PROYECTO FIN DE MÁSTER

POSGRADO EN INFORMÁTICA

UNIVERSIDAD DE ALMERIA

ESCUELA POLITÉCNICA SUPERIOR

MÁSTER EN TÉCNICAS INFORMÁTICA AVANZADAS

Optimización de código para arquitecturas superescalares de mapas autoorganizativos de Kohonen (SOMs)

Curso 2010/2011

Alumno/a:
José Gámez Poza

Director/es:
Gracia Ester Martín Garzón
José Jesús Fernández Rodríguez

Optimización de código para arquitecturas superescalares de mapas autoorganizativos de Kohonen (SOMs)

José Gámez Poza

Resumen—En este proyecto se estudian y aplican técnicas de desarrollo eficiente sobre un tipo de algoritmos conocimos con el nombre de SOM (Self-organizing map) o mapas autoorganizativos de kohonen. Para ello se ha partido de una serie de herramientas realizada por la universidad de helsinki, que nos proporcionan programas para la inicialización, el entrenamiento y visualización de estos mapas. Esta implementación nos aporta un punto de partida para aplicar técnicas que nos permitan mejoras los tiempos de ejecución del algoritmo. En lugar de realizar una optimización de cada herramienta el estudio se ha centrado en la optimización del programa de entrenamiento del mapa. Se han realizado una serie de mejoras centradas en aprovechar el paralelismo a nivel de instrucción en ordenadores superescalares. Apoyándose en mediciones de una serie de datos reales usados para el entrenamiento de este tipo de algoritmos se analiza el grado de optimización conseguido con cada una de las mejoras realizadas.

Resumen—In this project efficient development techniques for a type of algorithms, known as SOM (Self-organizing map) or autoorganizing Kohonen maps, are studied and applied. To this end, the work has been started with a number of tools, done by University of Helsinki, which provide us programs for the initialization, training and visualization of these maps. This implementation gives us an initial point to apply techniques allowing us to improve the execution times of the algorithm. Instead of realizing an optimization of each tool, the study is centered in the optimization of the training program of the map. A number of improvements have been realized concerning the exploitation of the parallelism of superscale computers at instruction levels. Based on the measurements of a number of real data used for the training of this type of algorithms, the optimization degree achieved with each realized improvement is analyzed.

Índice de Terminos—Optimización, mapas autoorganizativos, computadoras superescalares.

1 Introducción.

En este proyecto se pretende realizar una optimización de un algoritmo para arquitecturas superescalares; en la mayoría de los casos, cuando se implementan los algoritmos no se tiene en cuenta las características de los procesadores donde van a ser ejecutados. Por tanto, no se utilizan las máximas prestaciones de los mismos, desaprovechando parte de sus capacidades. Para conseguir un mayor rendimiento de los programas, los desarrolladores usan técnicas de optimización con el fin de conseguir mejoras en la velocidad de ejecución de las aplicaciones.

Cuando se plantea realizar una aplicación para resolver un problema, existen en algunas ocasiones limitaciones de tiempo que se necesitan cumplir. Estas limitaciones sólo se pueden comprobar cuando tenemos el programa finalizado y realizamos una ejecución del mismo. Ante una aplicación que no cumpla las especificaciones de tiempo se plantean alternativas para conseguir los resultados esperados. Este tipo de soluciones pasa por realizar una mejora en el hardware de la máquina, u optimizar el código que hemos desarrollado, con el fin de aprovechar en mayor medida la máquina de la que disponemos [1].

Para mejorar el rendimiento de una aplicación tenemos varias alternativas: usar arquitecturas más avanzadas y además desarrollar códigos que exploten adecuadamente el hardware de las arquitecturas. En este trabajo centramos nuestro interés en la última alternativa desarrollando código eficiente para obtener mejores tiempos, aprovechando al máximo el hardware del que disponemos. Para realizar

esta mejora de tiempos es necesario aprovechar al máximo el grado de paralelismo que dispone la aplicación [2], y hacerlos explicito mediante el hardware de que se dispone actualmente [3].

En nuestro caso nos hemos planteado optimizar un algoritmo de clasificación no supervisado denominado SOM. Se aplicarán técnicas de desarrollo eficiente sobre procesadores superescalares con el fin de obtener el mejor código para esta arquitectura. A partir de este código podremos plantear otras formas de obtener paralelismo para mejorar el rendimiento de estos algoritmos.

El principal uso de este tipo de algoritmos es realizar una clasificación automática de los datos, con el fin de poder visualizar las relaciones no-lineales de datos multidimensionales [4]. Este tipo de algoritmos, se engloban dentro de las técnicas de las redes neuronales, las cuales han sido desarrolladas y evolucionadas a partir de su utilización en el aprendizaje automático. El objetivo de este tipo de técnicas es obtener métodos computacionales, que sean capaces de inducir conocimiento a partir de ejemplos de un conjunto de datos de entrenamiento, que representan el sistema a estudiar. Así pues, los SOMs (Self-Organizing Map) son clasificadores no supervisados que han recibido una gran atención por parte de la comunidad científica. En particular, en el área de microscopia electrónica tridimensional y biología estructural. En esta rama, se emplean comúnmente para la clasificación de imágenes de macromoléculas biológicas. Una desventaja del algoritmo es su complejidad computacional. Por ello, en este trabajo se propone el empleo de técnicas de desarrollo de software eficientes para acelerar el algoritmo sobre los procesadores superescalares actuales. La evaluación de la implementación se realizará con datos experimentales de microscopia electrónica y sobre los computadores de que dispone el grupo de investigación.

El objetivo principal de este trabajo es el de mejorar el código del que partimos para que se ejecute de forma más rápida en ordenadores superescalares. Para ello se aplicarán técnicas que permitan aprovechar el paralelismo en estos procesadores. De esta forma se dispondrá de un código óptimo para procesadores superescalares, que podrá ser aprovechado aplicando otras técnicas de paralelismo y mejorando el rendimiento en los supercomputadores actuales.

En las siguientes secciones se muestran las técnicas de optimización de códigos que permiten aprovechar la arquitectura. Posteriormente se revisan las técnicas de clasificación existentes. Seguidamente presentamos el algoritmo SOM y una implementación del mismo: SOM_PAK. Por último mostramos las optimizaciones realizadas y los resultados obtenidos. Para finalizar se obtienen las conclusiones del trabajo realizado.

2 Optimizaciones para procesadores superescalares.

La contribución de la arquitectura al aumento de las prestaciones de los computadores ha venido de la mano del *paralelismo* y del aprovechamiento de la *localidad* de acceso a los datos, como estrategias para aprovechar las mejoras tecnológicas [2]. Se disponen pues, de dos estrategias para aumentar las prestaciones, replicar componentes del sistema o segmentar el uso de éstos [5].

En cualquier aplicación, existe un paralelismo implícito que puede ser extraído según el nivel de abstracción en el que nos centremos. Por tanto, es posible aumentar el rendimiento de las aplicaciones basándonos en el análisis y optimización de cada uno de estos niveles.

Como podemos observar en la Figura 1 dentro de una aplicación nos encontramos varios niveles de paralelismo implícito, que pueden ser explotados para mejorar el rendimiento. Según el nivel de abstracción en el que nos centremos los paralelismos se pueden clasificar [6]:

- *A nivel de programa.* Se intenta obtener el paralelismo mediante la ejecución de los programas que componen una aplicación en diferentes computadores o nodos de cómputo.
- *A nivel de función.* Se basa en la posibilidad de ejecutar las funciones que forman un programa en paralelo. Esto se podrá hacer siempre que las dependencias de datos lo permitan.

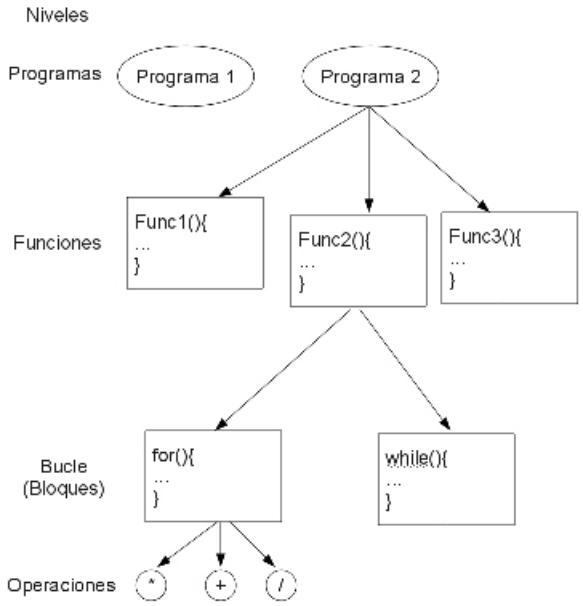

Figura 1. Niveles de Paralelismo.

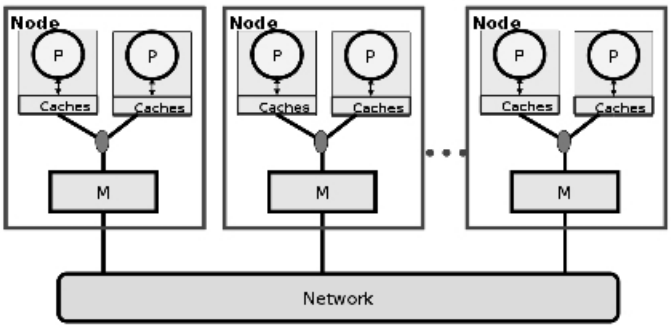

Figura 2. Arquitectura de un supercomputador tipo cluster

- *A nivel de bucle.* Se apoya en la idea de que cada ejecución de un bucle supone una tarea, por tanto, se pueden ejecutar en paralelo varias iteraciones de un bucle.
- *A nivel de operaciones.* Se basa en el aprovechamiento de que varias operaciones o instrucciones puedan ejecutarse de forma paralela.

En la actualidad, existen computadores de altas prestaciones que son capaces de mejorar el rendimiento de las aplicaciones. Para conseguirlo, hacen efectivo el aprovechamiento de los diferentes niveles de paralelismo descritos anteriormente, ofreciendo los recursos necesarios para que simultáneamente se completen los procesos paralelos.

Para aprovechar el primer y segundo nivel de paralelismo (tanto de programa como de función), las computadoras de altas prestaciones se organizan en nodos de cómputo. Estos nodos son interconectados de tal forma que puedan compartir recursos tal y como se muestra en la Figura 2. En cada nodo, se puede ejecutar en paralelo varios programas o funciones. De esta forma, se consigue que la aplicación disminuya su tiempo de ejecución. Ejemplos de este tipo de computadores se pueden encontrar en la siguiente referencia [3]. El supercomputador que actualmente ofrece más recursos es *Tianhe-1A* que se encuentra en el National Supercomputing Center de Tianjin [7]. Este supercomputador contiene seis mil ciento cuarenta y cuatro CPUs y cinco mil ciento veinte GPUs, distribuidos en dos mil quinientos sesenta nodos de cómputo y quinientos doce de operaciones.

Si disminuimos un nivel de abstracción en la arquitectura y nos centramos en los componentes que forman cada nodo, podemos ver como en cada uno tenemos generalmente procesadores y GPUs. Los primeros, son capaces de aportar paralelismo a nivel de procesos y funciones, por tener varios cores que permiten varias hebras de proceso, y paralelismo a nivel de instrucción, por tratarse de procesadores superescalares. Los segundos (GPU), son capaces de ofrecer paralelismo a nivel de datos, ya que nos permiten realizar la misma operación sobre varios datos simultáneamente. En el caso del computador *Tianhe-1A*, en cada nodo disponemos de dos procesadores Intel Xeon EP CPUs (4 cores) y un AMD ATI Radeon 4870x2 con 2 GPUs.

El objetivo de este proyecto es centrarnos en la explotación del paralelismo a nivel de instrucción. Para ello se pretende obtener el código más óptimo que se pueda ejecutar en un procesador superescalar. A partir de este código, y en posteriores trabajos, se podrá

seguir paralelizando la aplicación en los diferentes niveles de abstracción para que, de esta forma, se puedan obtener las máximas prestaciones de los supercomputadores que actualmente existen.

2.1 Elementos hardware en los que se centra la mejora de rendimiento a nivel de instrucción.

Si se analiza el número de *cliclos por instrucción* (CPI) que son necesarios para que se ejecute cada instrucción en un cauce, se puede obtener la siguiente formula:

$$CPI = CPIIdeal + Pext + PDatos + PCotrl$$

[6]. Siendo *CPIIdeal* la constante que marca el menor número de ciclo por instrucción que se necesitan para ejecutar una instrucción, *Pext* el número de ciclos de parada por riesgos estructurales, *PDatos* el número de ciclos de parada por riesgos de dependencias de datos y *PCotrl* el número de ciclos de parada producidos por riesgos de control.

Teniendo en cuenta que *CPIIdeal* es constante, tenemos que centrarnos en los demás elementos para realizar una optimización a nivel de instrucción. Así pues, para que nuestros programas presenten la mayor eficiencia posible, se tiene que estudiar bajo qué condiciones se producen estas paradas en los ordenadores superescalares, con el fin de minimizar el tiempo de penalización y por tanto disminuir el número de ciclos necesarios para ejecutar cada instrucción. En esta sección intentaremos explicar cuales son estos elementos que producen las paradas con el fin de poder evitarlos.

En una única instrucción básica, el paralelismo que se puede extraer es mínimo. Por lo tanto, se necesita un bloque con varias instrucciones básicas para poder realizar un solapamiento de las mismas y mejorar el rendimiento [6]. Así pues, del análisis de estos bloques, y de la organización y planificación que se realice de ellos, depende el éxito o el fracaso de una optimización a nivel de instrucción. En cualquier programa, si una sentencia depende de otra, no es posible ejecutar esas dos instrucciones de forma paralela, aunque parte de su ejecución puede solaparse por el

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
```

Figura 3. Dependencia de datos

cauce. Por tanto, en cualquier programa existen ciertos aspectos que provocan que dos instrucciones no puedan ser ejecutadas de forma paralela. Estos aspectos son [6]:

- *Dependencia de datos: Read After Write (RAW)* Es el principal motivo por el que se limita el paralelismo en un procesador. Una instrucción *j* es dependiente de *i*, si *i* produce un dato que es usado por *j*. O si *j* es dependiente de *k*, y *k* depende de *i*. En este caso, estas dos instrucciones no se pueden solapar y tienen que ser ejecutadas una detrás de la otra (*i* tiene que ejecutarse antes que *j*). Este tipo de dependencia se denomina 'verdadera' porque suele estar asociada a las propias características del código. No obstante se pueden tratar de eliminar la penalización que introducen estableciendo una planificación correcta en la escritura del código. Un ejemplo de este tipo de dependencia lo podemos ver en la Figura 3.
- *Dependencias de escritura de datos: Write After Read y Write After Write (WAR, WAW)* Se producen cuando dos instrucciones usan el mismo registro para guardar el resultado de una operación, pero no existe un flujo real de datos entre ellas. Esta dependencia es evitable mediante renombramiento de registros. Es necesario por tanto que se realice una buena planificación de los registros para que se evite este tipo de dependencia. Esta labor de planificación se realiza generalmente por el compilador, aunque en algunas ocasiones también puede ser realizado por el hardware.
- *Las dependencias de control.* Son las más usuales, ya que un programa no es una estructura secuencial y presenta estructuras de bifurcación. Una instrucción *i* que se encuentra dentro de una bifurcación, se ejecutará después de que la instrucción de

```
if P1 {
      S1;
};
if P2 {
      S2;
}
```

Figura 4. Dependencia de control

Instruction producing result	Instruction using result	Latency in clock cycles
FPALUop	Another FP ALU op	3
FPALUop	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figura 5. Latencias computador ficticio

salto sea evaluada. Así en el ejemplo de la Figura 4, *S1* depende de su instrucción de control *P1*. Y *S2* depende de su instrucción de control *P2*. Pero no existe dependencia entre *S1* y *S2*.

Estas dependencias son las que los compiladores actuales intentan evitar. Por consiguiente, para que no se produzcan dependencias de datos dentro de un cauce, la instrucción dependiente tiene que ser separada de su fuente un número de ciclos igual a la latencia que presente el cauce. La capacidad de un compilador para realizar esta planificación depende tanto del paralelismo que podamos obtener del programa, como de la latencia de las unidades funcionales que tengamos en el cauce.

Para ver la importancia del orden y la dependencia de datos veamos un ejemplo descrito en la referencia [6]. En él tenemos un computador ficticio con las latencias que muestra la Figura 5. Supongamos además el Código 1 con su traducción en código máquina. A partir de este código, sin realizar ninguna planificación, la ejecución sería la que muestra la Figura 6. Pero si utilizamos el *stall* de después de la carga para ejecutar la instrucción *DADDUI Rl,Rl,#-8*, podemos reducir la ejecución a siete ciclos (ver Figura 7).

En algunas ocasiones estas reordenaciones se detectan por el compilador. Pero existen en otras ocasiones en las que el compilador no es capaz de detectarlas, y es el programador, el

```
                                    Clock cycle issued
Loop:  L.D      F0,0(R1)            1
       stall                        2
       ADD.D    F4.F0.F2            3
       stall                        4
       stall                        5
       S.D      F4,0(R1)            6
       DADDUI   Rl,Rl,#-8           7
       stall                        8
       BNE      Rl,R2,Loop          9
```

Figura 6. Bucle sin ordenar.

```
Loop:  L.D      F0,0(R1)
       DADDUI   Rl,Rl,#-8
       ADD.D    F4.F0.F2
       stall
       stall
       S.D      F4,8(R1)
       BNE      Rl,R2,Loop
```

Figura 7. Bucle con ordenar.

que modifica la organización del código para que se genere un programa más eficiente.

```
for (i = 1000; i>0; i =i-1)
  x [ i ] = x [ i ] + s;

  Loop:  L.D FO.O(Rl)       ;F0=array element
         ADD.D F4,F0,F2     ;add scalar in F2
         S.D F4,0(R1)       ;store result
         DADDUI Rl,Rl,#-8   ;decrement pointer
                            ;8 bytes (per DW)
         BNE Rl,R2,Loop     ; branch R1!=R2
```

Código 1. Bucle sencillo

En nuestro caso, intentaremos realizar las modificaciones sobre el código a optimizar con la intención de intervenir en la planificación del compilador, obteniendo una ejecución mejorada y que necesite menos ciclos de parada provocados por las dependencias.

En esta sección hemos mostrado cuales son los elementos principales que limitan el paralelismo a nivel de instrucción. Para mejorar este paralelismo podemos intervenir mediante la utilización de algunas técnicas que hacen que el compilador pueda cambiar la planificación de las instrucciones, generando de esta forma menos dependencias. Éstas técnicas son las que se comentan en la siguiente sección de este documento.

2.2 Técnicas de optimización

Cuando se intentan desarrollar aplicaciones que aprovechen todo el potencial de un computador, el programador intenta sacar el máximo rendimiento enfocando el desarrollo del código bajo diferentes puntos de vista, según el nivel de abstracción en el que se centre. En nuestro caso se intentan aprovechar las capacidades para obtener el paralelismo a nivel de instrucción. Para conseguirlo, se modifican los programas para obtener el máximo partido del cauce y la utilización de las diferentes unidades funcionales. De esta forma se pretende que no se generen ciclos de parada en los que no se pueda ejecutar ninguna instrucción, para conseguir que el procesador siempre tenga una instrucción que ejecutar, aprovechando todas las capacidades del mismo y por tanto minimizando el tiempo de ejecución de una aplicación.

Otro aspecto que se tiene en cuenta a la hora de realizar códigos eficientes es el de las diferentes velocidades de acceso a memoria. Así pues, se realizan programas que minimicen el número de fallos en caché, para que la carga de datos se realice de la forma más rápida posible. En este sentido, los códigos serán más eficientes en la medida en la que tengan más localidad espacial y temporal en el acceso a los datos. Este aspecto tiene un impacto muy importante en el rendimiento cuando el tamaño de los datos con los que trabaja el programa es muy elevado.

En esta sección se mostrarán algunas técnicas que se aplican en los códigos, y que hacen que no se tengan que planificar ciclos de parada producidos por las dependencias descritas en el apartado anterior, y por la pérdida de tiempo derivada del acceso a las diferentes jerarquías de memoria. Además, se indica la finalidad de cada una de las técnicas y se ilustrará con un ejemplo de implementación usando el lenguaje de alto nivel C.

2.2.1 Aprovechar el paralelismo a nivel de instrucción.

Para empezar se indicarán algunas optimizaciones que se pueden realizar de carácter general y que son válidas para cualquier código que queramos desarrollar.

La primera de ellas es la directiva *register*. Ésta se podrá situar delante de la declaración del tipo de la variable, e indica al compilador que, siempre que pueda, sitúe la variable en un registro. Es ideal cuando tenemos un dato que vaya a ser referenciado en múltiples ocasiones en nuestro programa. El compilador intentará guardar ese dato en un registro, minimizando de esta forma el tiempo de acceso al mismo ya que al encontrarse en un registro se reutiliza más fácilmente que teniendo que cargarlo de caché. Aunque un uso abusivo de esta directiva o la combinación con otras optimizaciones puede producir efectos contraproducentes, al no tener registros suficientes para guardar los datos en memoria y producirse riesgos de nombre que el compilador tiene que resolver usando la memoria caché. Estas resoluciones pueden producir una mala planificación y limitar el paralelismo.

La segunda optimización que vamos a destacar es la de eliminar las operaciones que necesiten de muchos recursos para su ejecución, como por ejemplo la división. La utilización de la operación de división supone un coste computacional alto, ya que necesita muchos ciclos de reloj para completar su fase de ejecución. Por tanto, si existe una dependencia de datos o de control sobre esta instrucción, al necesitar de muchas unidades funcionales y necesitar muchos ciclos para su finalización, producirá ciclos de espera en otras instrucciones.

```
/* Codigo original */
float A[N][N];
register float factor=5.0;
register int i, j;
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i][j] /= factor;

/* Codigo tras realizar la eliminacion de la
   division */
float A[N][N];
register float factor=5.0;
register float ifactor;
register int i, j;
ifactor = 1.0/factor;
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i][j] *= ifactor;
```

Código 2. Eliminación de divisiones

En algunas ocasiones, cuando la división se

realiza dentro de un bucle, podemos minimizar el número de operaciones a realizar. Es el caso que se muestran en el código 2, se ilustra como se puede cambiar N*N divisiones, por una división y N*N multiplicaciones. Esta modificación supone un menor tiempo de cómputo.

Aunque estas optimizaciones generales son importantes y hay que tenerlas en cuenta a lo largo de todo el código, existe un elemento en la programación cuya optimización es esencial ya que presenta estructuras cíclicas, y supone una rama de ejecución en cada iteración, con los riesgos de control que se pueden producir. Pero también nos puede aportar paralelismo a nivel de instrucción, pues los elementos de su cuerpo pueden ser ejecutados de forma paralela, siempre que no exista dependencia de datos entre las diferentes iteraciones. Otro aspecto que nos hace centrarnos en estas estructuras, es su utilidad en el recorrido de matrices, pudiéndose aprovechar la localidad espacial. Es por este motivo por el que describiremos con más detalle las mejoras que se pueden realizar sobre los bucles.

Una de las primeras optimizaciones referidas a los elementos de salto que vamos a mostrar, es la eliminación de código independiente del interior del bucle. Esta optimización es sencilla de realizar, aunque en algunas ocasiones no se tiene en cuenta. La idea es eliminar las operaciones costosas con la utilización de constantes que se realicen dentro del bucle, ya que éstas consumen ciclos de procesamiento innecesarios, y en algunos casos producen dependencias de datos que el compilador no es capaz de detectar. Para eliminarlas del cuerpo del bucle, se ejecutan sólo una vez al inicio del mismo y se guarda el cálculo para utilizarlo en el interior. Podemos ver un ejemplo en el Código 3. En él, se guarda la ejecución del logaritmo en una variable (factor), para después utilizarla en la realización de la multiplicación. De esta forma, sólo tenemos que ir acumulando el valor de la multiplicación en un registro. En el ejemplo mostrado, se eliminan además N operaciones de logaritmo, que como indicamos anteriormente son costosas de realizar pues requieren muchos ciclos, y pueden generan ciclos de parada producidos por la dependencia de datos.

Otro aspecto importante es la eliminación de los saltos condicionales (if) dentro de los bucles. Con esta optimización se intenta mejorar la planificación de la ejecución mediante la eliminación de riesgos de control. El motivo de que este hecho se produzca, es porque se realiza un mejor uso de los circuitos de predicción que disponen internamente casi todos los procesadores superescalares, pudiéndose eliminar riesgos de control de forma dinámica. Esta circuitería consigue predecir si el salto va a ser realizado hacia adelante o hacia atrás, e insertan esa instrucción en el cauce mediante la modificación del contador de programa. Para realizar esta predicción los circuitos usan las estadísticas de los saltos realizados anteriormente, por tanto, un if en el interior de un bucle puede alterar las predicciones realizadas por este elemento del procesador. Además cuando se encuentra con una instrucción de salto, el compilador necesita planificar como mínimo la instrucción de carga de dirección de salto y la comparación, que mediante la eliminación sólo tienen que ser planificadas una vez, en lugar de en cada iteración de bucle.

```
/* Codigo original */
float A[N];
register int i;
for(i=0; i<N; i++)
  A[i]*=log(10.5*sin(5.0));

/* Codigo tras realizar la eliminacion de
    calculos redundantes */
float A[N];
register int i;
Register float factor;
Factor = log(10.5*sin(5.0));
for(i=0; i<N; i++)
  A[i]*= factor;
```

Código 3. Ejemplo de optimización para evitar cálculos redundantes

Pero una de las construcciones más simples para aprovechar el paralelismo a nivel de instrucción en los bucles es el *desenrollado de bucle*. La idea de esta técnica consiste en modificar el cuerpo del bucle, replicando las sentencias de su interior varias veces e incrementando los índices. Mediante esta técnica eliminamos el número de saltos que hay que realizar. Por tanto, se eliminan algunas dependencias de control y se pueden utilizar algunas paradas producidas por estas dependencias para la eje-

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F4,F0,F2
        ADD.D   F8,F6,F2
        ADD.D   F12,F10,F2
        ADD.D   F16.F14.F2
        S.D     F4,0(R1)
        S.D     F8,-8(R1)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        S.D     F16,8(R1)
        BNE     Rl,R2,Loop
```

Figura 8. Planificación de un desenrollado de bucle.

cución de otras instrucciones del bucle.

Si retomamos el ejemplo mostrado en el Código 1 y realizamos un desenrollado de bucle, obtendríamos el Código 4. Mediante este código se han eliminado algunas instrucciones DADDUI y BNE que además eran dependientes. Con este nuevo código, el compilador puede hacer una mejor planificación, reduciendo el número de paradas por la eliminación de dependencias. Así pues, bajo las condiciones indicadas en el apartado anterior (ver Figura 5), el compilador podría realizar la planificación que se muestra en la Figura 8. Como se puede ver, se han eliminado las paradas (stall) que aparecían en la Figura 7. Por tanto, se ha obtenido un código que es más rápido de ejecutar.

Como hemos visto, la técnica de desenrollado de bucle puede proporcionar grandes resultados a la hora de optimizar un código. Pero tal y como nos indica en [6], existen una serie de decisiones y transformaciones que hay que tener en cuenta:

- Determinar que el desenrollado es interesante ya que las instrucciones del cuerpo del bucle son independientes. Con excepción de las instrucciones de mantenimiento del bucle.
- Se usan diferentes registros para evitar restricciones innecesarias que se puedan imponer por la utilización del mismo registro para cálculos diferentes. En nuestro caso esta restricción no podrá ser controlada directamente, porque es el compilador el encargado de hacer las asignaciones. Aunque sí se puede controlar indirectamente mediante la utilización de la directiva *register* indicada anteriormente.
- Eliminar las comprobaciones extra y las instrucciones de salto, y ajustar la terminación del bucle y el código de repetición.
- Determinar que las cargas y las escrituras en el desenrollado del bucle se pueden intercambiar cuando son independientes con respecto a las de otra iteración. Para realizar esta transformación se requiere el análisis de las direcciones de memoria, para ver que no se refiere a la misma dirección.
- Planificar el código, mediante la preservación de cualquier dependencia para obtener el mismo resultado que en el código original.

En nuestro caso no podemos intervenir directamente en estas acciones porque son tarea del compilador y del procesador, pero sí que podemos intervenir indirectamente mediante la reordenación de nuestro código, de manera que enfoquemos la planificación que se tiene que realizar.

```
Loop:   L.D F0,0(R1)
        ADD.D F4.F0.F2
        S.D F4,0(R1)     ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1)    ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1)    ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

Código 4. Código ensamblador de desenrollado de bucle

Pero el desenrollado de un bucle no siempre tiene que producir ventajas en el rendimiento. Como se ha visto anteriormente al desenrollar un bucle se disminuyen las paradas, pero también existe un aumento del código fuente generado. Este aumento del número de líneas hace que el código ocupe más espacio y puede provocar problemas de fallos en la caché. Otro

factor casi más importante es el posible déficit de registros que se puede producir en el desenrollado del bucle. Este factor es denominado *register pressure* [6].

Se puede ver un ejemplo de la aplicación de esta técnica utilizando el lenguaje C en el Código 5. En él, se realiza un desenrollado de grado cuatro, porque cada iteración del nuevo bucle incluye la computación de cuatro iteraciones del bucle original. En este punto tenemos que comentar la importancia del grado, puesto que una de las ideas de esta optimización es aprovechar el número de unidades funcionales de que dispones en el procesador, creando un paralelismo en la ejecución de varias iteraciones del bucle. Los casos más generales son el desenrollado de grado 2 o de grado cuatro, ya que como anteriormente se mostró en las limitaciones, un desenrollado excesivo puede producir el efecto contrario por el déficit en la caché y por la limitación del número de registros. Así pues, el orden de desenrollado que mejores resultados aporte, estará estrechamente ligado con las características del procesador para el que estemos realizando las optimizaciones.

```
/* Codigo original */
int A[N],B[N];
register int i;
for(i=0; i<N; i++)
  B[i] = A[i] * c;

/* Codigo tras realizar el desenrrollado de
   grado 4*/
int A[N],B[N];
register int i;
Nend = 4 * (N / 4);
for(i=0; i<Nend; i+=4) {
  B[i] = A[i] * c;
  B[i+1] = A[i+1] * c;
  B[i+2] = A[i+2] * c;
  B[i+3] = A[i+3] * c;
}
for(i=Nend; i<N; i++)
  B[i] = A[i] * c;
```

Código 5. Ejemplo de desenrollado de bucle.

A pesar de ser una técnica muy fácil de aplicar, el desenrollado puede producir en algunas ocasiones una pérdida de rendimiento, ya que necesita un aumento de tamaño en la memoria caché [8]. Para ilustrar este caso veamos el ejemplo de Código 6, que realiza un desenrollado para aprovechar la carga en la variable *X[j]*. Si en este caso el tamaño de *N* para las columnas es lo suficientemente grande, puede producir que la fila no se pueda guardar en caché, ocasionando cada iteración del segundo bucle dos fallos en la caché. Uno al cargar y guardar el dato *X[i][j]*, y otro al cargar y guardar el dato *X[i+1][[j]*. Lo que en principio se planteaba como una mejora produce un efecto contrario.

```
/* Codigo original */
float A[N][N];
Float X[N],Y[N];
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i][j] += X[j]*Y[i];

/* Codigo tras realizar el desenrrollado de
   grado 2 para aprovechar la carga*/
float A[N][N];
Float X[N],Y[N];
for(i=0; i<N; i+=2){
  for(j=0; j<N; j++){
    A[i][j]+= X[j]*Y[i];
    A[i+1][j] += X[j]*Y[i+1];
  }
}
```

Código 6. Ejemplo de desenrollado para aprovechar la carga.

Estas optimizaciones son las que generalmente se realizan por el programador, eligiendo una u otra o una combinación de varias. Pero todas estas técnicas pueden ser aplicadas por el compilador de forma agresiva cuando utilizamos opciones de optimización. Éstas, se encuentran disponibles en casi todos los compiladores, aunque sus resultados dependen del compilador que estemos utilizando.

Para indicar al compilador que hemos usado en este trabajo (gcc) que utilice las optimizaciones, usaremos la opción "-On", donde *n* es un número entre cero y dos, siendo cero la opción que devuelve un código sin optimización, y dos con la máxima. A mayor optimización el compilador necesitará un mayor tiempo para compilar la aplicación, pero el código obtenido será más eficiente. Así pues, podemos esforzarnos en realizar optimizaciones, y que el compilador las absorba al realizar las suyas propias, consiguiendo éste mejores tiempos que los nuestros. Incluso puede que no consigamos ninguna mejora respecto a la optimización más agresiva del compilador, en nuestro caso la -O2. Pero en algunas

ocasiones, como se verá posteriormente, sí se consigue mejorar el tiempo. Es por este motivo por lo que la labor de optimización es un proceso tedioso y en el que no siempre se obtienen los resultados esperados. Sí debemos de comentar que la mejor situación es conseguir que las mejoras del compilador se combinen con las realizadas por el programador, en este caso se conseguiría mejorar la optimización original del compilador, aunque suele ser complicado de conseguir.

2.2.2 Aprovechamiento de los accesos a la memoria caché

Cuando optimizamos el acceso a la jerarquía de memoria, necesitamos saber como se guardan los datos físicamente en la memoria principal. Este dato nos lo marca generalmente el lenguaje de programación que estemos usando. Dependiendo del lenguaje, los datos matriciales se sitúan por filas o por columnas en la memoria principal. Así pues, si sabemos como están los datos colocados, y puesto que la transferencia entre memoria principal y memoria caché se realiza por bloques de datos consecutivos, se podrán realizar implementaciones de los algoritmos de forma que los datos se vayan obteniendo en el orden en el que se sitúan en la memoria principal, de esta forma, se minimizarán los fallos en la memoria caché. Seguidamente para la mejor compresión de esta apreciación, lo detallaremos mediante un ejemplo.

Consideremos una matriz que se encuentra guardada en memoria tal y como se indica en la Figura 9. Supongamos además que tenemos intención de recorrer esa matriz obteniendo los valores guardados en ella y realizando un sumatorio de los mismo en la variable *sum*. Tendremos que recorrer nuestra matriz obteniendo cada elemento para poder realizar la operación deseada. Pero la forma en la que recorramos la matriz puede hacer que aprovechemos mejor la localidad en la caché o peor. Si miramos el Código 7, podemos ver la implementación de un recorrido por columnas de la matriz. Al estar la matriz guardada por filas en la memoria principal y suponiendo que sólo se puedan almacenar cinco elementos en la memoria caché. Podemos

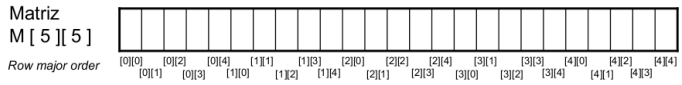

Figura 9. Matriz en memoria

ver como esta forma de recorrer los elementos produce muchos fallos en la memoria caché, pues el segundo elemento, fila 1 columna 0, no se encontraría en la memoria y produciría un fallo, repitiéndose esta situación con cada dato. En este caso con la solicitud de cada elemento se produciría un fallo y tendríamos que acceder a memoria principal con la pérdida de tiempo que ello supone (acceder al dato y copiar el bloque de memoria principal a memoria caché). En cambio, si estudiamos el ejemplo de Código 8, podemos apreciar como el recorrido de la matriz se realiza por filas. De esta forma, y suponiendo el mismo caso de memoria caché que indicamos anteriormente, podemos ver como el primer acceso produce fallo de caché pero los siguientes cuatro no. De esta forma sólo se produciría el fallo en caché cuando se cambiase de columna. A pesar de que ambos ejemplos realizan el mismo algoritmo el Código 8 resulta más eficiente en la ejecución de un ordenador superescalar suponiendo que la memoria del mismo este organizada tal y como se indica en la Figura 9, y bajo las condiciones del tamaño de bloque referidas en este supuesto. Con este ejemplo, queda más claro la importancia que tiene para el programador el hecho de conocer cómo se sitúan los datos en memoria, dependiendo del lenguaje y del compilador que se utilice a la hora de realizar códigos eficientes.

```
float A[200][200];
float sum=0.0;
for(j=0; j<200; j++)
  for(i=0; i<200; i++)
    sum += A[i][j];
```

Código 7. Código sin optimizar

```
float A[200][200];
float sum=0.0;
for(i=0; j<200; j++)
  for(j=0; i<200; i++)
    sum += A[i][j];
```

Código 8. Código optimizado

Tras haber visto como se organizan las

máquinas en las que ejecutamos nuestros programas, y de mostrar cómo se organizan los datos en memoria, hemos podido determinar algunas técnicas que hacen que mejore la eficiencia de nuestro algoritmo. Cómo son las técnicas de paralelización de instrucciones que aprovechan la arquitectura superescalar para ejecutar varias instrucciones simultáneamente. Y las que se basan en la localidad temporal de los datos, con el fin de aprovechar las característica de jerarquía de memoria en los ordenadores superescalares, ya que, como comentamos anteriormente, el acceso a la memoria caché (memoria de primer nivel) es mucho más rápido que el acceso a la memoria principal, o a los discos físicos [1] [9] [8]. En las secciones posteriores se describirá el algoritmo sobre el que vamos a trabajar y mostraremos como se han aplicado estas optimizaciones a dicho algoritmo.

3 Estado del arte sobre los algoritmos de agrupamiento.

Los algoritmos de agrupamiento se basan en una técnica denominada análisis de cluster. El análisis de cluster, también llamado segmentación o análisis taxonómico, es una técnica cuya finalizad es la de hacer una partición de un conjunto de objetos en grupos. De esta forma, los objetos que quedan englobados en el mismo cluster comparten un perfil (son similares entre si), mientras los objetos de diferentes cluster tiene un perfil totalmente distinto. Estas técnicas son utilizadas para obtener características relevantes de los objetos [11].

Una de las características necesarias para realizar este tipo de análisis, es que los objetos deben tener propiedades que puedan ser valoradas de forma numérica. Por lo tanto en el caso de no disponer de esta característica, se tendrá que realizar una transformación del atributo para representarlo como un número.

De forma genérica para realizar este tipo de análisis se realizan los siguientes pasos:

1) Encontrar la similitud o disimilitud entre cada par de objetos. En este paso, se debe calcular las distancias entre objetos usando una función de distancia. Generalmente la más utilizada es la distancia euclidea, la cual mide la proximidad entre dos puntos en el espacio. Aunque existen otras formas de realizar este cálculo tal y como es descrito más adelante.
2) Agrupar los objetos en un árbol binario de jerarquía de cluster. En este paso se deben agrupar los objetos que se encuentran próximos. Una forma de hacer esto es ordenar las distancias entre pares de elementos de acuerdo a la proximidad que existe entre ellos.
3) Por último debemos determinar donde dividir el árbol de jerarquía de cluster.

La definición del problema a tratar por estos algoritmos es la que sigue: dado un conjunto de objetos y opcionalmente la caracterización de cada elemento del cluster mediante un vector de valores que determinan los atributos (generalmente numéricos), se pretende clasificar los objetos de forma que cada objeto se asocie con un elemento del cluster. Se puede plantear incluso el problema de clasificación de acuerdo a criterios de similitud entre los elementos y no de acuerdo a características del cluster definidas previamente. Es nuestra idea realizar la agrupación por similitud y extraer el conocimiento.

Para realizar esta agrupación se parte de una muestra de objetos de entre el conjunto global. A partir de ésta se pretende obtener agrupaciones de objetos (cluster) de modo que los objetos pertenecientes al mismo grupo sean semejantes entre si, mientras que objetos de grupos diferentes tengan la mayor diferencia posible.

Conviene sin embargo estar siempre alerta ante el peligro de obtener como resultado del análisis no una clasificación de los datos, sino una disección de los mismos en distintos grupos que sólo existen en la memoria del ordenador. Es por tanto necesario el conocimiento que el analista tenga acerca del problema y que determinará cuáles de los grupos obtenidos son significativos y cuáles no.

Una vez definido el espacio sobre el que se pretende trabajar, tenemos que establecer una medida que determine el grado de proximidad entre los diferentes objetos. Estas medidas determinan el grado de semejanza entre dos objetos. La idea es que cuanto mayor semejanza

tengan los objetos mayor probabilidad tendrá el método de clasificación de ponerlos en el mismo grupo. Los objetos tienen mayor posibilidad a menor semejanza de quedar englobados en grupos distintos.

Para determinar las semejanza entre los diferentes objetos se utilizan generalmente el cálculo de distancias. Muchas son las formas de calcular estas distancias, pero las más utilizadas son:

- **Distancia euclidea.** La distancia euclidiana en tres dimensiones entre dos puntos con p_1 en (x_1, y_1, z_1) y p_2 en (x_2, y_2, z_2) está definida por $d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2 + ... + (z_1 - z_2)^2$
- **Distancia euclidea ponderada.** Es similar a la distancia euclidea pero con sus elementos ponderados. $d^2(x_i, x_j) = \sum_{K=1}^{n} a_k(x_{ki} - x_{kj})^2$. Donde a_k representan los coeficientes de ponderación. Con la utilización de esta forma de cálculo de similitud conseguimos dar más importancia a unos elementos o atributos del objeto que a otros, utilizando para ello los coeficientes.
- **Distancia Manhattan.** La distancia Manhattan mide la distancia entre dos puntos en los diferentes planos. Así, si tenemos p_1 en (x_1, y_1) y p_2 en (x_2, y_2) la distancia Manhattan se calcula como $|x_1 - x_2| + |y_1 - y_2|$
- **Distancia de Pearson.** Esta distancia está basada en el coeficiente de correlación de Pearson, el cual es calculado a partir de los valores muestrales y la desviación estándar. El coeficiente de correlación *r* toma valores desde –1 (correlación negativa más grande) a +1 (correlación positiva más grande). La distancia de Pearson dp es calculada como $dp = 1 - r$ y éste está entre 0 y 2.
- **Distancia Mahalanobis.** Define la similitud diferenciándose de la distancia euclidea en que tiene en cuenta la correlación entre las variables. Se calcula como: $d^2(x_i, x_j) = (x_i - x_j)^T C^{-}1(x_i - xj)$. Donde C es una matriz diagonal cuyos elementos representan la varianza.

Ninguna de estas formas de cálculo de distancias (medición del grado de similitud) es mejor que otra. Por tanto dependerá del problema que estemos analizando y será responsabilidad del programador la elección de una u otra, teniendo en cuenta cual puede ser la que mejor se adecue al problema tratado.

Como se puede apreciar todas las distancias anteriores están definidas utilizando variables numéricas. Puede darse el caso de que en algunas ocasiones, nos encontremos con que no todas las variables que definen el objeto se encuentran en el tipo de dato que nos interesa. En estos casos tendremos que realizar una transformación antes de usar los datos para el cálculo de las distancias, o usar éstos para la interpretación y análisis de resultados y desecharlos en el cálculo de similitudes. En este caso Anderberg [12] y Gordon [13] proponen soluciones para este problema.

Según la forma en la que se plantee la realización de este tipo de algoritmos se pueden ver como: un método estático, si se describe como un conjunto de objetos que dan lugar a un agrupamiento, dados los valores de sus características y que no varía. Pero también puede verse como un método dinámico, si utilizamos la información que conocemos de los objetos de una agrupación anterior como elemento de entrada, e ingresamos nuevos objetos para redefinir una nueva agrupación [11].

3.1 Métodos de clasificación de cluster

Una vez definido cuales son los métodos mediante los cuales podemos calcular las distancias en los algoritmos de agrupamiento, pasamos a comentar los diferentes métodos de clasificación que se pueden realizar de este tipo de algoritmos. Un posible método de clasificación es aquel que atiende a si el número de clases es conocido a priori o se va generando dinámicamente. Así pues se puede dividir en:

- *Agrupación en número de clases desconocido:* Son usados cuando no disponemos de información del número de clases en las que se debe de dividir el grupo de objetos. En este tipo, es el propio algoritmo el que debe de decidir si aumentar o disminuir el número de grupos, así como reordenar los objetos en el nuevo conjunto de elementos creados. En este tipo, el número

de grupos se va recalculando según se van procesando objetos en base a unos parámetros de división o agrupamiento que son definidos.
- *Agrupación en un número conocido de clases:* A diferencia del caso anterior, se ofrecen una partición única de los objetos en un número de cluster. Éste número de clases suele ser fijado de antemano, proporcionando como parámetro este dato al algoritmo. O no determinado ”a priori” y tendrá que ser el algoritmo el que determine cuál es el mejor número de agrupaciones en las que se debe dividir, teniendo en cuenta los objetos dados. En ambos casos el número de grupos se calcula como primera operación del algoritmo.

Otro tipo de método de clasificación son los denominados Métodos jerárquicos. Éstos, no ofrecen una única partición, sino un conjunto de ellas anidadas de forma jerárquica. Dependiendo de las ramas que elijamos tendremos una agrupación u otra. Estos métodos se pueden subdividir a su vez en:

- *Métodos aglomerativos.* En ellos se parten de todos los objetos que queremos clasificar formando un grupo en sí mismo. Cada grupo estará formado por un único objeto. A partir de éstos se realizan asociaciones de esos objetos formando nuevos grupos, los cuales vuelven a asociar en un proceso iterativo hasta que todos forman un único grupo.
- *Métodos divisivos.* En este caso estaríamos ante un procedimiento opuesto al anterior. En él se inicia partiendo de un único grupo con todos los objetos y se van realizando divisiones hasta que lleguemos a grupos de un único objeto que formarían las hojas de la jerarquía.

Como se puede deducir de la definición de los anteriores algoritmos, al utilizar las técnicas jerárquicas dependiendo de cómo se realicen las divisiones o uniones obtendremos agrupaciones diferentes.

Es necesario destacar que las técnicas jerárquicas pueden resultar ineficientes para tamaños grandes de la muestra. Por lo tanto no se utilizarán estos tipos de algoritmos en el análisis que vamos a presentar en esta memoria.

3.1.1 Algoritmos de agrupamiento con número de clases desconocidas

En este tipo de algoritmos de agrupación se clasifican los objetos en un conjunto de clases desconocidas. A priori no podremos determinar el número de grupos que se crearán. En el avance de la ejecución es el algoritmo el encargado de determinar si debe de crear un nuevo grupo, este proceso normalmente se realiza mediante la división de un grupo en dos, quedando la mitad de los objetos del grupo original asignados a cada uno de los nuevos grupos. O por el contrario se deben de reducir el número de clases, este proceso se realiza mediante la fusión de dos grupos. Para determinar la actuación que se debe de realizar en cada caso (dividir un grupo o fusionar dos grupos), se deben definir en el algoritmo unos criterios para la creación o la eliminación de los grupos .

Un ejemplo de este tipo de algoritmos es el algoritmo Min-Max. Éste se basa en la introducción paulatinamente de centroides de clases e ir determinando la distancia a la que se encuentran los patrones (elementos característicos u objetos a clasificar) a dichos centroides. Si en algún caso esta distancia superara un umbral establecido se crea un nuevo centroide (grupo) y se vuelven a clasificar los objetos. El proceso se repite hasta que no se producen cambios en los centroides. Este algoritmo sólo precisa para su ejecución la determinación de un valor de umbral para la autoorganización de las clases. Creará automáticamente una nueva clase cuando la distancia entre centroides supere el umbral.

La especificación formal de este algoritmo es la siguiente:

1) Se obtiene al azar un elemento de los P disponibles y se inserta en la clase α.
2) Se calcula la distancias de α_1 a los P-1 vectores no agrupados y se toma la máxima distancia, formando una nueva clase con aquél que la maximiza.
3) Se agrupan los restantes vectores que quedan en alguna de las clases existentes.

4) Si la distancia de alguno de los vectores está a una distancia superior a un valor umbral: crear una nueva clase con ese vector y se vuelve al paso 3; en otro caso, se termina.

Las agrupaciones obtenidas como solución de este tipo de algoritmos, estarán determinadas principalmente por el umbral que hace que se creen nuevas clases y por el orden en el que se vayan tomando los elementos. Debido a la influencia del orden en el que se tomen los objetos a agrupar y de la aleatoriedad del primer paso, nos encontramos ante un algoritmo que puede que no realice agrupaciones que nos permitan obtener información útil. Por éste motivo, en muchas ocasiones se utilizan este tipo de algoritmos para obtener el dato de los grupos que aproximadamente deberíamos de realizar, para posteriormente, aplicar un algoritmo con número de clases conocido.

3.1.2 Algoritmos de agrupamiento con número de clases conocidas

Este tipo de algoritmos nos agrupa los objetos en un número de clases que debemos determinar a priori y que representa una de las entradas del mismo. Así pues, se irán procesando los elementos por parte del algoritmo y los irá clasificando según al número de grupos que le hemos indicado, quedando asignados en alguno de ellos. En este tipo de algoritmos se usa heurísticas tanto para determinar el número de clases como para realizar la agrupación.

Un ejemplo de los métodos de agrupación con número de clases conocidas, es el popularmente denominado como algoritmo de las k-medias. Éste, además de ser muy utilizando, ha sufrido a lo largo del tiempo varias modificaciones para adaptarlo a diferentes tipos de datos, tal y como se puede apreciar en el artículo [14]. En él se proponen dos variantes Binary-K-means y Divisive-Binary-K-means. La intención principal en la que se basa este algoritmo es en minimizar la distancia interna de las diferentes clases que se define como el centroide de un agrupamiento. Así pues, podemos decir que consiste en minimizar la distancia de cada elemento de la clase a los centroides. Y maximizar la distancia entre centroides.

Existen diversas formas de definir formalmente el algoritmo, pero una de ellas es:

1) Se seleccionan inicialmente una configuración para los cluster. Bien con la elección de los n primero objetos o mediante la definición de unos centroides elegidos al azar.
2) Mientras que exista un vector diferente a los anteriores realizar los siguientes pasos:
   a) Se distribuyen las muestras entrantes entre los grupos. Se asigna cada vector a la clase con una distancia euclídea menor (aunque se puede utilizar otro método para el cálculo de distancias).
   b) Se calculan los nuevos centroides de los cluster.

Una de las ventajas que presenta este algoritmo es que es sencillo y eficiente. Además procesa los patrones secuencialmente por lo que requiere un almacenamiento mínimo. Por otra parte, el principal inconveniente que presenta el algoritmo de las k-medias es que su comportamiento depende del parámetro K que indica el número de agrupaciones que queremos hacer [15]. Sobre todo que este parámetro sea fijo durante toda la ejecución.

En el caso que vimos en la subsección anterior el algoritmo tenía la posibilidad de crear una nueva clase para un elemento que no se parezca a ninguno de los anteriores, pero en este caso debe asignarlo a un grupo de entre los existentes. Así pues, un objeto que sea muy diferente a los anteriormente procesados, puede provocar un desplazamiento del centroide. Además si no realizamos una asignación inicial de centroides (por ejemplo de forma aleatoria), los primeros elementos tienen mucha influencia en el algoritmo pues determinan la configuración inicial de los agrupamientos. Sin embargo, los últimos elementos que se seleccionen apenas influyen en la solución final.

Son por los motivos indicados por lo que para utilizar estos algoritmos, aunque en general es válido para cualquier algoritmo de agrupamiento, hay que tener cuidado con los elementos de entrada. Una mala elección de

los objetos de entrenamiento puede producir una agrupación no acorde a la realidad y que no represente correctamente la estructura subyacente.

A pesar de los defectos anteriormente citados, el algoritmo de las k-medias ha sido muy utilizado por ser sencillo y por los pocos recursos que consume. Es por eso por lo que a partir de él se han obtenido una serie de mejoras. Así pues podemos citar el "Algoritmo de agrupamiento secuencial", el cual presenta como novedades que el número de agrupamientos se considera como un valor máximo. Además existe una fase de evaluación que permite reducir el número de agrupamientos en base a algunos criterios y procesar por lotes los patrones (elementos de entrenamiento del cluster). O el "Algoritmo de ISODATA" que presenta heurísticas para dividir (aumentar el número de agrupamientos) o mezclar (reducir el número de agrupamientos). Como se puede apreciar en los ejemplos casi todas las mejoras realizadas han ido en la línea de flexibilizar el número de clases, posicionándose entre los algoritmos con número de clases conocido y los de número de clases desconocido.

Pero la principal efectividad de un algoritmo de agrupamiento depende de la forma y tamaño de los grupos naturales que están contenidos en los datos procesados de entrenamiento. La mayoría de los métodos presupone alguna estructura inicial de los datos y no tratan de inferir la organización subyacente de los mismos. Por tanto no se puede afirmar que un método es mejor que otro, o que un mismo algoritmo funcionará siempre bien para cualquier tipo de dato, porque la calidad de los resultados que se obtengan dependerá tanto de las presunciones del método que se indiquen como de los datos sobre los que hayan sido ejecutados [14].

En el caso de imágenes obtenidas mediante un telescopio el algoritmo más utilizado es el de mapas autoorganizativos de Kohonen (SOMs), este tipo de algoritmo tiene su principal aplicación en la inteligencia artificial, por su gran capacidad para inferir conocimiento a partir de un conjunto de ejemplos que son utilizados como entrenamiento. Además como cualquier red neuronal, a mayor número de datos su entrenamiento es mayor y sus agrupaciones se adecuan más a la realidad. Son por estos motivos por los que las optimizaciones se realizarán sobre este algoritmo.

4 EL ALGORITMO SOM.

Los mapas autoorganizativos o SOM (Self-Organization Map), también llamados redes de Kohonen, son un tipo de red neuronal. Este tipo de redes, por la teoría en la que se basan, y las amplias posibilidades de aplicación que tienen, se han convertido en uno de los principales paradigmas en el campo de la clasificación y aprendizaje. En la actualidad los campos de aplicación más importantes son: análisis de datos experimentales, reconocimiento de patrones, análisis del habla, robótica, diagnóstico médico e industrial y control de procesos.

En un mapa autoorganizativo existen dos elementos característicos que lo forman. Uno es una capa generalmente bidimensional de nodos de salida. Y el otro una entrada de dimensión n que define las entradas que se usarán para el entrenamiento de la red. En la Figura 10 podemos observar los diferentes elementos que forman el mapa organizativo. Destacamos el vector de entrada de n componentes (X), m_i, que es un vector de dimensión igual a la entrada (X), que está asociado al nodo i, y que se conoce como vector diccionario o vector de referencia.

El objetivo de esta estructura es realizar un proceso de entrenamiento sobre la misma para descubrir la estructura subyacente de un conjunto de datos que son introducidos en ella, obteniendo conocimiento de estos datos. Este proceso se realiza mediante un aprendizaje no supervisado, denominado así porque no se conoce *a priori* la clase a la que pertenece cada patrón de entrada en la red. En este tipo de aprendizaje es la red la que establece una serie de categorías, atendiendo a los elementos de entrada y a los criterios de similitud que se le indiquen.

De forma conceptual, podemos decir que la finalidad del algoritmo del mapa autoorganizativo es asociar el patrón de entrada a un nodo de salida. De forma que los elementos

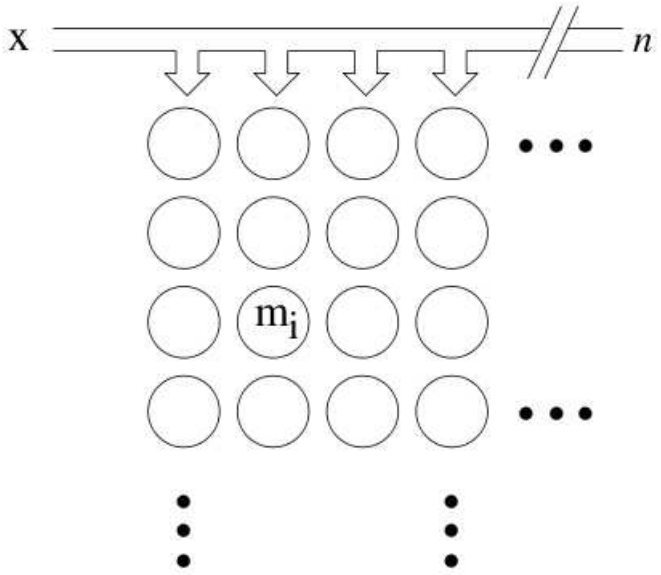

Figura 10. Matriz de nodos (o neuronas) de salida de un mapa auto-organizativo

que se encuentran asociados sobre nodos adyacentes son más similares que los asignados a nodos distantes. Antes de continuar definiremos cuándo unos nodos son adyacentes (son vecinos) y cuándo no, ya que existen varias formas de definir la vecindad.

Aunque existen varios tipos de red neuronal, ya que podemos tener redes de varias dimensiones y de diferentes tipologías, en nuestro caso utilizaremos redes de dos dimensiones. Las formas más utilizadas para este tipo de redes son las tipologías rectangular y hexagonal. En la tipología rectangular, cada nodo dispone de cuatro vecinos o nodos adyacentes. El superior, el inferior, el de la izquierda y el de la derecha. Estos nodos se disponen como se muestra en la Figura 11. En ella el nodo denotado con (2,1) dispone de un vecino superior (2,0), un vecino inferior (2,2), un vecino derecho (3,1) y un vecino a la izquierda (1,1). Tal y como se indicó anteriormente estos vecinos tendrán más semejanzas con el nodo (2,1) que por ejemplo el nodo (0,0), que se encuentra más distante. La utilización de esta tipología como se puede apreciar en la figura obtiene como resultado una red neuronal con forma rectangular. El otro caso utilizado es el denominado *topología hexagonal* que da lugar a una red neuronal en forma de colmena, tal y

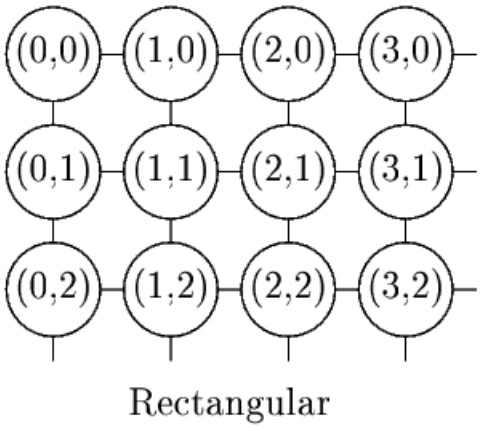

Figura 11. Topología de vecinos Rectangular

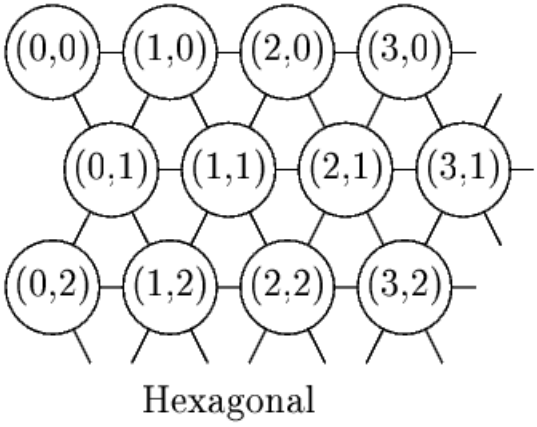

Figura 12. Topología de vecinos Hexagonal

como se puede ver en la Figura 12. En este caso cada nodo dispone de seis vecinos o nodos adyacentes. Estos nodos formarían los vértices de un hexágono rodeando al nodo del que son vecinos. Así pues, en la Figura 12 el nodo (1,1) tiene seis nodos adyacentes que son: (1,0) , (2,0), (2,1) (2,2), (1,2) y (0,1). Estos nodos son más similares al nodo indicado que el nodo (3,2) que se encuentra más distante.

Una vez definido cuál es el objetivo que pretende el algoritmo pasamos a definir cómo se realiza esta agrupación. Inicialmente, se parte de una red neuronal vacía, ésta puede ser inicializada de diferentes formas, bien asignándole vectores de pesos aleatorios a cada elemento o nodo, o mediante un algoritmo de inicialización. Esta etapa de inicialización es muy importante ya que nos guiará el proceso de agrupamiento posterior. Tras este proceso

de inicialización, cada neurona tiene un vector asociado (vector de pesos o prototipo) que tiene la misma dimensión que los datos de entrada. En cada iteración del algoritmo, se le proporciona a la red un elemento (vector) de los que queremos clasificar. Se calcula la similitud (generalmente mediante la distancia euclídea) del vector de entrada con cada uno de los prototipos de las neuronas. El vector de pesos que tiene una mayor similitud (menor distancia) con el vector de entrada determina la neurona ganadora (BMU Best-Matching Unit o Unidad con mejor ajuste). Queda pues este vector asociado al agrupamiento de dicha neurona. Tras esta etapa, se produce un ajuste de los prototipos de la neurona ganadora y de su vecindad, esta última puede variar según la tipología elegida tal y como vimos anteriormente. Una vez realizado el ajuste, se continúa el proceso con el siguiente elemento de entrada. Como resultado del algoritmo se obtiene cada uno de los vectores de entrada y en que neurona ha quedado englobada. Además tendremos una nueva red cuyos vectores prototipo han quedado modificados por este proceso. Tendremos por tanto una nueva red que tiene un mayor conocimiento de la estructura subyacente, y que puede ser usada en posteriores ejecuciones con más vectores (elementos) de este tipo de problema, produciéndose de esta forma un entrenamiento que hará que cada vez los elementos queden mejor representados.

Es importante a la vista del algoritmo elegir bien los vectores de entrada, ya que de los datos de entrenamiento que se hayan usado a lo largo del tiempo para la mejora de la red dependerá la utilidad de la misma, y que represente mejor o peor la estructura subyacente. Aunque tal y como se indicó, lo importante para una red son los vectores prototipo ya que son los elementos representativos de los diferentes nodos que forman el mapa. Y son los que determinan que un elemento de entrada quede englobado en un nodo o en otro.

5 SOM_PAK. Una implementación del algoritmo SOM.

En lugar de realizar una implementación del algoritmo SOM propia se ha partido de una existente que se encuentra bastante madura. Este paquete de aplicaciones denominado SOM_PAK está formado por un conjunto de programas que nos permiten trabajar sobre algoritmos organizativos. Nos proporcionan pues una serie de programas para realizar la inicialización de la red, el entrenamiento de la misma y la posterior visualización de los resultados. En nuestro caso el que nos interesa es el de entrenamiento de la red y es por tanto sobre éste sobre el que realizaremos las optimizaciones.

Tenemos diferentes programas que nos suministran las herramientas necesarias para trabajar con el algoritmo SOM. De acuerdo con [16], los programas y opciones más importantes son:

- *vfind* - Este programa, nos permite la creación de un buen mapa inicial. Para conseguirlo crea diferentes mapas mediante una serie de inicializaciones internas y procedimientos de entrenamiento internos. El criterio para seleccionar un buen mapa es que tenga una baja cuantización del error.
- *lininit* - Inicializa el mapa de forma ordenada, usando los vectores que se le proporcionen como entrada al programa. Cada uno de éstos será el prototipo de uno de los nodos.
- *mapinit* - Permite realizar una inicialización de forma genérica.
- *randinit* - Nos proporciona una inicialización del mapa de forma aleatoria. Para ello se generan vectores prototipo de dimensión N donde cada una de sus posiciones son un número aleatorio.
- *sammon* - Genera un mapa de Sammon de dimensión N, siendo N la dimensión del vector de entrada y asociando a aquella célula de dimensión dos, la cual tiende a aproximarse a la distancia euclidea de la entrada.
- *umat* - Este programa, utilizado para mostrar los datos, genera un fichero postsScript con escala de grises en el que se puede apreciar la distancia entre los vectores de referencia de las vecindades del mapa.
- *qerror* - Nos indica la cuantización del error a partir de una serie de vectores de entrada

y de un mapa.

- *vcal* - Este programa es usado para preparar los datos para ser visualizados. Su utilidad es la calibración de los datos de entrada con las etiquetas que se pueden proporcionar al mapa.
- *planes* - Este programa de visualización genera un código PostScript de la imagen del mapa en escala de grises pero filtrando por un plano (dimensión) en particular. Si el archivo de entrada de datos también se le aporta al programa, la trayectoria formada por la mejor unidad de coincidencia se devuelve en otro archivo diferente.
- *visual* -Este programa es utilizado para la visualización y generación de una lista de coordenadas que indica cual ha sido la unidad en la que ha quedado englobada cada elemento de la entrada.
- *vsom* - Es el programa que realiza el entrenamiento de la red. Mediante una serie de vectores de entrada (elementos encargados de entrenar la red) aplica el algoritmo descrito en el apartado anterior. Como resultado, nos devuelve un nuevo mapa entrenado que se adecua mejor a la organización de los datos.
- *-buffer* - Es una opción del programa vsom. Esta opción introducida a partir de la versión 3.0 del paquete nos permite realizar la lectura de un número de líneas del fichero de entrada. Estas líneas son cargadas en la memoria principal. Así pues, por ejemplo, si definimos un buffer de cien, el programa leerá de cien en cien líneas sobre el fichero de entrada, cargando estos datos en la memoria principal del ordenador.

De todos estos programas en el proyecto nos hemos centrado en la optimización del denominado **vsom**, pues es éste el que nos proporciona el entrenamiento del Mapa.

6 Análisis del código SOM_PAK y optimizaciones realizadas.

En cualquier algoritmo, podemos encontrar ciertas partes que durante su ejecución son más referenciadas que otras. Éste hecho produce que a la hora de abordar una optimización de código, debamos de realizar un análisis para identificar estas porciones de código y, de este modo, centrar los esfuerzos en hacer un desarrollo eficiente sobre ellas. El motivo de esta forma de actuación es porque mejorando estas secciones de código, que generalmente consumen más tiempo de ejecución, (bien son más veces referenciadas por varias funciones, o bien porque consume más tiempo por ser funciones complejas) conseguiremos mejorar el rendimiento global del algoritmo. Nos encontramos pues ante una fase complicada y delicada. Y de la elección de concentrar nuestros esfuerzos en una parte u otra, puede depender el éxito o el fracaso de la optimización y en consecuencia del rendimiento del algoritmo. La complicación de esta fase radica en cómo identificar estas partes, pues la llamada a una función u otra depende en muchas ocasiones del resultado de una variable que evaluamos en una clausula if, de los datos de entrada, o de las configuraciones que hemos hecho en el programa. Es pues difícil determinar sobre la especificación del algoritmo cual será la función que consuma más tiempo. Pero resulta algo más sencillo cuando ya disponemos de una implementación, ya que como veremos más adelante existen herramientas que nos determinan qué funciones son las que consumen un mayor tiempo al realizar una ejecución.

En nuestro caso partimos de una implementación inicial del algoritmo SOM que nos proporciona el paquete SOM_PAK, por lo que partimos de un código inicial que pretendemos acelerar mediante técnicas de desarrollo eficiente. Además existen una serie de herramientas que nos ayudan en este proceso de análisis, de forma que no tendremos que hacerlo manualmente explorando el código e intuyendo por donde puede pasar o depurando el mismo. Las herramientas utilizadas nos proporcionarán datos estadísticos que nos indicarán que partes del código son en las que tendremos que poner el mayor esfuerzo en su optimización. En nuestro caso hemos hecho uso de una herramienta denominada gprof.

Gprof es una herramienta bajo licencia GNU que puede ser descargada libremente y modificada. La última versión es de Diciembre de 2010 (la 2.21) aunque en nuestro caso hemos

utilizado la 2.20 por ser la que nos proporciona la distribución linux que estamos usando. Tal y como se nos indica en la documentación [17], este software es utilizado para determinar qué partes de un programa son las que están consumiendo el mayor tiempo de la ejecución. El programa nos ofrece estadísticas de los tiempos que consume cada función, pudiendo identificar las partes más lentas o que más veces son llamadas. Éste grupo de funciones son candidatas a ser reescritas para conseguir un mejor rendimiento global del programa. Al grupo de funciones a optimizar se le denomina normalmente puntos calientes.

Para conseguir este propósito hay que realizar tres pasos. El primero es generar el programa con datos que permiten a gprof generar las estadísticas. Este procedimiento es similar a cuando se compila y se enlaza un programa con datos para su posterior depuración. En este caso, la opción que debemos poner al compilador gcc (usado en este trabajo) es "-pg". Una vez realizada la compilación con dicho parámetro, debemos de ejecutar el código fuente normalmente. Y como resultado de esta ejecución se generará un fichero denominado *gmon.out*. Este fichero contendrá las estadísticas de la ejecución de nuestra aplicación. Si lo abrimos podremos ver que es un fichero binario que no se puede interpretar. Para realizar la interpretación de este fichero y como último paso para obtener las estadísticas de ejecución, hay que ejecutar gprof seguido del nombre del programa ejecutable. La ejecución se deberá realizar estando dentro de la carpeta donde se encuentre el programa ejecutable y el fichero "gmon.out" generado en el paso anterior. Como resultado de esta ejecución nos aparecerá una tabla con las estadísticas de cada función ejecutada en el programa. Si alguna de las funciones no aparecen será porque tienen un tiempo de ejecución despreciable.

En nuestro caso la ejecución de gprof nos ha proporcionado los datos representado mediante la Tabla 1. En ella se indican los porcentajes de tiempos que han consumido cada función y los tiempos consumidos por las mismas, tanto acumulativos como los que ha necesitado para su ejecución. El orden en el que aparecen las funciones es de la función que más porcentaje ha consumido primeramente, a la que menos tiempo ha consumido en último lugar.

%time	Cumulative second	self second	name
64.95	12.90	12.90	find_winner_euc
29.61	18.78	5.88	adapt_vector
4.08	19.59	0.81	getline
1.16	19.82	0.23	load_entry
0.20	19.86	0.04	find_conv_to_ind

Tabla 1
Tabla con la ejecución de gprof

Del análisis de la Tabla 1, se puede deducir como la función que más tiempo consume en el algoritmo es la función find_winner_euc, cuyo código es el representado mediante el Código 9. No es extraño el resultado obtenido, ya que es la que realiza la comparativa del cálculo de distancia con los vectores prototipo, para determinar el elemento en el que queda englobado el vector de entrada. A la vista de los resultados obtenidos y teniendo también en cuenta la distancia con la siguiente función, podemos concluir que: find_winner_euc será la sección de código en la que más esfuerzo y tiempo nos interesa invertir en su optimización [1]. Es por tanto nuestro *punto caliente*.

Aunque adapt_vector es la segunda función en la que más tiempo se invierte, no se realizaron optimizaciones sobre ella. El motivo es porque al compararla con el punto caliente vemos que es mucho menor el porcentaje de ejecución. Además al estudiar el código se observó que el esfuerzo que se tenía que realizar para mejorar el código iba a ser mucho mayor que los resultados obtenidos. Por tanto no era rentable centrarse en la optimización de este método. En consecuencia, tampoco resultaría rentable abordar un estudio de desarrollo eficiente sobre las funciones getline, load_entry y find_conv_to_ind.

Antes de abordar las optimizaciones que se realizaron sobre el Código 9, es necesario realizar un estudio de la función para entender bien la lógica de la misma. Como ya indicamos en secciones anteriores, tenemos que centrar nuestro análisis en las sentencias condicionales y cíclicas, ya que son éstas las que presentan

rupturas en el código, son las que producen los saltos en la ejecución. Además son generalmente las encargadas, como en este caso, de realizar los recorridos de vectores. Y es un punto donde podemos aprovechar la localidad temporal.

```
int find_winner_euc(struct entries *codes,
    struct data_entry *sample,
      struct winner_info *win, int knn)
{
  struct data_entry *codetmp;
  int dim, i, masked;
  float diffsf, diff, difference;
  eptr p;

  dim = codes->dimension;
  win->index = -1;
  win->winner = NULL;
  win->diff = -1.0;

  /* Go through all code vectors */
  codetmp = rewind_entries(codes, &p);
  diffsf = FLT_MAX;

  while (codetmp != NULL) {
      difference = 0.0;
      masked = 0;

      /* Compute the distance between codebook
          and input entry */
      for (i = 0; i < dim; i++)
      {
        if ((sample->mask != NULL) && (sample
            ->mask[i] != 0))
        {
          masked++;
          continue; /* ignore vector
              components that have 1 in mask
              */
        }
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff * diff;
        if (difference > diffsf) break;
      }
      if (masked == dim)
        return 0; /* can't calculate winner,
            empty data vector */
      /* If distance is smaller than previous
          distances */
      if (difference < diffsf) {
        win->winner = codetmp;
        win->index = p.index;
        win->diff = difference;
        diffsf = difference;
      }

      codetmp = next_entry(&p);
    }
    if (win->index < 0)
     ifverbose(3)
       fprintf(stderr, "find_winner_euc: can't
           find winner\n");
  return 1; /* number of neighbours */
}
```

Código 9. Procedimiento del programa que consume más porcentaje de tiempo de ejecución.

La primera estructura que es caso de estudio es la sentencia while. Éste bucle realiza un recorrido de todos los vectores prototipos de los diferentes nodos que forman la red neuronal en el algoritmo SOM. En nuestra implementación la función que obtiene los elementos uno a uno. Apoyándonos para su consecución en las funciones "rewind_entries", que nos devuelve el primer elemento, y "next_entry", que obtiene los siguientes elementos. En este proceso de carga de vectores de prototipo, pienso que no se podrá mejorar mucho el rendimiento, ya que no existen operaciones aritméticas ni de salto de las cuales se pueda hacer una optimización. Pero existe un bucle interior en el que se realizan operaciones aritméticas que puede ser susceptible de ser optimizado. Nos referimos al bucle *for* del Código 9. En él se realiza el cálculo de la distancia euclídea entre el vector que presentamos a la red (representado mediante la variable *sample*, vector de entrada) y cada uno de los vectores prototipo de la misma. Estos vectores son los que se guardan en la variable *codetmp*. En esta construcción tenemos por tanto operaciones condicionales dentro de un bucle y variables que son muy utilizadas. Así pues, se podrán aplicar varias de las técnicas presentadas en secciones anteriores para mejorar esta función y en consecuencia el comportamiento global del algoritmo.

Seguidamente mostraremos las propuestas de optimización realizadas. Se indicará por qué se han aplicado y sobre qué mejoras están fundamentadas. En la próxima sección se presentarán los resultados obtenidos de las ejecuciones realizadas y se podrá ver en qué medida mejora las ejecuciones originales.

La primera mejora que se realizó es incluir la directiva "register" en las variables de tipo float *diff* y *difference*. La utilización de esta directiva delante del tipo de la variable indica al compilador que, en la medida de lo posible, las variables sean guardadas siempre en un registro de memoria [8]. Se piensa que los dos

datos elegidos son los más idóneos para ser guardadas en registros, ya que como podemos apreciar en el Código 9, estos datos son utilizados de forma masiva en todas las iteraciones de bucle. Es importante no realizar un uso abusivo de esta directiva ya que una utilización indebida puede producir un efecto contraproducente. Si pretendemos guardar todas las variables en registros no dispondremos de espacio suficiente, ocasionando muchos intercambios entre memoria caché y el registro. En el resto de mejoras se mantuvo esta directiva porque se vio que presentaba una reducción del tiempo de ejecución.

```
for (i = 0; i < dim; i++)
{
  if (difference > diffsf) break;
  if ((sample->mask != NULL) && (sample->mask[
      i] != 0))
  {
    masked++;
    continue; /* ignore vector components that
        have 1 in mask */
  }
  diff = codetmp->points[i] - sample->points[i
      ];
  difference += diff * diff;
}
```

Código 10. Cambio de orden de las sentencias if.

```
  Nend = 4 * (dim/4);
  for (i = 0; i < Nend; i+=4)
  {
    if ((sample->mask != NULL) && (sample->
        mask[i] != 0))
    {
      masked++;
      continue; /* ignore vector components
          that have 1 in mask */
    }
    diff = codetmp->points[i] - sample->points
        [i];
    difference += diff * diff;
    if (difference > diffsf) break;
    // Segunda iteraccion del desenrrollado
    if ((sample->mask != NULL) && (sample->
        mask[i+1] != 0))
    {
      masked++;
      continue; /* ignore vector components
          that have 1 in mask */
    }
    diff = codetmp->points[i+1] - sample->
        points[i+1];
    difference += diff * diff;
    if (difference > diffsf) break;
    // Tercera iteraccion del desenrrollado
    if ((sample->mask != NULL) && (sample->
        mask[i+3] != 0))
    {
      masked++;
      continue; /* ignore vector components
          that have 1 in mask */
    }
    diff = codetmp->points[i+3] - sample->
        points[i+3];
    difference += diff * diff;
    if (difference > diffsf) break;
    // Cuarta iteraccion del desenrrollado
    if ((sample->mask != NULL) && (sample->
        mask[i+4] != 0))
    {
      masked++;
      continue; /* ignore vector components
          that have 1 in mask */
    }
    diff = codetmp->points[i+4] - sample->
        points[i+4];
    difference += diff * diff;
    if (difference > diffsf) break;
  }

  //iteracciones finales
  for (i = Nend; i < Nend && difference <
      diffsf; i++){
    if ((sample->mask != NULL) && (sample->
        mask[i] != 0))
    {
      masked++;
      continue; /* ignore vector components
          that have 1 in mask */
    }
    diff = codetmp->points[i] - sample->points
        [i];
    difference += diff * diff;
    if (difference > diffsf) break;
  }

  if (masked == dim)
      return 0; /* can't calculate winner,
          empty data vector */

    /* If distance is smaller than previous
        distances */
    if (difference < diffsf) {
      win->winner = codetmp;
      win->index = p.index;
      win->diff = difference;
      diffsf = difference;
    }

    codetmp = next_entry(&p);
  }
```

Código 11. Mejora de desenrollado de grado 4

En la segunda optimización, se buscó mejorar la predicción de salto dentro del bucle. Para ello se realizó una mejora muy sencilla pero que en algunas ocasiones produce unos rendimientos bastante importantes. La idea es alterar el orden de las sentencias condicionales *if* para ver si la lógica de predicción hardware mejora los tiempos de ejecución del algoritmo.

La implementación de esta mejora es la que presenta el Código 10. Sólo se ha mostrado la porción del bucle porque es la única parte modificada.

```
/* Compute the distance between codebook and
    input entry */

    if(sample->mask != NULL){
      for (i = 0; i < dim && difference <
          diffsf; i++)
      {
        flag = (sample->mask[i] != 0);
        masked += flag;
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff*(1-flag);
      }
    }else{
      for (i = 0; i < dim && difference <
          diffsf; i++)
      {
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff;
      }
    }
```

Código 12. Mejora mediante eliminación de condicionales en el interior de un bucle

Tras realizar esta optimización se planteó realizar una mejora que aprovechara las unidades funcionales de que dispone el ordenador superescalar. Para ello se realizó un desenrollado de bucle de grado 4. El código de esta optimización se puede apreciar en el Código 11. En esta mejora hay que destacar cómo en las iteraciones finales se vuelve a comprobar que la variable *difference*, ya que en el caso de que sea menor que la cota (variable *diffsf*), no será necesario la ejecución del cuerpo de bucle, ahorrando algo de tiempo. En este supuesto en el bucle principal habremos salido por la interrupción del bucle mediante break.

```
 /* Compute the distance between codebook and
     input entry */
    Nend = 4 * (dim/4);
    if(sample->mask != NULL){
      for (i = 0; i < Nend && difference <
          diffsf; i+=4)
      {
        //ITER 1
        flag = (sample->mask[i] != 0);
        masked += flag;
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff*(1-flag);
        //ITER 2
        flag = (sample->mask[i+1] != 0);
        masked += flag;
        diff = codetmp->points[i+1] - sample->
            points[i+1];
        difference += diff*diff*(1-flag);
        //ITER 3
        flag = (sample->mask[i+2] != 0);
        masked += flag;
        diff = codetmp->points[i+2] - sample->
            points[i+2];
        difference += diff*diff*(1-flag);
        //ITER 4
        flag = (sample->mask[i+3] != 0);
        masked += flag;
        diff = codetmp->points[i+3] - sample->
            points[i+3];
        difference += diff*diff*(1-flag);
      }
      //iteracciones finales
      for (i = Nend; i < Nend && difference <
          diffsf; i++){
        flag = (sample->mask[i] != 0);
        masked += flag;
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff*(1-flag);
      }
    }else{
      for (i = 0; i < Nend && difference <
          diffsf; i+=4)
      {
        //ITER 1
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff;
        //ITER 2
        diff = codetmp->points[i+1] - sample->
            points[i+1];
        difference += diff*diff;
        //ITER 3
        diff = codetmp->points[i+2] - sample->
            points[i+2];
        difference += diff*diff;
        //ITER 4
        diff = codetmp->points[i+3] - sample->
            points[i+3];
        difference += diff*diff;
      }
      //iteracciones finales
      for (i = Nend; i < Nend && difference <
          diffsf; i++){
        diff = codetmp->points[i] - sample->
            points[i];
        difference += diff*diff;
      }
    }
```

Código 13. Desenrollado de bucle de grado 4 realizado sobre la mejora 4

La cuarta optimización que se realizó tenía la intención de eliminar las sentencias condicionales que se encontraban en el interior del bucle *for*. Tal y como se indicó, las sentencias de bifurcación dentro de un bucle producen una penalización, muy alta ya que no permiten

que se realice una predicción de salto correcta, perdiendo algunos ciclos en cada iteración. Tal y como se puede ver en el Código 12 en nuestro caso hemos eliminado el primer *if*, y lo hemos implementado al principio del bucle usando, un flag para guardar la condición. El segundo, lo hemos llevado al interior de la condición del bucle. De esta forma eliminamos las dos sentencias condicionales.

En la última optimización se realiza un desenrollado de bucle del código generado con la mejora 4 (Código 12). El trozo que nos interesa de esta mejora es el que se indica en el Código 13. El grado de desenrollado es 4. Y se puede apreciar la similitud a la hora de operar con la mejora 3. Lo cual nos indica que el realizar un desenrollado de bucle es una tarea bastante mecánica.

Como se puede apreciar en todas las optimizaciones anteriores, el realizar código más eficiente produce una implementación menos legible del algoritmo. Es por este motivo por el que se realiza en una última fase de desarrollo de código. Además, en algunos casos, se debe primar el entendimiento de las líneas en pos de la eficiencia. Por este motivo sólo se realizan las optimizaciones en ciertas partes del programa y siempre tras comprobar que el programa original que se ha generado no cubre nuestras necesidades temporales como era nuestro caso.

7 Resultados obtenidos y análisis de las mejoras con respecto a las realizadas por el compilador

En esta sección se analizan los resultados obtenidos mediante las ejecuciones realizadas con las diferentes optimizaciones. Además, con el fin de poder comparar las mejoras con las que realiza el compilador, se han generado diferentes programas utilizando para ello varias opciones de compilación. Así pues, se han compilado de todas las optimizaciones realizadas dos versiones de los programas: Una con la realización de una compilación con la opción -O0, de esta forma podemos ver cómo se comportaba el algoritmo sin incorporar ninguna mejora de rendimiento por parte del compilador. La otra con la opción -02, para ver si las optimizaciones automáticas llevadas a cabo por el compilador mejoraban los tiempos de las mejoras realizadas en el código.

De cada uno de los programas generados se han realizado cinco ejecuciones y se ha obtenido el promedio. El motivo de realizar cinco ejecuciones es para poder apreciar que el tiempo es estable y no existe influencia de otros programas que se esté ejecutando en la máquina.

Antes de empezar el análisis de los datos tenemos que comentar que en todas las figuras siguientes se ha usado: *Orig* para referirnos a los tiempos obtenidos con el código original. Y cada una de las mejoras se representará mediante la letra *M* y el número de la mejora. Así pues, *M1* se referirá a la mejora realizada con la directiva *register*, *M2* al cambio de orden en las sentencias if, *M3* al desenrollado de bucle de la *mejora 2*, *M4* a la eliminación de condicionales del cuerpo del bucle, y *M5* al desenrollado de bucle de la *mejora 4*.

En una primera fase, se realizó una toma de tiempos de las optimizaciones descritas en el apartado anterior pero sin utilización de un buffer para la lectura. En la segunda fase se realizaron las mediciones sobre los mismos programas pero utilizando un buffer de lectura de 500 posiciones. Éste nos lo proporciona la opción -buffer que nos permitía el paquete SOM_PAK tal y como se indicó en anteriores secciones.

Si analizamos la Figura 13, que representa la comparativa de tiempos entre todas las mejoras realizadas y el código original, podemos observar cómo con la *mejora 1*, se reduce el tiempo en más de diez segundos. Sin embargo, con las otras mejoras se obtienen tiempos similares al original. Este hecho que puede resultar extraño, ya que los otros códigos contienen a la mejora uno, puede tener su explicación en la planificación de los registros que el compilador tiene que hacer. Así pues, al modificar las sentencias condicionales del interior del bucle o hacer un desenrollado, el compilador realiza una peor planificación de los datos que guarda en los registros, y tenemos que hacer uso de caché. Esto se puede apreciar sobre todo en la *mejora 3*, donde implementamos un desenrollado mante-

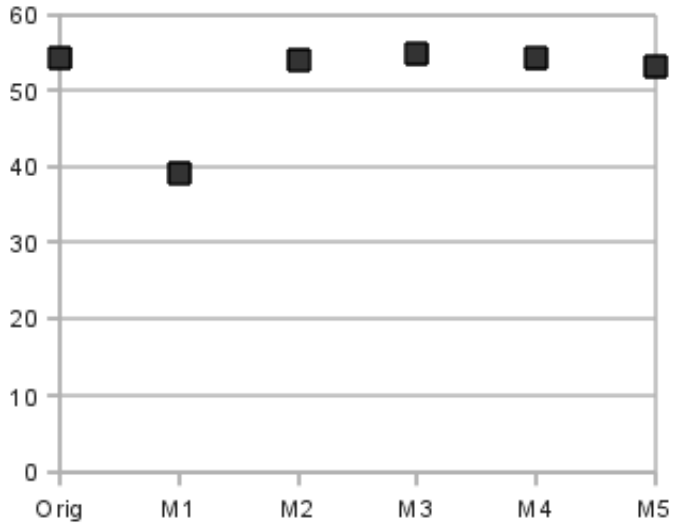

Figura 13. Gráfica comparativa con la opción -O0 y sin buffer

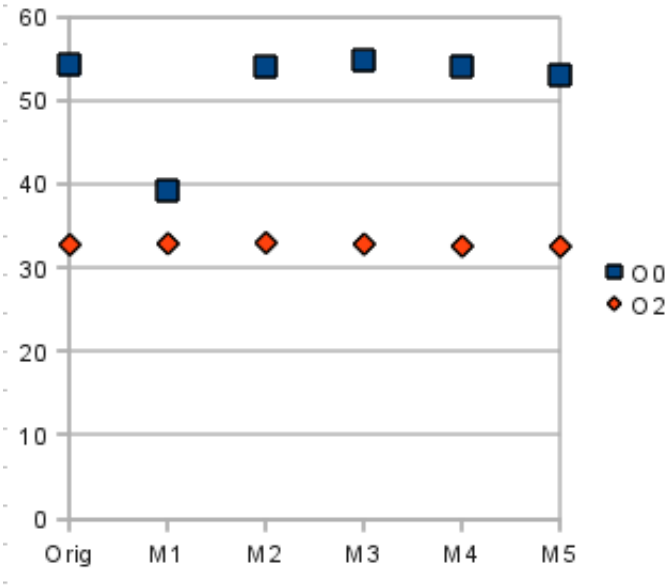

Figura 14. Comparativa de las opciones -O0 y -O2 sin buffer

niendo las sentencias if en el interior del bucle y obtenemos un tiempo similar al original. En la *mejora 5* obtenemos mejores tiempos porque hemos eliminado algunos riesgos de control al quitar los saltos condicionales del cuerpo del bucle.

A pesar de la optimizaciones realizadas, y de que hemos conseguido reducir el tiempo de ejecución en más de diez segundos con la mejora 2, tal y como se puede observar en la Figura 13. Si realizamos una comparativa entre los tiempos sin aplicar opciones de optimización del compilador, y los resultados obtenidos tras su aplicación, podemos observar (ver Figura 14) que las mejoras del compilador son muchos más agresivas y reducen el mejor tiempo que hemos conseguido. La mejor opción para obtener mejoras que nos podemos plantear es implementar un código que se convine bien con las optimizaciones del compilador y mejore los tiempos obtenidos con la opción -O2.

En este punto del trabajo se observó, al revisar la documentación de SOM_PAK, que en la versión sin parámetros la implementación del algoritmo no realizaba ningún uso de buffer en todo el proceso. Así pues, estábamos realizando todas las lecturas desde disco con la consiguiente demora que suponía. Se pensó pues en utilizar la opción de buffer que proporciona SOM_PAK. De esta forma se cargarán un número de datos del disco a la memoria principal, en nuestro caso quinientos elementos. Así pues en una segunda fase, se repitieron todas las ejecuciones con esta opción.

En esta segunda fase se esperaba que, como muchas de nuestras mejoras estaban encaminadas en el aprovechamiento de la localidad, los resultados fueran mejores. Si realizamos pues una comparativa de los tiempos obtenidos en la primera fase, y los que obtenemos en esta segunda, sin ninguna intervención del compilador. Obtenemos la gráfica que se muestra en la Figura 15. En esta gráfica podemos ver que el tiempo necesario para la ejecución del algoritmo se ha reducido, llegando en el caso de la *mejora 2* a ser de veinticinco segundos. A pesar de esta reducción podemos ver cómo se vuelven a repetir los tiempos relativos, lo cual nos indica que con el buffer hemos conseguido localidad temporal en los datos, pero el problema de los registros comentado anteriormente sigue presente.

Si analizamos la comparativa entre los tiempos con las diferentes opciones del compilador, obtenemos la Figura 16. En esta gráfica podemos ver cómo los datos son muy similares a la opción del buffer. En este caso la opción de la *mejora 2* vuelve a ser la que más se acerca a las mejoras del compilador. Pero existe un dato que merece la pena destacar y que se puede

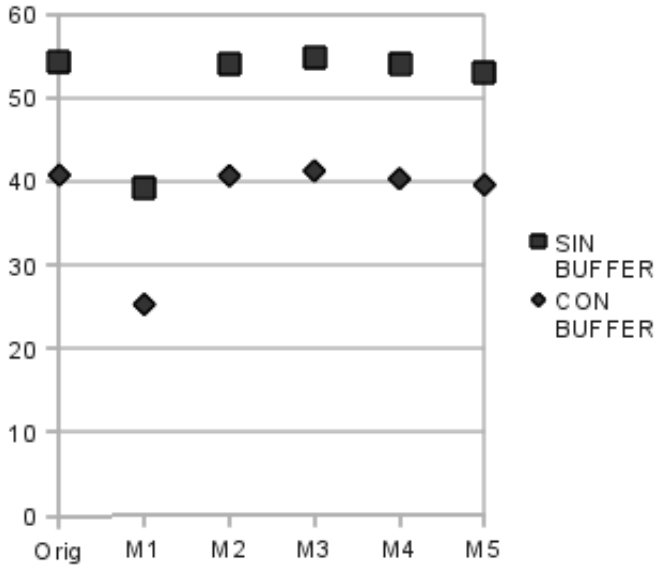

Figura 15. Comparativa de los tiempos con buffer y sin buffer con la opción -O0 del compilador.

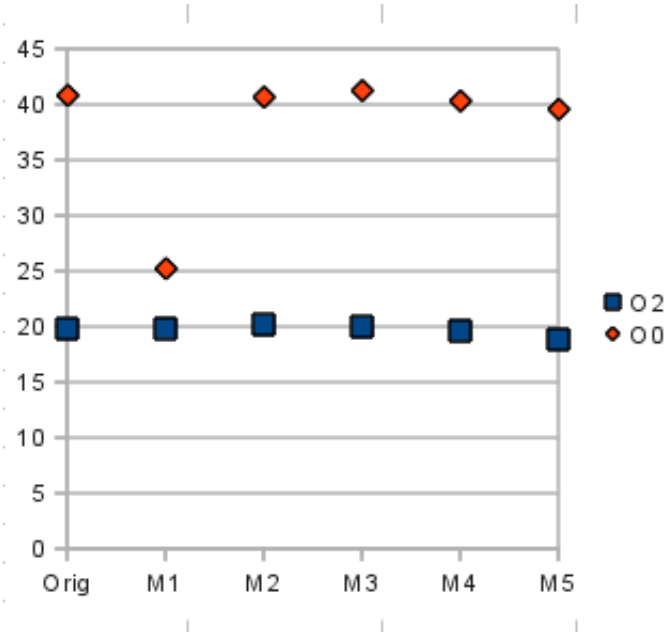

Figura 16. Comparativa de las opciones -O0 y -O2 con buffer.

apreciar mejor en la Figura 17. Es el tiempo obtenido mediante la *mejora 5* y la optimización *O2*, en ella hemos conseguido que nuestras mejoras se combinen con las del compilador, obteniendo una mejora de un segundo respecto de la segunda mejor ejecución, y de treinta y seis segundos respecto al código original del que partíamos.

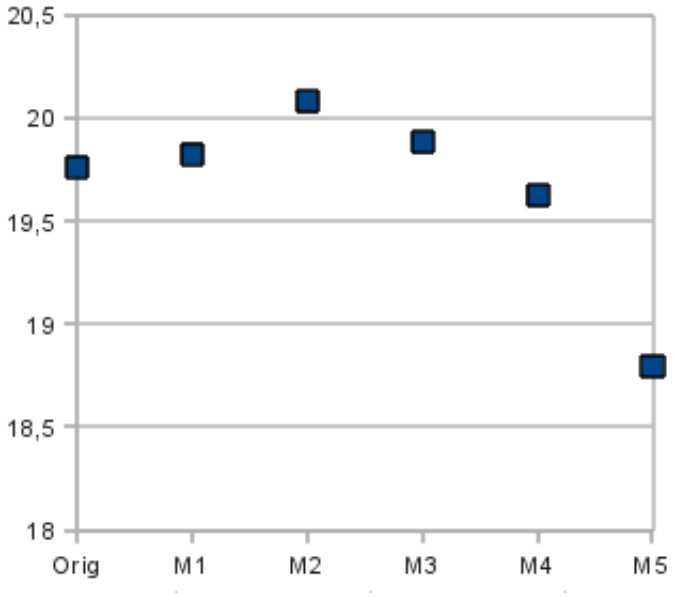

Figura 17. Resultados de la opción -O2 con buffer.

Tras analizar las gráficas anteriores, se puede comprobar que la optimización M5 combina bien con las optimizaciones realizadas por el compilador. En este punto se realizan una serie de ejecuciones aumentando el tamaño del problema para apreciar si esta mejora es real o es consecuencia de la carga del sistema. Así pues se obtienen datos de mapas de tamaños desde 20x20 nodos hasta 80x80 nodos. Y con vectores de entrenamiento, y por tanto cada prototipo, de tres mil seiscientas componentes.

El resultado de estas ejecuciones es el que se puede observar en la Figura 18, en ella se muestran los tiempos obtenidos por las diferentes mejoras con y sin la optimización del compilador. Hay dos datos que llaman la atención en la figura: el primero, es la diferencia que aporta el utilizar las mejoras del compilador, se obtienen mejoras de hasta setecientos diecinueve segundos. El segundo, es como la combinación de la *mejora 5* con las realizadas por el compilador es la que mejores resultados aporta, este hecho se puede apreciar por la separación que existe entre esta mejora y el resto con el aumento del tamaño del problema. Podemos apreciar con mayor claridad el dato en la Figura 19, que muestra sólo las ejecuciones con la optimización del compilador. El tiempo exacto de mejora para el tamaño más

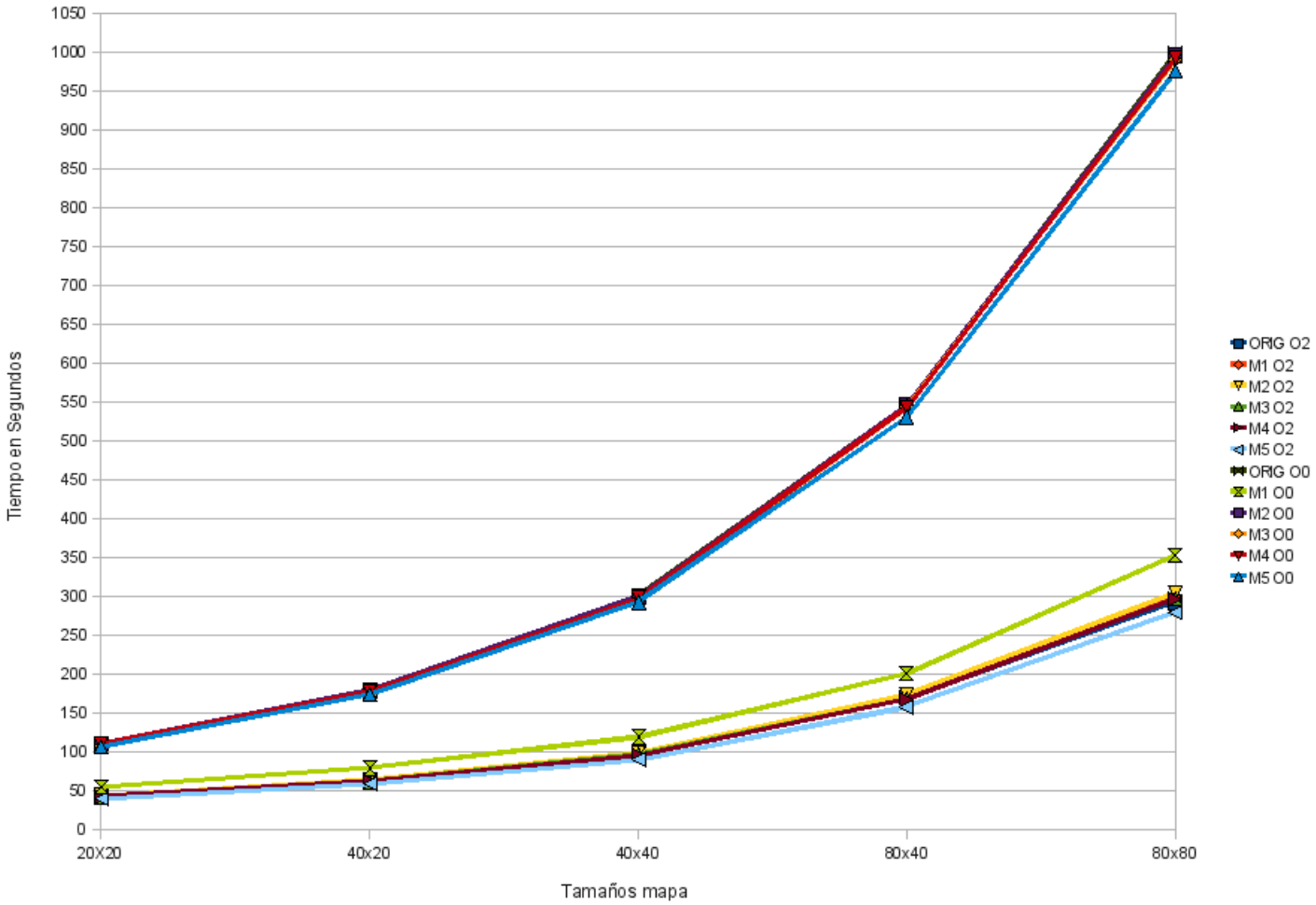

Figura 18. Gráfica de tiempos de todas las ejecuciones para diferentes tamaños.

grande, es de trece segundo con respecto al código original optimizado por el compilador.

Como consecuencia de los resultados obtenidos podemos decir que hemos conseguido uno de los objetivos que nos proponíamos: realizar una mejora del código original del que partíamos para ordenadores superescalares, con el fin de disponer del mejor código posible para su posterior paralelización.

8 Conclusión

A partir del objetivo marcado en este documento en el que se proponía una mejora del tiempo invertido por el algoritmo SOM, y tras aplicar técnicas de desarrollo eficientes, se pueden obtener las siguiente conclusiones que pasamos a enumerar:

1) Aunque se puede plantear seguir mejorando este código para los ordenadores superescalares, a la vista del coste que ha supuesto la generación de un código que combine con el compilador y obtenga tiempos menores, pensamos que para disminuir el tiempo de ejecución del algoritmo SOM es imprescindible explotar otros niveles de paralelismo del programa utilizando para ello arquitecturas multinucleo.
2) Se ha conseguido realizar una implementación del algoritmo en la que se ha mejorado el tiempo inicial del que partíamos. Así pues, se ha obtenido una mejora de más de un segundo en algunos casos. Teniendo en cuenta en esta comparativa las ejecuciones realizadas con o sin buffer.
3) A partir del código obtenido se pueden plantear mejoras en otros niveles de

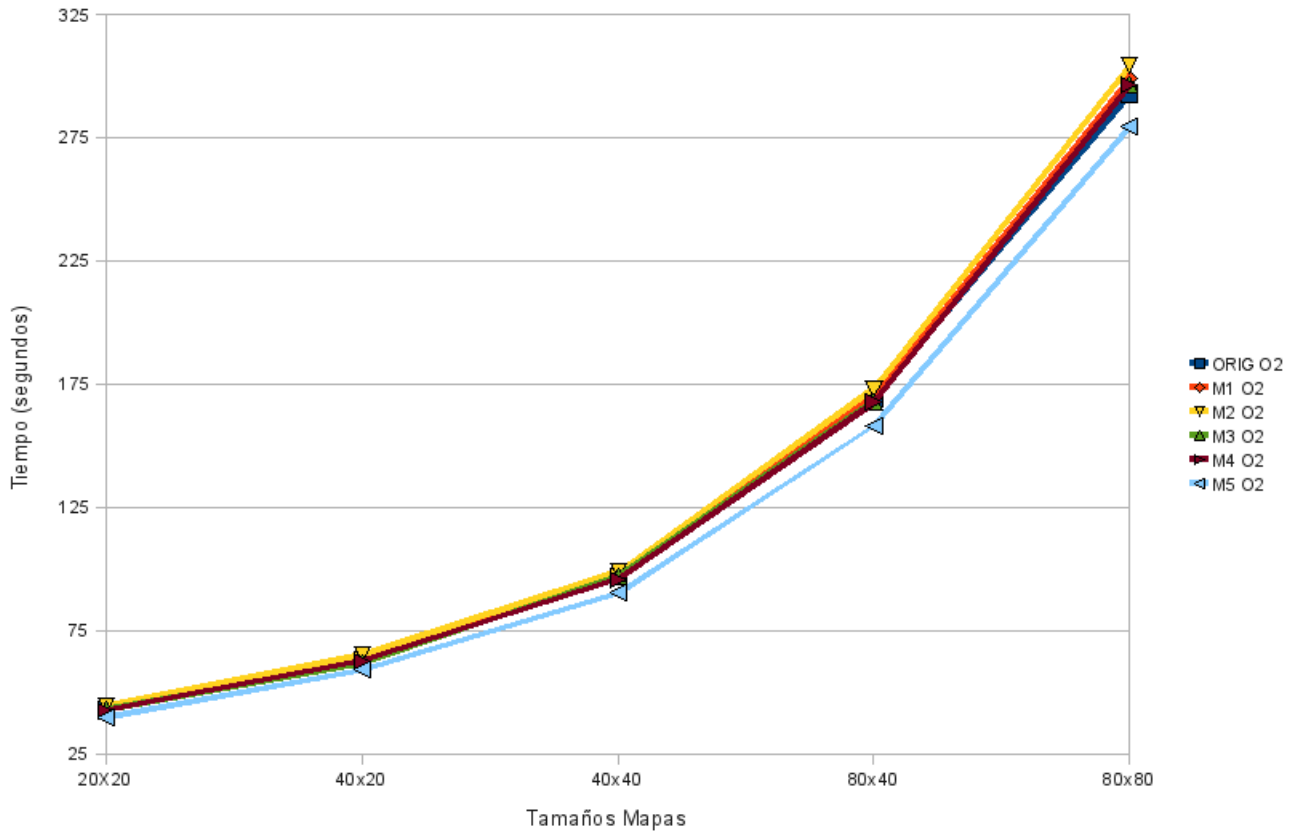

Figura 19. Gráficas de tiempos utilizando las optimizaciones del compilador para diferentes tamaños.

paralelización para seguir mejorando los tiempos invertidos en la ejecución del algoritmo.

4) Se puede deducir, a la vista de los resultados obtenidos, que existe una gran dificultad al intentar conseguir mejorar las optimizaciones realizadas por el compilador. Este detalle se deduce del hecho de que en ninguno de los casos estudiados hemos conseguido obtener tiempos inferiores a los que se obtenían usando las opciones de optimización del compilador. Así pues, la única expectativa de mejora que actualmente podemos esperar, es la de conseguir que nuestras mejoras se combinen con las del compilador, obteniendo código que se ejecute más rápido. Se puede apreciar este caso en la ejecución llevada a cabo con la última mejora y la utilización de un buffer para la lectura, en la cual conseguimos el mejor tiempo de todos los obtenidos (18.677 segundos).
5) Cómo se puede ver en este trabajo, el desarrollo eficiente es una técnica que puede aportar mejoras de tiempo, pero a la vista de los resultados obtenidos, las mejoras aportadas no son todo lo satisfactorias que se puede esperar, más aun cuando se disponen de herramientas como los compiladores actuales que aplican estas técnicas de forma automática, consiguiendo unos resultados superiores a las que pueda conseguir un programador.

Bibliografía

[1] C. Severance and K. Dowd, *High Performance Computing*, U. of Michigan, Ed. University of Michigan, August 25, 2010.

[2] D. Culler, J. P. Singh, and A. Gupta, *Parrallel Computer Architecture. A Hardware/Software Approach*, M. Kaufmann, Ed. Morgan Kaufmann, 1999.

[3] Top 500 supercomputer sites. [Online]. Available: http://www.top500.org/

[4] M. R. Sánchez, "Nuevos métodos para análisis visual de mapas auto-organizativos," 2004. [Online]. Available: http://oa.upm.es/251/1/tesis_Manuel_Rubio.pdf

[5] M. Anguita, A. Prieto, and J. Ortega, *Arquitectura de Computadores*, THOMSOM., Ed. THOMSOM., 2005.
[6] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach 4th Edition*, M. K. Publications, Ed. Morgan Kaufmann Publications, 2007.
[7] Tianhe-1 (th-1) supercomputer. [Online]. Available: http://www.nscc-tj.gov.cn/en/show.asp?id=191
[8] A. Fog, *Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms*, 2009.
[9] R. P. Garg and I. Sharapov, *Techniques for Optimizing Applications: High Performance Computing*, I. Sun Microsystems, Ed. Sun Microsisten., Junio 2001.
[10] F. C. Solorio, "Análisis de cúmulos(cluster)." [Online]. Available: http://lc.fie.umich.mx/ calderon/estadistica/cumulos.htm
[11] F. A. C. Romero, "Agrupamiento dinámico con lógica difusa." Master's thesis, Santiago de Chile, 2001.
[12] M. R. Anderberg, *Cluster Analysis for Applications.*, N. Y. Academic Press, Ed. Academic Press, New York, 1973.
[13] A. GORDON, *Classification. 2nd Edition.*, Chapman and Hall, Eds. Chapman and Hall, 1999.
[14] A. MARÍN and J. W. BRANCH B, "Aplicación de dos nuevos algoritmos para agrupar resultados de búsquedas en sistemas de catálogos públicos en línea (opac)." *Revista Interamericana de Bibliotecología.*, vol. 31, pp. 47–65, 2008.
[15] S. Bermejo, "K-means." [Online]. Available: http://petrus.upc.es/ microele/neuronal/xn/-docs/4_kmeans.pdf
[16] "Tecnical reports of som_pak." [Online]. Available: http://www.cis.hut.fi/research/papers/som_tr96.ps.Zres
[17] J. Fenlason and R. Stallman, *GNU gprof*, FSF. [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

En este proyecto se estudian y aplican técnicas de desarrollo eficiente sobre un tipo de algoritmos conocimos con el nombre de SOM (Self-organizing map) o mapas autoorganizativos de kohonen. Para ello se ha partido de una serie de herramientas realizada por la universidad de helsinki, que nos proporcionan programas para la inicialización, el entrenamiento y visualización de estos mapas. Esta implementación nos aporta un punto de partida para aplicar técnicas que nos permitan mejoras los tiempos de ejecución del algoritmo. En lugar de realizar una optimización de cada herramienta el estudio se ha centrado en la optimización del programa de entrenamiento del mapa. Se han realizado una serie de mejoras centradas en aprovechar el paralelismo a nivel de instrucción en ordenadores superescalares. Apoyándose en mediciones de una serie de datos reales usados para el entrenamiento de este tipo de algoritmos se analiza el grado de optimización conseguido con cada una de las mejoras realizadas.

In this project efficient development techniques for a type of algorithms, known as SOM (Self-organizing map) or autoorganizing Kohonen maps, are studied and applied. To this end, the work has been started with a number of tools, done by University of Helsinki, which provide us programs for the initialization, training and visualization of these maps. This implementation gives us an initial point to apply techniques allowing us to improve the execution times of the algorithm. Instead of realizing an optimization of each tool, the study is centered in the optimization of the training program of the map. A number of improvements have been realized concerning the exploitation of the parallelism of superscale computers at instruction levels. Based on the measurements of a number of real data used for the training of this type of algorithms, the optimization degree achieved with each realized improvement is analyzed.