normalmente bloquea al proceso que lo invoca, retorna cuando se estableció una conexión o se levantó una excepción.

Transferencias de Datos

Luego de establecida la conexión, los datos pueden comenzar a circular. Para enviar o recibir datos hay varias posibilidades tales como las primitivas de I/O:

```
write( s , buf , sizeof(buf));
read( s , buf , sizeof(buf));
```

o las nuevas primitivas

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

Estas primitivas son similares a las anteriores solo que tienen un flag opcional donde se puede especificar

MSG_OOB	enviar/recibir datos fuera de banda
MSG_PEEK	verificar sin leer
MSG_DONTROUTE	enviar datos sin rutear

Descartando los sockets

Una vez que el socket fué utilizado, se deberá cerrar con la primitiva **close**:

```
close( s )
```

Si hay datos en el socket, el sistema intentará enviar los datos, si luego de un tiempo no puede terminar, los datos serán descartados.
Si el usuario lo desea, puede cerrar una parte del socket

```
shutdown( s , que )
```

donde que == 0 indica el usuario no quiere leer más del socket, que = 1 indica que el usuario no quiere escribir más en el socket y que = 2 indica que se cancela tanto el envío como la recepción de datos.

H6 El Internet-Super-Server inetd

Frecuentemente, los servicios son realizados por los llamados deamons. Un deamon es un programa que abre un cierto port y espera por la llegada de conecciones. Si una ocurre, este crea un proceso que acepta la conección mientras que el padre continúa escuchando a la espera de nuevos requerimientos. Este concepto tiene la desventaja de que por cada servicio ofrecido se necesita un deamon corriendo que escuche en un determinado por a la espera de una conección, lo que implica generalmente un desperdicio de los recursos del sistema como por ejemplo espacio de swap.

Por lo tanto, casi todas las instalaciones de UNIX corren un "super-server" que crea sockets para un conjunto de servicios y espera en todos ellos simultáneamente usando la llamada al sistema select. Cuando un host remoto requiere uno de estos servicios, el super-server nota esto y dispara el servidor especificado para dicho port.

El super-server comunmente usado es el inetd (Internet Deamon). Es inicializado en tiempo de booteo, y toma la lista de servicios que va a manejar de un archivo de configuración generalmente llamado /etc/inetd.conf. Además de los servidores invocados, hay un número de servicios triviales que son servidos directamente por el inetd, llamados servicios internos. Ellos incluyen por ejemplo chargen el cual simplemente genera un string de caracteres, y daytime el cual retorna la idea que tiene el sistema de la hora del día.

Una entrada en el archivo de configuración consiste en una línea simple compuesta por los siguientes campos:

service type protocol wait user server cmdline

El significado de cada campo es el siguiente:

service da el nombre del servicio. El nombre del servicio debe ser trasladado a un número de port mirando en el archivo /etc/services.

type especifica un tipo de socket, ya sea stream (orientado a conecciones) o dgram (para protocolos datagrama). Los servicios basados en TCP deberán siempre usar stream, mientras que los basados UDP deberán siempre usar dgram.

protocol nombra el protocolo de transporte utilizado por el servicio. Este deberá ser un nombre de protocolo válido declarado en el archivo /etc/protocols.

wait esta opción se aplica sólo a sockets dgram. Puede ser wait o nowait. Si se especifica wait, el inetd sólo ejecutará un server a la vez para el port especificado. De otra manera, el inmediatamente continuará escuchado en el port luego de la ejecución del server.

Esto es útil para servers "single-threaded" que leen todos los datagramas hasta que no lleguen mas, y luego terminan. Muchos de los servidores RPC son de este tipo y por lo tanto deberán especificar wait. El tipo opuesto, servers "multi-threaded", permiten que un número ilimitado de instancias corran simultáneamente; estos son raramente usados. Este tipo de servers deberán especificar nowait.

Los sockets stream deberán siempre especificar nowait.

user Esta es la identidad de login del usuario bajo el cual el proceso es ejecutado. Esto generalmente será root, pero algunos servicios pueden requerir el uso de diferentes cuentas.

Es una muy buena idea aplicar el principio de privilegios mínimos, el cual dice que no se debería correr un comando bajo una cuenta privilegiada si el programa no lo requiere para su correcto funcionamiento. Por ejemplo, el servidor de news NNTP correrá como news, mientras que aquellos

servicios que impliquen un riesgo de seguridad (tales como tftp o finger) son generalmente ejecutados como nobody.

server da el camino absoluto del ejecutable correspondiente al servidor. Los servicios internos son marcados con la palabra clave internal.

cmdline Esta es la línea de comandos a ser pasada al servidor. Esta incluye incluye el argumento 0, esto es, el nombre del comando. Este campo se deja vacío para los servicios internos.

```
# See "man 8 inetd" for more information.
#
# If you make changes to this file, either reboot your machine or send the
# inetd a HUP signal:
# Do a "ps x" as root and look up the pid of inetd. Then do a
# "kill -HUP <pid of inetd>".
# The inetd will re-read this file whenever it gets that signal.
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
echo    stream tcp     nowait  root    internal
echo    dgram  udp     wait    root    internal
discard stream tcp     nowait  root    internal
discard dgram  udp     wait    root    internal
daytime         stream tcp     nowait  root    internal
daytime         dgram  udp     wait    root    internal
chargen         stream tcp     nowait  root    internal
chargen         dgram  udp     wait    root    internal
time    stream tcp     nowait  root    internal
time    dgram  udp     wait    root    internal
#
# These are standard services.
#
ftp     stream tcp     nowait  root    /usr/sbin/tcpd  wu.ftpd
telnet  stream tcp     nowait  root    /usr/sbin/tcpd  in.telnetd
#
# Use this one instead if you want to snoop on telnet users (try to use this
# for ethical purposes, ok folks?) :
# telnet stream tcp     nowait  root    /usr/sbin/tcpd  /usr/sbin/in.telnetsnoopd
#
# If you want to read NNTP news via TERM, comment out the nntp
# line below, and use a command like this once the TERM
# connection is up:  tredir 119 my.nntp.host:119
# You'll also want to do this: set NNTPSERVER my.nntp.host ; export NNTPSERVER
nntp    stream tcp     nowait  root    /usr/sbin/tcpd  in.nntpd
#
# This is for BSD sendmail:
# smtp  stream tcp     nowait  root    /usr/sbin/tcpd  sendmail -v
# This is set up for running Smail:
# smtp  stream tcp     nowait  root    /usr/sbin/tcpd  /usr/bin/rsmtp  -bs
#
# The comsat daemon notifies the user of new mail when biff is set to y:
comsat      dgram  udp     wait    root    /usr/sbin/tcpd  in.comsat
#
# Shell, login, exec and talk are BSD protocols.
#
```

```
shell    stream tcp     nowait root    /usr/sbin/tcpd  in.rshd -L
login    stream tcp     nowait root    /usr/sbin/tcpd  in.rlogind
# exec   stream tcp     nowait root    /usr/sbin/tcpd  in.rexecd
# talk   dgram  udp     wait   root    /usr/sbin/tcpd  in.talkd
ntalk    dgram  udp     wait   root    /usr/sbin/tcpd  in.talkd
#
# Kerberos authenticated services
#
# klogin stream tcp     nowait root    /usr/sbin/tcpd  rlogind -k
# eklogin        stream tcp     nowait root    /usr/sbin/tcpd  rlogind -k -x
# kshell stream tcp     nowait root    /usr/sbin/tcpd  rshd -k
#
# Services run ONLY on the Kerberos server
#
# krbupdate      stream tcp     nowait root    /usr/sbin/tcpd  registerd
# kpasswd        stream tcp     nowait root    /usr/sbin/tcpd  kpasswdd
#
# Pop et al
#
# pop2   stream tcp     nowait root    /usr/sbin/tcpd  in.pop2d
pop3     stream tcp     nowait root    /usr/sbin/tcpd  in.pop3d
#
# The Internet UUCP service.
#
# uucp   stream tcp     nowait uucp    /usr/sbin/tcpd  /usr/lib/uucp/uucico      -l
#
# Tftp service is provided primarily for booting.  Most sites
# run this only on machines acting as "boot servers."
#
# tftp   dgram  udp     wait   nobody /usr/sbin/tcpd  in.tftpd
# bootps         dgram  udp     wait   root    /usr/sbin/in.bootpd     in.bootpd
#
# Finger, systat and netstat give out user information which may be
# valuable to potential "system crackers."  Many sites choose to disable
# some or all of these services to improve security.
# Try "telnet localhost systat" and "telnet localhost netstat" to see that
# information yourself!
#
finger   stream tcp     nowait nobody /usr/sbin/tcpd  in.fingerd -w
systat   stream tcp     nowait nobody /usr/sbin/tcpd  /bin/ps  -auwwx
netstat  stream tcp     nowait root    /usr/sbin/tcpd  /bin/netstat      -a
gzip     stream tcp     nowait root    /bin/gzip       gzip    -f
gunzip   stream tcp     nowait root    /bin/gunzip     gunzip  -f
compress       stream tcp     nowait root    /bin/compress   compress -c
uncompress     stream tcp     nowait root    /usr/bin/uncompress     uncompress
zip      stream tcp     nowait root    /usr/bin/zip    zip
unzip    stream tcp     nowait root    /usr/bin/unzip  unzip
bwt      stream tcp     nowait root    /home/fredy/bwt/bin/wrapper     1
/home/fredy/bwt/bin/bwtscript
unbwt    stream tcp     nowait root    /home/fredy/bwt/bin/wrapper     1
/home/fredy/bwt/bin/unbwtscript
#
# Ident service is used for net authentication
auth     stream tcp     nowait root    /usr/sbin/in.identd     in.identd
#
```

```
# These are to start Samba, an smb server that can export filesystems to
# Pathworks, Lanmanager for DOS, Windows for Workgroups, Windows95, Lanmanager
# for Windows, Lanmanager for OS/2, Windows NT, etc.  Lanmanager for dos is
# available via ftp from ftp.microsoft.com in bussys/MSclient/dos/. Please read
# the licensing stuff before downloading. Use the TCP/IP option in the client.
# Add your server to the \etc\lmhosts (or equivalent) file on the client.
netbios-ssn     stream  tcp     nowait  root    /usr/sbin/smbd  smbd
netbios-ns      dgram   udp     wait    root    /usr/sbin/nmbd  nmbd
#
# Sun-RPC based services.
# <service name/version><sock_type><rpc/prot><flags><user><server><args>
#
# rstatd/1-3    dgram   rpc/udp wait    root    /usr/sbin/tcpd  rpc.rstatd
# rusersd/2-3   dgram   rpc/udp wait    root    /usr/sbin/tcpd  rpc.rusersd
# walld/1               dgram   rpc/udp wait    root    /usr/sbin/tcpd  rpc.rwalld
#
# End of inetd.conf.
```

Un inetd.conf ejemplo es mostrado arriba. Si no se quiere proveer un determinado servicio, por ejemplo por razones de seguridad, se puede anular la línea correspondiente a ese servicio ya sea borrándola o poniendo un numeral (#) de manera que esta sea tratada como un comentario.

Los archivos services y protocols

Los números de port en los cuales ciertos servicios "estandar" son ofrecidos están definidos en la RFC "Números Asignados". Para permitir a los programas clientes y servidores convertir un nombre de servicio a estos números de port, al menos una parte de esta lista es mantenida en cada host; está almacenada en un archivo llamado /etc/services. Una entrada tiene el siguiente formato:

service port/protocol [aliases]

Aquí, **service** especifica el nombre del servicio, **port** define el número de port en el cual el servicio es ofrecido y **protocol** define que protocolo de transporte es usado. Comunmente este es udp o tcp. Es posible para un servicio ser ofrecido para más de un protocolo, así como también ofrecer diferentes servicios en el mismo port, siempre y cuando los protocolos sean diferentes. El campo **aliases** permite especificar nombres alternativos para el mismo servicio.

Usualmente, usted no necesita cambiar el archivo services que viene con el software de red. De cualquier manera abajo damos un ejemplo de este archivo.

```
# Network services, Internet style
#
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, most entries here have two entries
# even if the protocol doesn't support UDP operations.
# Updated from RFC 1340, ``Assigned Numbers'' (July 1992).  Not all ports
# are included, only the more common ones.
#
#       from: @(#)services      5.8 (Berkeley) 5/9/91
#       $Id: services,v 1.9 1993/11/08 19:49:15 cgd Exp $
#
tcpmux          1/tcp           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
```

```
discard         9/udp           sink null
systat          11/tcp          users
daytime                 13/tcp
daytime                 13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp          # message send protocol
msp             18/udp          # message send protocol
chargen                 19/tcp          ttytst source
chargen                 19/udp          ttytst source
ftp             21/tcp
# 22 - unassigned
telnet          23/tcp
# 24 - private
smtp            25/tcp          mail
# 26 - unassigned
time            37/tcp          timserver
time            37/udp          timserver
rlp             39/udp          resource        # resource location
nameserver      42/tcp          name            # IEN 116
whois           43/tcp          nicname
domain          53/tcp          nameserver      # name-domain server
domain          53/udp          nameserver
mtp             57/tcp                          # deprecated
bootps          67/tcp          # BOOTP server
bootps          67/udp
bootpc          68/tcp          # BOOTP client
bootpc          68/udp
tftp            69/udp
gopher          70/tcp          # Internet Gopher
gopher          70/udp
rje             77/tcp          netrjs
finger          79/tcp
www             80/tcp          http    # WorldWideWeb HTTP
www             80/udp                  # HyperText Transfer Protocol
link            87/tcp          ttylink
kerberos        88/tcp          krb5    # Kerberos v5
kerberos        88/udp
supdup          95/tcp
# 100 - reserved
hostnames       101/tcp         hostname        # usually from sri-nic
iso-tsap102/tcp         tsap            # part of ISODE.
csnet-ns        105/tcp         cso-ns  # also used by CSO name server
csnet-ns        105/udp                 cso-ns
rtelnet         107/tcp         # Remote Telnet
rtelnet         107/udp
pop2            109/tcp         postoffice      # POP version 2
pop2            109/udp
pop3            110/tcp         # POP version 3
pop3            110/udp
sunrpc          111/tcp
sunrpc          111/udp
auth            113/tcp         tap ident authentication
sftp            115/tcp
uucp-path       117/tcp
```

```
nntp            119/tcp       readnews untp  # USENET News Transfer Protocol
ntp             123/tcp
ntp             123/udp                      # Network Time Protocol
netbios-ns      137/tcp                 # NETBIOS Name Service
netbios-ns      137/udp
netbios-dgm     138/tcp                 # NETBIOS Datagram Service
netbios-dgm     138/udp
netbios-ssn     139/tcp                 # NETBIOS session service
netbios-ssn     139/udp
imap2           143/tcp                 # Interim Mail Access Proto v2
imap2           143/udp
snmp            161/udp                      # Simple Net Mgmt Proto
snmp-trap       162/udp         snmptrap     # Traps for SNMP
cmip-man        163/tcp                 # ISO mgmt over IP (CMOT)
cmip-man        163/udp
cmip-agent      164/tcp
cmip-agent      164/udp
xdmcp           177/tcp                 # X Display Mgr. Control Proto
xdmcp           177/udp
nextstep        178/tcp       NeXTStep NextStep     # NeXTStep window
nextstep        178/udp         NeXTStep NextStep     # server
bgp             179/tcp                 # Border Gateway Proto.
bgp             179/udp
prospero        191/tcp                 # Cliff Neuman's Prospero
prospero        191/udp
irc             194/tcp                 # Internet Relay Chat
irc             194/udp
smux            199/tcp                 # SNMP Unix Multiplexer
smux            199/udp
at-rtmp         201/tcp                 # AppleTalk routing
at-rtmp         201/udp
at-nbp          202/tcp                 # AppleTalk name binding
at-nbp          202/udp
at-echo         204/tcp                 # AppleTalk echo
at-echo         204/udp
at-zis          206/tcp                 # AppleTalk zone information
at-zis          206/udp
z3950           210/tcp       wais            # NISO Z39.50 database
z3950           210/udp         wais
ipx             213/tcp                 # IPX
ipx             213/udp
imap3           220/tcp                 # Interactive Mail Access
imap3           220/udp                      # Protocol v3
ulistserv       372/tcp                 # UNIX Listserv
ulistserv       372/udp
#
# UNIX specific services
#
exec            512/tcp
biff            512/udp         comsat
login           513/tcp
who             513/udp         whod
shell           514/tcp       cmd             # no passwords used
syslog          514/udp
printer         515/tcp       spooler         # line printer spooler
```

```
talk            517/udp
ntalk           518/udp
route           520/udp                 router routed   # RIP
timed           525/udp                 timeserver
tempo           526/tcp         newdate
courier         530/tcp         rpc
conference      531/tcp         chat
netnews                 532/tcp         readnews
netwall         533/udp                                 # -for emergency broadcasts
uucp            540/tcp         uucpd           # uucp daemon
remotefs        556/tcp         rfs_server rfs  # Brunhoff remote filesystem
klogin          543/tcp                         # Kerberized `rlogin' (v5)
kshell          544/tcp                         # Kerberized `rsh' (v5)
kerberos-adm    749/tcp                         # Kerberos `kadmin' (v5)
#
webster         765/tcp                         # Network dictionary
webster         765/udp
#
# From ``Assigned Numbers'':
#
#> The Registered Ports are not controlled by the IANA and on most systems
#> can be used by ordinary user processes or programs executed by ordinary
#> users.
#
#> Ports are used in the TCP [45,106] to name the ends of logical
#> connections which carry long term conversations.  For the purpose of
#> providing services to unknown callers, a service contact port is
#> defined.  This list specifies the port used by the server process as its
#> contact port.  While the IANA can not control uses of these ports it
#> does register or list uses of these ports as a convienence to the
#> community.
#
ingreslock      1524/tcp
ingreslock      1524/udp
prospero-np     1525/tcp                # Prospero non-privileged
prospero-np     1525/udp
rfe             5002/tcp                # Radio Free Ethernet
rfe             5002/udp                # Actually uses UDP only
#
#
# Kerberos (Project Athena/MIT) services
# Note that these are for Kerberos v4, and are unofficial.  Sites running
# v4 should uncomment these and comment out the v5 entries above.
#
#kerberos       750/udp                 kdc     # Kerberos (server) udp
#kerberos       750/tcp         kdc     # Kerberos (server) tcp
krbupdate       760/tcp         kreg    # Kerberos registration
kpasswd                 761/tcp         kpwd    # Kerberos "passwd"
#klogin         543/tcp                 # Kerberos rlogin
eklogin         2105/tcp                # Kerberos encrypted rlogin
#kshell         544/tcp         krcmd   # Kerberos remote shell
gzip            10001/tcp       gzip
gunzip          10002/tcp       gunzip
compress        10003/tcp       compress
uncompress      10004/tcp       uncompress
```

```
zip             10005/tcp       zip
unzip           10006/tcp       unzip
bwt             10007/tcp
unbwt           10008/tcp
#
# Unofficial but necessary (for NetBSD) services
#
supfilesrv      871/tcp                 # SUP server
supfiledbg      1127/tcp                # SUP debugging
```

De manera similar al archivo services, la librería de red necesita trasladar nombres de protocolo a números de protocolo. Esto es hecho mirando en el archivo /etc/protocols. Este tiene una entrada por línea, cada una conteniendo el nombre del protocolo y el número asociado. Tener que tocar este archivo es aun mas raro que tener que tocar el archivo /etc/services. Un ejemplo de este archivo se da a continuación.

```
#
# protocols     This file describes the various protocols that are
#               available from the TCP/IP subsystem.  It should be
#               consulted instead of using the numbers in the ARPA
#               include files, or, worse, just guessing them.
#

ip      0       IP      # internet protocol, pseudo protocol number
icmp    1       ICMP    # internet control message protocol
igmp    2       IGMP    # internet group multicast protocol
ggp     3       GGP     # gateway-gateway protocol
tcp     6       TCP     # transmission control protocol
pup     12      PUP     # PARC universal packet protocol
udp     17      UDP     # user datagram protocol
idp     22      IDP     # WhatsThis?
raw     255     RAW     # RAW IP interface
# End.
```

H7 PVM (Parallel Virtual Machine)

Introducción

PVM es un sistema de software que permite que una colección de computadoras potencialmente heterogéneas concectadas a través de distintas clases de redes sean usadas como un único recurso computacional concurrente.

Los programas de usuario escritos en C, C++ o Fortran acceden a PVM a través de rutinas de librería.

El método de generación de los ejecutables es el usual (deben incluirse las librerías y sus headers).

Los deamons (pvmd) proveen comunicación y control de procesos entre las computadoras que conforman la máquina virtual.

pvmd3 - PVM version 3 daemon

pvmd3 es un proceso deamon que coordina los hosts unix que conforman una máquina virtual. Un pvmd3 debe correr en cada host. Ellos proveen las funciones de comunicación y de control de procesos requeridas por los procesos PVM de los usuarios.

Este deamon puede ser arrancado manualmente. Se le puede pasar como parámetro un hostfile para que corra automáticamente los pvmds remotos. Los pvmds remotos y el local pueden también ser arrancados desde el programa consola PVM, pvm, o desde la consola gráfica XPVM. Ambas consolas serán descriptas más adelante.

El nombre del ejecutable correspondiente al deamon es pvmd3. El es usualmente arrancado por el script envolvente del shell, $PVM_ROOT/lib/pvmd.

Para bajar los pvmds el método preferido o aconsejado es utilizar el comando halt dentro de las consolas. Esto mata todas las tareas PVM, todos los deamons remotos, el deamon local, y, en el caso de la consola pvm, la consola misma.

Si el master pvmd (el local, el primero en iniciarse, el que levantó a los demás) es matado manualmente debe enviársele una señal SIGTERM para permitirle matar a los pvmds remotos y eliminar varios archivos utilizados internamente por él (clean up).

El pvmd puede ser matado de manera tal que deje el archivo /tmp/pvmd.uid en uno o más hosts de la máquina virtual. uid es el user ID (extraído del archivo /etc/passwd) del usuario. Esto evitará que el PVM pueda relevantarse en dichos hosts (son archivos de lockeo). El borrado de estos archivos solucionará el problema.

Cada host en la máquina virtual debe tener una entrada en el host file o debe ser agregado a mano con el comando add. Las líneas que comienzan con un número (#) son ignoradas. El siguiente es un host file simple:

```
# mi primer host file
lidi
delphi
tdg1
tdg2
```

Este especifica los nombres de cuatro hosts a ser configurados dentro de la máquina virtual.

El master pvmd para un grupo es arrancado a mano o mediante una consola en localhost, y el arranca esclavos en todos los hosts restantes utilizando comandos tales como el rsh (es configurable). El host master puede aparecer en cualquiera de las líneas del hostfile, pero debe aparecer.

El formato simple de arriba trabaja bien si usted tiene el mismo login name en las cuatro máquinas y el nombre del master host en el archivo $HOME/.rhosts en las otras tres.

Si este no es el caso, existen varias opciones disponibles que permiten, entre otras cosas, cambiar el login name, especificar el path del ejecutable correspondiente al pvmd, especificar el path donde el pvmd debe buscar las tareas a disparar o especificar el directorio de trabajo. También permite el uso de variables de ambiente que son expandidas en cada host. Cada opción tiene un default. El formato del host file permite cambiar los defaults asociados a las diferentes opciones dinámicamente.

Las siguientes variables de ambiente son usadas por PVM y algunas pueden ser utilizadas para personalizar su ambiente. El método usual es setearlas y exportarlas en el archivo $HOME/.profile o en el archivo $HOME/.cshrc según el shell que usted use.

PVM_ROOT

Indica el path a partir del cual las librerías PVM y los programas del sistema están instalados, por ejemplo /usr/local/pvm3 (ubicación usual para instalación pública) o $HOME/pvm3 (ubicación usual para instalación privada). Esta variable debe ser seteada en cada host en el que el PVM sea usado para que el PVM funcione. No tiene un valor por default.

PVM_EXPORT

Contiene los nombres de las variables de ambiente a exportar desde una tarea padre a tareas hijas a través del pvm_spawn().

Múltiples nombres deben ser separados por ':'. Si esta variable no está seteada no se exporta ninguna variable del ambiente del padre.

PVM_DEBUGGER

El debugger script a utilizar cuando pvm_spawn() es llamado con PvmTaskDebug seteado. El valor por default es $PVM_ROOT/lib/debugger.

PVM_DPATH

[v3.3.2 y posteriores] El path del startup script asociado al pvmd (el default es $PVM_ROOT/lib/pvmd). Es sobreescrita por la opción del host file dx=.

Las siguientes variables de ambiente son usadas por el PVM internamente. Con la excepción de PVM_ARCH, sus valores no deben ser modificados. Las listamos aquí sólo a título informativo.

PVM_ARCH

El nombre de arquitectura PVM del host en el cual está, usada para distinguir entre máquinas con diferente formato de ejecutables (a.out). Copias de un programa para las diferentes arquitecturas son instaladas en directorios paralelos nombrados por la arquitectura PVM. Usualmente $PVM_ROOT/bin/$PVM_ARCH.

PVMSOCK

Es pasada por el pvmd a las tareas disparadas, y da la dirección del socket local del pvmd.

PVMEPID

Contiene el process id esperado de una tarea disparada por el pvmd. La tarea utiliza este valor para identificarse ella misma cuando se reconecta al pvmd.

PVMTMASK

[v3.3 y superiores] La libpvm trace mask, pasada por el pvmd a las tareas disparadas.

pvm (consola/monitor)

pvm es una tarea PVM "independiente" que permite al usuario encuestar y modificar interactivamente la máquina virtual. La consola puede ser arrancada y parada muchas veces en cualquiera de los hosts en la máquina virtual sin afectar a PVM ni a las aplicaciones que pudieran estar ejecutándose.

Cuando es arrancada pvm determina si PVM esta corriendo actualmente y si no automáticamente ejecuta pvmd3 en ese host, pasándole a pvmd3 las opciones en la línea de comandos y el host file. Por esto no es necesario que PVM esté corriendo para arrancar la consola. Una vez arrancada la consola da el siguiente prompt:

pvm>_

Los siguientes comandos estan disponibles en la consola:

```
add hostname(s)      - add hosts to virtual machine
alias                - define/list command aliases
conf                 - list virtual machine configuration
delete  hostname(s)  - delete hosts from virtual machine
echo                 - echo arguments
halt                 - stop pvmds
help [command]       - print helpful information about a command
id                   - print console task id
jobs                 - list running jobs
kill task-tid        - terminate tasks
mstat host-tid       - show status of hosts
ps -a                - list all PVM tasks
pstat task-tid       - show status of tasks
quit                 - exit console
reset                - kill all tasks
setenv               - display/set environment variables
sig signum task      - send signal to task
spawn [opt] a.out    - spawn task
        opts are:
                -(count) number of tasks, default is 1
                -(host)  spawn on host, default is any
                -(ARCH)spawn on hosts of ARCH
                -?       enable debugging
                ->       redirect task output to console
                -> file  redirect task output to file
                ->>file  redirect task output append to file
trace                - set/display trace event mask
unalias              - undefine command alias
version              - show libpvm version
```

pvm lee $HOME/.pvmrc antes de leer comandos de la tty, por lo tanto este archivo puede ser usado para customizar (personalizar) el ambiente de la consola, por ejemplo:

```
alias ? help
alias j jobs
setenv PVM_EXPORT DISPLAY
# print my id
echo new pvm shell
id
```

Para otro ejemplo de console startup script .pvmrc ver $PVM_ROOT/doc/example.pvmrc

EJEMPLOS

pvm

Levanta pvmd3 en el host local si no está corriendo. Si está corriendo se conecta a él.

pvm hostfile

Si el pvmd3 está corriendo ignora el host file y se conecta al pvmd3.

Si no, levanta la consola y el pvmd3, el cual lee el host file hostfile y agrega los hosts en él listados a la máquina virtual (probablemente arrancando los deamons remotos [si qiuere profundizar en el *probablemente* lea la página del man que describe en detalle el formato del host file {man pvmd3}]).

XPVM (consola/monitor gráfico)

XPVM es una consola y/o monitor gráfico para PVM que corre sobre X windows. Provee una interface gráfica a los comandos e información de la consola de PVM (pvm), junto con diversas "vistas" animadas para monitorear la ejecución de programas PVM. Estas vistas proveen información acerca de las interacciones entre tareas en un programa PVM paralelo, para permitir la puesta a punto y el análisis de performance (entre otras cosas).

El análisis puede hacerse en tiempo real o en playback post-mortem a partir de archivos de traza salvados previamente.

Para consultas específicas sobre las distintas facilidades de XPVM, referirse a la información apropiada en el help en línea que provee la aplicación:

```
PVM Console Command Help:
-------------------------
* Hosts
* Tasks
        - Spawn
        - On-The-Fly
        - Kill
        - Signal
        - System Tasks
* Reset
* Quit
* Halt
```

XPVM Views & Trace Controls:
---
* Views
  - Network
  - Space-Time
  - Utilization
  - Message Queue
  - Call Trace
  - Task Output
* Traces

Questions or Problems:
---
* Author

NOTA

Esta consola, al igual que la anterior, permite especificar un host file. El archivo que utiliza como host file es el archivo $HOME/.xpvm_hosts. El formato de este archivo no es completamente compatible con el formato del host file descripto anteriormente lo que puede generar algunos problemas.

libpvm

libpvm3.a, libfpvm3.a - PVM C and Fortran programming libraries

Todas las aplicaciones deben ser linkeadas con la librería libpvm para poder comunicarse con otras entidades en el sistema PVM. La librería básica (libpvm3.a), escrita en C, directamente soporta aplicaciones C y C++. La librería Fortran (libfpvm3.a) está compuesta de funciones "envolventes" para convertir llamadas Fortran a llamadas C.

Resumiendo, las aplicaciones escritas en C deben ser linkeadas con la librería básica de PVM. Las aplicaciones Fortran deben ser linkeadas con ambas librerías.

En algunos sistemas operativos, los programas PVM deben ser linkeados con otras librerías (conteniendo, por ejemplo, funciones para el manejo de sockets).

Por último, programas que usan funciones de manejo de grupos deben también ser linkeados con libgpvm3.a.

Las subrutinas de la librería libpvm pueden ser divididas en cinco clases:

Pasaje de mensajes

pvm_bufinfo, pvm_freebuf, pvm_getrbuf, pvm_getsbuf,
pvm_initsend, pvm_mcast, pvm_mkbuf, pvm_nrecv,
pvm_pack, pvm_precv, pvm_probe, pvm_psend,
pvm_recv, pvm_recvf, pvm_send, pvm_sendsig,
pvm_setmwid, pvm_setrbuf, pvm_setsbuf, pvm_trecv,
pvm_unpack

Control de tareas

pvm_exit, pvm_kill, pvm_mytid, pvm_parent,
pvm_pstat, pvm_spawn, pvm_tasks

Funciones de la librería de grupos

pvm_barrier, pvm_bcast, pvm_gather, pvm_getinst,
pvm_gettid, pvm_gsize, pvm_joingroup, pvm_lvgroup,
pvm_reduce, pvm_scatter

Control de la máquina virtual

pvm_addhosts, pvm_config, pvm_delhosts, pvm_halt,
pvm_mstat, pvm_reg_hoster, pvm_reg_rm,
pvm_reg_tasker, pvm_start_pvmd

Misceláneos

pvm_archcode, pvm_catchout, pvm_getopt,
pvm_hostsync, pvm_notify, pvm_perror, pvm_setopt,
pvm_settmask, pvm_tidtohost

En llamadas exitosas, la mayoría de las funciones de la librería libpvm retornan la constante PvmOk. En caso de error se retornan otros valores. Por ejemplo, el valor de retorno PvmNoHost indica que no existe un host en la máquina virtual con el nombre dado, o que el nombre no puede ser resuelto en una dirección de IP (problemas con el resolver).

Para ver exhaustivamente la lista de los valores de error ver la página del man asociada a libpvm (man libpvm).

La distribución de los archivos de la librería es la siguiente:

$PVM_ROOT/include/fpvm3.h
Fortran header file

$PVM_ROOT/include/pvm3.h
C header file

$PVM_ROOT/include/pvmsdpro.h
Header file for tasker, hoster and resource manager tasks

$PVM_ROOT/include/pvmtev.h
Header file for tasks manipulating trace events

$PVM_ROOT/lib/$PVM_ARCH/libpvm3.a
C (base) library

$PVM_ROOT/lib/$PVM_ARCH/libfpvm3.a
Fortran wrapper library

$PVM_ROOT/lib/$PVM_ARCH/libgpvm3.a
Group function library

NOTAS

* Para información específica acerca de una subrutina referirse a la página del manual asociada a ella.

* Existen subrutinas que no están incluidas en este resumen, una forma de ver cuales son es analizar el subdirectorio $PVM_ROOT/man/man3. Algunas de las que no están son sólo para compatibilidad con versiones anteriores de PVM.

aimk

aimk - Portable make wrapper script

aimk es un programa "envolvente" para el make, usado para seleccionar opciones portablemente para construir el PVM y aplicaciones PVM en varias máquinas. A cada port de PVM se le asigna un nombre de arquitectura. Este nombre es usado durante la compilación (por ejemplo para inclusión condicional) y al momento de la ejecución (para seleccionar un ejecutable o un host).

aimk usa el valor de la variable de ambiente $PVM_ARCH si esta está seteada, en otro caso llama a $PVM_ROOT/pvmgetarch para determinar el nombre de la arquitectura. pvmgetarch es un scrip que "hurga" en distintas partes del sistema para determinar el nombre de arquitectura correcto. Es actualizada a medida que se definen nuevos ports de PVM, y puede ser aumentada localmente.

Para una descripción mas detallada del funcionamiento de aimk ver su página del man.

EJEMPLOS

El siguiente archivo Makefile.aimk construye e instala hello, creando el directorio binario PVM si este no existe. Puede ser corrida concurrentemente sobre máquinas de diferentes tipos, compartiendo el mismo directorio fuente (por ejemplo, utilizando NFS).

```
LDIR   =      -L$(PVM_ROOT)/lib/$(PVM_ARCH)
PVMLIB=       -lpvm3
SDIR   =      ..
BDIR   =      $(HOME)/pvm3/bin
XDIR   =      $(BDIR)/$(PVM_ARCH)
CFLAGS=       -g -I$(PVM_ROOT)/include
LIBS   =      $(LDIR) $(PVMLIB) $(ARCHLIB)

$(XDIR):
        - mkdir $(BDIR) $(XDIR)

hello: $(SDIR)/hello.c $(XDIR)
        $(CC) $(CFLAGS) -o $@ $(SDIR)/$@.c $(LIBS)
        mv $@ $(XDIR)
```

Variables de entorno usadas por este script:

$PVM_ROOT	Root path of PVM installation.
$PVM_ARCH	PVM architecture name for machine.

Archivos relacionados con este script

$PVM_ROOT/lib/aimk	The aimk program
$PVM_ROOT/conf/$PVM_ARCH.def	Arch config file

Un ejemplo simple

```
/* hello.c */
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("can't start hello_other\n");

        pvm_exit();
        exit(0);
}

/* hello_other.c */
#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];

        ptid = pvm_parent();

        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);
```

```
        pvm_exit();
        exit(0);
}
```

H8 MPI - MESSAGE-PASSING INTERFASE

Introducción

El forum de MPI definió la sintaxis y la semántica de un núcleo estandar de rutinas de librería que son muy útiles para un amplio rango de usuarios y eficientemente implementables en una amplia gama de computadoras.

Los diseñadores del MPI trataron de tomar las características más interesantes de los sistemas de pasajes de mensajes existentes (incluído PVM).

El standard MPI define la interface de usuario y la funcionalidad de una amplia gama de rutinas de pasajes de mensajes.

Una de las características más importantes del MPI, como la de todos los standard, es el grado de portabilidad. Esto significa que el mismo código fuente que utilizamos para el pasaje de mensajes, puede ser utilizado en máquinas con distintas arquitecturas siempre y cuando se disponga de una librería de MPI.

La portabilidad es esencial, pero el standard sólo será usado si dicha portabilidad no es lograda a expensas de eficiencia. El MPI fue cuidadosamente diseñado para permitir implementaciones eficientes en las distintas plataformas.

Objetivos del MPI

Diseño de una interface de pasaje de mensajes para la programación de aplicaciones.

Permitir una comunicación eficiente.

Permitir implementaciones que puedan ser usadas en arquitecturas heterogéneas.

Posibilitar el linkeo a Fortran y a C.

Proveer un mecanismo de comunicación seguro.

Definir una interface no muy diferente a las existentes actualmente (Ej. PVM).

Definir una interface que pueda ser implementada en distintas plataformas y que su implementación no signifique cambios significativos al software preexistente.

La interface no deberá interferir con el manejo de procesos (thread-safety).

¿ Qué incluye el MPI ?

Comunicación Punto a Punto.

Operaciones Colectivas.

Manejo de grupos de procesos.

Dominios de comunicación.

Topología de procesos.

Manejo de ambiente y administración.

Binding con Fortran 77 y C.

¿ Qué no incluye el MPI ?

Operaciones de memoria compartida explícitas.

Operaciones muy dependientes del S.O (recepción por interrupción, ejecución remota, ...).

Herramientas para la construcción de programas (editores, make's, ...).

Facilidades para depurar.

Soporte específico para threads.

Soporte para manejo de tareas.

Funciones de I/O.

Comunicación punto a punto

El MPI provee un conjunto de funciones send y receive que permiten la comunicación de datos tipados con un tag asociado.

La información sobre tipos es necesaria para soportar la heterogeneidad de arquitecturas, el tag permite la selectividad de los mensajes.

El MPI soporta dos tipos de llamadas, bloqueantes y no bloqueantes. En el send bloqueante, el llamado retorna cuando el buffer puede ser sobreescrito. En el receive bloqueante, el llamado retorna cuando el buffer contiene el mensaje.

Los llamados no bloqueantes permiten solapar transmisión con computación o múltiples transmisiones entre sí y con computación; estas siempre vienen en dos partes, la inicialización de la operación y el testeo de finalización.

Según los distintos problemas y arquitecturas, puede ser mas apropiado un modo de operaciones send/receive sobre otros (Ej. asincrónico vs. sincrónico) . Como el objetivo principal del MPI es estandarizar las operaciones sin sacrificar performance, se decidió incluir varios modos de comunicación punto a punto. Los modos permiten elegir la semántica de la operación send/receive.

Modos de comunicación

ESTANDAR

La completación del send no implica necesariamente que el receive ha comenzado.

BUFFEREADO

Funciona de manera similar al Standard, pero el usuario puede garantizar que existe una cierta cantidad de espacio disponible de buffers.

SINCRONICO

Rendevouz entre el emisor y el receptor.

LISTO

Se efectúa la transmisión cuando el transmisor y el receptor están en condiciones (Similar a comunicación sincrónica con guardas).

Comunicaciones colectivas

Las comunicaciones colectivas transmiten datos entre procesos de un grupo.

El MPI provee las siguientes funciones de comunicación colectiva:

- Sincronización de barrera.
- Funciones de comunicación global.
  - Broadcast: 1 a todos.
  - Gather Data: Todos a 1.
  - Scatter Data: Datos de 1 a todos.
  - All Gather: Variación de gather donde todos reciben el resultado.
  - Scatter/Gather : Todos contra todos.
- Operaciones de reducción global.
  - Suma
  - Máximo
  - Mínimo
  - etc.

Descripción de la implementación de MPI MPICH

El objetivo del Forum MPI fué desarrollar un estandar para escribir programas que usen pasaje de mensajes que sea práctico, portable, eficiente, y flexible.

Diseñando el MPI el Forum MPI tomó las características más atractivas de varios sistemas de pasaje de mensajes.

El MPI es una especificación (como el C o el Fortran) y existen varias implementaciones. De aqui en mas se describen caracteristicas de la implementación de MPI llamada MPICH.

Notas

Para comenzar a trabajar con MPI hay que agregar el subdirectorio bin del MPI al path.

Uno puede compilar y linkear sus propios programas MPI con los comandos mpicc y mpif77

Documentación

En el directorio doc de la distribución hay varios documentos Postscript. Esto incluye una guía introductoria (guide.ps) y un manual de usuario (manual.ps).

Existen páginas man para cada subrutina MPI.

El comando mpiman arranca un xman para las páginas man de MPI.

Distribución de los archivos

/usr/local/mpi/	MPI software directory
/usr/local/mpi/COPYRIGHT	Copyright notice
/usr/local/mpi/README	various notes and instructions
/usr/local/mpi/bin/	binaries, including mpirun
/usr/local/mpi/examples/basic	elementary MPI programs
/usr/local/mpi/doc/	documentation
/usr/local/mpi/include/	include files
/usr/local/mpi/lib/	library files

Comunicación punto a punto

Operaciones bloqueantes

MPI_Send realiza un send bloqueante en modo estandar.

MPI_Recv realiza un receive bloqueante en modo estandar.

Operaciones no bloqueantes

MPI_ISend realiza un send no-bloqueante en modo estandar.

MPI_IRecv realiza un receive no-bloqueante en modo estandar.

Ambas llamadas comienzan la operación y retornan inmediatamente. Luego deben ir testeando la finalización de la operación. Una vez finalizado el send el buffer está libre para enviar otro mensaje. Una vez finalizado el receive el buffer contiene el mensaje recibido.

Operaciones de completación asociadas a las llamadas no bloqueantes

MPI_Wait Permite bloquearse hasta que la operación asociada se complete.

MPI_Test Permite testear si se completó la operación asociada.

Nota

El send bloqueante se completa cuando retorna la llamada, el no bloqueante cuando Wait o Test retornan exitosamente.

Modos de comunicación

STANDARD

La completación del send no necesariamente significa que el receive asociado ha comenzado.

El MPI decide cuando bufferear los mensajes.

Puede llegar a completarse un send aunque no exista aun un receive asociado.

BUFFERED

Permite garantizar una cierta cantidad de espacio de buffers. El espacio debe ser provisto explícitamente por la aplicación.

Fuera de lo anterior sus características son similares a las del modo standard.

SYNCHRONOUS

Rendevous entre el emisor y el receptor.

El send puede comenzar haya o no un receive asociado. Sin embargo, el send se completará exitosamente sólo si es comenzado un receive que machee con el. Por lo tanto, la completación de un send sincrónico no sólo indica que el buffer asociado puede ser reutilizado, también indica que el receptor llegó a algún punto determinado de su ejecución (al receive).

READY

Permite simplificar el protocolo subyacente y potencialmente ganar performance.

Un send en modo ready puede comenzar sólo si existe un receive pendiente.

En algunos sistemas esto permite evitar la operación de hand-shake y optimiza la performance.

Fuera de esto, su semántica es igual a la del modo estandar.

NOTAS

Para estos modos de comunicación adicionales se proveen tres funciones send.

MPI_Send => MPI_BSend, MPI_SSend y MPI_RSend
MPI_ISend => MPI_IBSend, MPI_ISSend y MPI_IRSend

Los receives son los usuales (MPI_Recv y MPI_IRecv)

Comunicadores

Entre otras cosas un comunicador sirve para definir un grupo de procesos que pueden comunicarse entre sí.

El comunicador por defecto es MPI_COMM_WORLD.

Los procesos pueden acceder y setear atributos de los comunicadores. Por ejemplo, para conocer su identidad dentro del comunicador un proceso puede utilizar la función MPI_Comm_rank.

Comunicaciones colectivas

* Sincronización de barrera entre todos los miembros de un grupo

* Funciones de comunicación global

** Broadcast
** Gather Data
** Scatter Data
** All Gather
** Scatter/Gather

* Operaciones de reducción global (Suma, Máximo, Mínimo, funciones definidas por el usuario, etc.

** Reducción
** Reducción y scatter
** Prefijo

mpirun

Descripción

"mpirun" es un shell script que fué hecho con la intención de facilitarle al usuario la tarea de comenzar trabajos en distintas arquitecturas/hosts.

Según el ambiente de trabajo, debe proveerse un archivo que liste las diferentes máquinas que mpirun puede usar para correr trabajos remotos. Este archivo puede especificarse cada vez que uno corre mpirun con la opcion -machinefile <machine-file name>. Si no se le pasa esta opción el archivo usado por default esta en util/machines.<arch>.

Notas

* El subdirectorio bin es un soft link al subdirectorio util.
* En LINUX el nombre del machinefile por default es machines.LINUX

mpirun típicamente se invoca de esta manera mpirun -np <número de procesos> <nombre de programa y argumentos>

Parámetros

Las opciones para mpirun deben ir antes del programa que se desea ejecutar

mpirun [mpirun_options...] <progname> [progname_options...]

Entre las cosas que se pueden realizar con estas opciones figuran:

Especificar la arquitectura a utilizar
Especificar un machinefile
Especificar el número de procesos a correr en cada máquina
Redireccionar la entrada, la salida y el error estandar
Seleccionar el modo de operación "verborrágico"
Forzar la comunicación con debuggers (gdb, xgdb, ...)

Múltiples arquitecturas pueden ser manejadas poniendo múltiples argumentos -arch y -np. Por ejemplo, para correr un programa en 2 sun4s y en 3 rs6000s, siendo la máquina local una sun4, usar

./mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program

Lo anterior asume que el programa corre en ambas arquitecturas. Si se necesitan diferentes ejecutables, el string '%a' será reemplazado con el nombre de la arquitectura. Por ejemplo, si los programas son program.sun4 y program.rs6000, entonces el comando es

./mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program.%a

Si en cambio los ejecutables estan en diferentes directorios; por ejemplo, /tmp/me/sun4 y /tmp/me/rs6000, entonces el comando es

./mpirun -arch sun4 -np 2 -arch rs6000 -np 3 /tmp/me/%a/program

Es importante especificar la arquitectura con -arch antes de especificar el número de procesadores. También, el primer -arch debe referirse al procesador en el cual el trabajo será arrancado.

Específicamente, si -nolocal no está especificado, entonces el primer -arch debe referirse al procesador en el cual mpirun esta corriendo.

Ejemplo

Código fuente:

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"

#define RUN

void main(int argc, char **argv)
{

        int myid, numprocs, count;
        int namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
```

```
        char buf[100];
        MPI_Status status;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        fprintf(stderr,"Process %d on %s\n", myid, processor_name);

#ifdef RUN

        if (myid == 0)
        {
/*              MPI_Recv(buf, 100, MPI_CHAR, 1, 1, MPI_COMM_WORLD, &status);*/
                if (MPI_Recv(buf, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status) != MPI_SUCCESS) {
                        fprintf(stderr, "Recibiendo el string\n");
                        exit(-1);
                }
                MPI_Get_count(&status, MPI_CHAR, &count);
                fprintf(stderr, "<%d><Recibio %d bytes de %d con tag %d>\n", myid, count,
status.MPI_SOURCE, status.MPI_TAG);
                printf("from t%x: %s\n", 1, buf);
        }
        else {
                strcpy(buf, "hello, world from ");
                if (gethostname(buf + strlen(buf), 64)) {
                        perror("gethostname");
                        exit(-1);
                };

                if (MPI_Send( buf, strlen(buf)+1, MPI_CHAR, 0, 1, MPI_COMM_WORLD ) !=
MPI_SUCCESS ) {
                        fprintf(stderr, "Enviando el string\n");
                        exit(-1);
                }
        }
#endif
        MPI_Finalize();
        exit(0);
}
```

Linea de comando:

```
mpirun -machinefile machine-file -np 2 hello
```

Salida:

```
Process 0 on lidi.
Process 1 on lidi.
```

```
<0><Recibio 22 bytes de 1 con tag 1>
from t1: hello, world from lidi
```

Contenido del machine-file:

```
lidi
```

Makefile:

```
# Generated automatically from Makefile.in by configure.
ALL: default
##### User configurable options #####

INCLUDE    = ../include
SHELL      = /bin/sh
ARCH       = LINUX
COMM       = ch_p4
MPIR_HOME  = /usr/local/mpich
CC         = /usr/local/mpich/lib/LINUX/ch_p4/mpicc
CLINKER    = $(CC)
F77        = /usr/local/mpich/lib/LINUX/ch_p4/mpif77
FLINKER    = $(F77)
CCC        = /usr/local/mpich/lib/LINUX/ch_p4/mpiCC
CCLINKER   = $(CCC)
AR         = ar crl
RANLIB     = ranlib
PROFILING  = $(PMPILIB)
OPTFLAGS   =  -Wall -O6
MPE_LIBS   = -L/usr/X11R6/lib -lmpe -lX11 -lm
MPE_DIR    = /usr/local/mpich/mpe
MPE_GRAPH  = -DMPE_GRAPHICS
PATHLIB    = ../lib
#

### End User configurable options ###

CFLAGS          = $(OPTFLAGS) -I$(INCLUDE) $(MPIR_HOME)/include
CFLAGSMPE       =  $(CFLAGS)  -  $(MPE_DIR)  $(MPE_GRAPH)  $(INCLUDE)
$(MPIR_HOME)/include
CCFLAGS         = $(CFLAGS)
#FFLAGS         = '-qdpc=e'
FFLAGS          = $(OPTFLAGS)
EXECS  = hello

default: $(EXECS)

all: default

hello:  hello.c
        $(CLINKER) $(OPTFLAGS) -o hello hello.c

clean:
        /bin/rm -f *.o *~ $(EXECS)
```

Publicaciones

4° Jornada de Investigación. Grupo Montevideo

Agosto de 1996
Universidad Nacional de Río Grande do Sul. Brasil
Publicación y Exposición: "Paralelización de un algoritmo de Compresión que utiliza Diccionario Estático".

Experiencia comparativa entre un algoritmo secuencial y otro distribuido en una red de computadoras sobre un sistema operativo Unix; ambos implementados en C y utilizando la interfaces de comunicación del s.o.(sockets).

2° Congreso Argentino de Ciencias de la Computación

Noviembre de 1996
Universidad Nacional de San Luis. San Luis
Publicación y Exposición: "Paralelización de un algoritmo de Compresión que utiliza Diccionario Estático".

Experiencia comparativa entre un algoritmo secuencial y otro distribuido en una red de computadoras sobre un sistema operativo Unix; ambos implementados en C y utilizando la interfaces de comunicación del s.o.(sockets).

2° Congreso Argentino de Ciencias de la Computación

Noviembre de 1996
Universidad Nacional de San Luis. San Luis
Publicación y Exposición: "Algoritmo Distribuido de Compresión de Datos - Experiencia comparativa con Sockets y PVM".

Experiencia comparativa entre el algoritmo anterior y otro implementado utilizando PVM(Paralell Virtual Machine), un conjunto de librerías que permite simular una gran máquina virtual compuesta por varias computadoras en red, sobre el s.o. Unix.

III Congreso Internacional de Ingeniería Informática. ICIE' 96-97

Abril 1997
Facultad de Ingeniería. Universidad de Buenos Aires
Publicación y Exposición: "Algoritmo Distribuido de Compresión de Datos. Una experiencia comparativa con Sockets y PVM".

Experiencia comparativa entre el algoritmo anterior y otro implementado utilizando PVM(Paralell Virtual Machine), un conjunto de librerías que permite simular una gran máquina virtual compuesta por varias computadoras en red, sobre el s.o. Unix.

Bibliografía

[Kern91]
El Lenguaje de Programación C
Brian W. Kernighan - Dennis M. Ritchie

[Bach86]
The Design of the Unix Operating System
Maurice J. Bach

[Hamm86]
Coding and Information Theory
Richard W. Hamming

[Nels91]
The Data Compression Book
Mark Nelson

[Kirc94]
The Linux Network Administrator's Guide
Olaf Kirch

[Wirz96]
The Linux System Administrator's Guide
Lars Wirzenius

[Gold95]
The Linux Programmer's Guide
Sven Goldt - Sven van der Meer - Scott Burkett - Matt Welsh

[John92]
The Linux Kernel Hacker's Guide
Michael K. Johnson

[Geis92]
PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing
Al Geist - Adam Beguelin - Jack Dongarra - Weicheng Jiang - Robert Manchek - Vaidy Sunderam

[Grop96]
MPI Installation Guide to mpich, a Portable Implementation of MPI
William Gropp - Ewing Lusk

[Lusk96]
MPI User's to mpich, a Portable Implementation of MPI
William Gropp - Ewing Lusk

[Come91]
Internetworking with TCP/IP. Volume I: Principles, Protocols, and Architecture
Douglas E. Comer

[Jain89]
Fundamentals of Digital Image Processing
Anil K. Jain

An Advanced 4.3BSD Interprocess Communication Tutorial
Samuel J. Leffler - Robert S. Fabry - William N. Joy - Phil Lapsley - Steve Miller - Chris Torek

Parallel Algorithms for Optimal Compression using Dictionaries with the Prefix Property
Sergio De Agostino - James A. Storer

[Andr91]
Concurrent Programming. Principles and Practice.
Gregory R. Andrews

Artículos

Variations on a Theme by Huffman
Robert A. Gallager
IEEE Transactions on Information Theorym, Vol IT-24, NO. 6
Noviembre 1978

Parallel Computing-Windows Style
Alexandre Alves
Byte
Mayo 1996

A Simple Data Compression Technique
Ed Ross
The C Users Journal
Octubre 1992

Lossless Data Compression
Steve Apiki
Byte
Marzo 1991

Data Compression using Huffman Coding
Dwayne Phillips
The C Users Journal
Febrero 1992

LZW Data Compression
Mark R. Nelson
Dr.Dobb's Journal
Octubre 1989

LZW Revisited
Shawn M. Regan
Dr.Dobb's Journal
Junio 1990

Data Compression with the Burrows Wheeler Transform
Mark Nelson
Dr.Dobb's Journal
Septiembre 1996

Papers

A Block-sorting Lossless Data Compression Algorithm
M. Burrow - D. J. Wheeler