Conceptos Fundamentales de Diseño en Sistemas de Programación Esqueletal

Fernando Saez Marcela Printista
LIDIC.
Universidad Nacional de San Luis.
Ejército de los Andes 950, San Luis, Argentina.
e-mail: {**bfsaez@unsl.edu.ar, mprinti@unsl.edu.ar**}

Abstract

In the last time the parallel programming community has worked to look for new templates or skeletons. A significant number of projects has built real systems, but none of these has achieved significant popularity neither developtment projects nor academic community. We reviewed some fundamentals concepts of design that should be supported by skeletal parallel programming systems to fortify their use and we present the development of a high level prototype adapts some of these essential concepts. This prototype resolves a high spectre of divide and conquer problems and their use is showed through three simple examples.

Keywords: Skeletal Programming System, Divide and Conquer, Design Concepts

Resumen

En los últimos tiempos, la comunidad de programación paralela ha trabajado mucho en soluciones basadas en patrones y esqueletos. Un número importante de proyectos han construido sistemas reales, pero ninguno de ellos ha alcanzado una popularidad notable en los entornos de desarrollo como tampoco en la comunidad científica paralela. El objetivo de este trabajo es revisar algunos conceptos fundamentales que deben ser tenidos en cuenta durante la etapa de diseño de patrones y que fortalecen el uso de un sistema de programación paralela esqueletal. Además, en este trabajo se presenta el desarrollo de un prototipo de alto nivel, el cual incorpora alguno de estos conceptos esenciales. Este prototipo resuelve un amplio espectro de problemas Divide y Vencerás y su uso es mostrado a través de tres ejemplos muy sencillos.

Palabras claves: Sistema de Programación Esqueletal, Divide y Vencerás, Conceptos de Diseño

1 INTRODUCCIÓN

Muchos algoritmos paralelos pueden ser caracterizados y clasificados por su adhesión a uno o más patrones de computación. La programación esqueletal propone que tales patrones sean abstraídos y provistos al programador como un conjunto de herramientas formando así un sistema de programación esqueletal. Este sistema de programación permite resolver un problema concreto a partir de la creación de instancias de métodos más generales. El sistema esta compuesto por un conjunto de

esqueletos (paradigmas) desde los cuales pueden ser derivados programas modelos que ilustren como el paradigma resuelve problemas específicos [5].

El objetivo del sistema esqueletal es reducir las dificultades del desarrollo de software por medio de la abstracción y la reusabilidad de programas paralelos y mejorar la performance dando acceso a los esqueletos optimizados en arquitecturas particulares.

A partir de la experiencia obtenida en la programación esqueletal, especialmente en el diseño de distintos esqueletos Divide y Vencerás desarrollados con propósitos específicos [7, 8, 9] y en base a lo presentado en [2], se ha podido interpretar conceptos fundamentales de diseño que deben ser soportados por un sistema de programación esqueletal. La aplicación de estos conceptos ha conducido al desarrollo de un primitivo de alto nivel cuyo objetivo es capturar estas características para que resulte altamente abstracto y funcional.

El objetivo de este trabajo es revisar y poner a discusión estos conceptos de diseño, mostrar el primitivo $D\&V$ desarrollado en forma de librería de funciones C soportadas en MPI [1] y poner en contexto la implementación del esqueleto $D\&V$ a través de ejemplos muy sencillos.

En la sección 2 se introduce una terminología clara y sencilla que permite definir cada uno de los conceptos, en la sección 3 se describe el esqueleto $D\&V$ implementado y se muestra como éste puede ser usado para resolver distintos tipos de problemas $D\&V$. Por último en la sección 4 se presentan las conclusiones y los trabajos futuros.

2 CONCEPTOS Y TERMINOLOGÍA

Con el propósito de formalizar las propiedades fundamentales de diseño que debe satisfacer un sistema de programación esqueletal, este trabajo adopta la terminología expuesta por Cole [2], la cual es clara y simplifica la comprensión de los conceptos.

El propósito de cualquier esqueleto es abstraer un patrón de *actividades* y de sus *interacciones*.

Entre los aspectos a ser evaluados cuando se identifican las actividades podemos identificar: el tamaño del gránulo, el grado de reusabilidad del código que implementa la actividad y el mapeo de las actividades a la arquitectura, entre otros.

En cuanto a las interacciones, es importante distinguir las interacciones, denominadas *internas*, que ocurren entre dos o más actividades. Estas son las encargadas de capturar el paradigma de computación paralela y la topología del grupo de procesos intervinientes. En una interacción interna, las actividades pueden estar asignadas a un mismo procesador o residir en distintos procesadores.

Por otro lado, las interacciones *externas* ocurren entre las actividades y el contexto que las contiene. Estas últimas serán las encargadas de la entrada-salida del primitivo. Por ejemplo, en una aplicación de procesamiento de imágenes debería ser posible administrar, de manera explícita, que distintos segmentos de una misma imagen que se encuentran almacenados en varios procesos conforman una única imagen.

Existe una clasificación que involucra el modo en cómo interactúa un esqueleto con su contexto y que está determinado por el origen y destino de la entrada y salida del esqueleto:

Centralizado-Centralizado (CC): es aquél cuyo conjunto de datos de entrada y de resultados se encuentran en un único proceso de todos los disponibles para la ejecución del esqueleto.

Distribuido-Distribuido (DD): es aquél cuyo conjunto de datos de entrada y de salida se encuentran distribuidos entre los distintos procesos.

Replicado-Replicado (RR): es aquél cuyo conjunto total de datos de entrada y de salida se encuentran replicados en cada uno de los procesos.

Las restantes combinaciones (CR, CD, DC, DR, RC, RD) pueden ser fácilmente verificadas.

Un problema de procesamiento de imágenes resuelto con el paradigma de programación pipeline permite ejemplificar la categoría CD. Suponga un problema cuyo objetivo es remover frecuencias específicas ($f_0, f_1, ..., f_{p-1}$) de una imagen digitalizada. Durante el procesamiento de imágenes es importante recuperar las imágenes intermedias que van surgiendo luego de la eliminación de cada f_i. Este problema puede ser resuelto por un pipeline CD de p etapas donde cada etapa del pipeline recibe una imagen ($DigI_i$), le aplica un filtro fil_i, obtiene una nueva imagen $DigI_{i+1}$ a la cual se le han eliminado las frecuencias $f_0, ..., f_i$, la almacena en un buffer de salida del proceso y la transmite a la siguiente etapa.

Otro ejemplo se puede observar en un pipeline que resuelve problemas denominados $All-Pairs$. Una computación $All-Pairs$ sobre un sistema de n elementos puede ser paralelizado por medio de un pipeline DD. Cada etapa contribuye con un subsistema de elementos y en ella se realiza una computación $All-Pairs$ sobre los elementos retenidos. Luego cada etapa recibe elementos de su vecino izquierdo, los hace interactuar con los elementos retenidos, los almacena en un buffer y los envía a su vecino derecho. Todos los elementos quedan igualmente distribuidos entre las etapas del pipeline. Una aplicación de este tipo de pipeline puede verse en [8].

Como se mencionó antes, un esqueleto debe proveer una abstracción de las interacciones entre las distintas actividades que conforman un programa paralelo, pero trae aparejado un proceso de estructuración que restringe el modo en que las actividades pueden interactuar. Existen dos tipos de restricciones:

- *Espaciales*: determinan que actividades pueden interactuar entre si (actividades socias) y especifican el rol (fuente y/o destino) de cada una en la interacción.
- *Temporales*: determinan el orden temporal correcto de interacción entre actividades socias.

Por ejemplo, un pipeline clásico es definido por una restricción espacial especificando que las interacciones se producirán entre actividades asignadas a dos etapas sucesivas. En otras palabras:

- Las actividades A_i y A_{i+1} son socias, para $0 \leq i < p-1$ y la interacción se producirá desde A_i hacia A_{i+1} (interacción: $I_{i,i+1}$).
- Las actividades A_i y A_{i-1} son socias, para $0 < i < p$ y la interacción se producirá desde A_{i-1} hacia A_i (interacción: $I_{i-1,i}$).

Una restricción temporal para este mismo ejemplo, debe determinar que durante el mismo ciclo de ejecución, toda etapa se comunique, en primer lugar, con la etapa anterior del pipeline y luego con la etapa posterior. En otras palabras, durante un mismo ciclo de ejecución, una actividad A_i deberá ejecutar la interacción $I_{i-1,i}$ y luego la interacción $I_{i,i+1}$, en ese orden.

En la próxima sección se describen algunos de los conceptos fundamentales que debe soportar un sistema de programación esqueletal para alcanzar un mayor grado de flexibilidad y permitir resolver una mayor diversidad de problemas.

2.1 Anidamiento

Informalmente el anidamiento de esqueletos es un término general usado para denotar una situación en donde dos o más esqueletos paralelos están activos en el mismo tiempo. El anidamiento de esqueletos es una propiedad importante que debe satisfacer un sistema esqueletal para poder combinar e integrar las distintas facilidades formando así estructuras no convencionales que permitan resolver algoritmos más complejos. También permite que el sistema simplifique la expresión de algoritmos

paralelos que operan sobre estructuras de datos irregulares, ya que permite expresiones directas de conceptos como “En paralelo, por cada vértice en un grafo, buscar su vecino mínimo” o “En paralelo, por cada fila en una matriz, sumar sus elementos”. En ambos casos las acciones internas (Buscar el vecino mínimo o sumar la fila) también pueden ser realizadas en paralelo.

Es posible identificar dos modos de anidamientos:

- *Transitorio*: una actividad podría invocar a otro esqueleto para procesar algún dato local o para realizar otra computación autónoma. Aquí se mantienen las interacciones externas entre el esqueleto anidado y la actividad invocante.
- *Persistente*: desde la perspectiva del esqueleto anidado, sus interacciones externas se convierten en interacciones internas entre la actividad invocante del esqueleto de nivel superior y el esqueleto anidado.

En el ejemplo del pipeline, una etapa formada por varios procesos podría recibir un dato, particionarlo en una colección de subdatos, procesarlos independientemente en otro pipeline interno y finalmente construir el resultado a entregar al nivel superior. Cada invocación del pipeline anidado es iniciada por una actividad del nivel superior.

En una actividad, bajo un anidamiento transitorio, la interacción externa se produce con una invocación a una función convencional (llamada al esqueleto anidado). En esta invocación, los datos de entrada deben ser explícitamente definidos por el programador. Al finalizar la ejecución del pipeline anidado, los datos de salida deben ser manipulados para construir la solución que se devolverá a la actividad invocante.

Por otro lado, la misma etapa del pipeline anterior podría invocar un pipeline persistente como su nivel interno, aquí cada dato requerido por la primera etapa del pipeline anidado interactúa internamente con la etapa anterior del pipeline de nivel superior. La invocación persistente del pipeline indica que los datos navegarán (interactuarán internamente) desde el esqueleto padre hacia el esqueleto hijo y luego nuevamente del hijo al esqueleto padre, por lo tanto, en este caso, no es necesario definir su entrada y salida.

2.2 Modo de Interacción

La estructura de un esqueleto especifica las actividades que lo conforman y las interacciones permitidas entre estas actividades. Siguiendo la terminología utilizada en [2], toda actividad realiza interacciones *implícitas* las cuales estan restringidas temporalmente por el esqueleto. Pero la estructura de un esqueleto también debería ser flexible y permitir que se produzcan interacciones no contempladas por la estructura y de esa manera atender necesidades excepcionales de la aplicación. El modo de interacción es el mecanismo que le permitirá al programador indicar la ocurrencia de interacciones *explícitas*. En este modo, las actividades realizan interacciones definidas específicamente en el código de una actividad. Este último modo debe ser manejado por el sistema de una forma cuidadosa, ya que si bien es ofrecido como una facilidad del esqueleto, no debe afectar la programabilidad ni romper la abstracción del patrón.

Suponga en el ejemplo del pipeline que procesa una secuencia de imágenes, que alguna etapa necesita descartar una cierta imagen de la secuencia. Para este caso se necesitarán acciones explícitas que indiquen cuando una actividad no debe dejar pasar la imagen a la etapa posterior. En este caso las actividades del pipeline tienen modo de interacción *explícito*.

3 SISTEMA ESQUELETAL: IMPLEMENTACIÓN

La necesidad de un sistema esqueletal que soporte la integración (anidamiento) y composición de los esqueletos es vital para componer programas paralelos más complejos que puedan ser expresados como una colección de actividades interactuantes. Aún con esta integración es casi imposible pensar que pueda resolver todos los problemas paralelos en forma eficiente,

es necesario un mecanismo que permita integrar estas construcciones de alto nivel con operaciones de comunicación y sincronización simples (Ej. Send, Receive, Barrier) y colectivas (Broadcast, Gather, Reduce). Para soportar estos mecanismos y evitar la introducción de nueva sintaxis, el sistema de programación esqueletal esta implementado como una librería de C, y hace uso de MPI para implementar los aspectos de comunicaciones. La librería MPI es altamente conocida por los programadores paralelos, y provee una excelente portabilidad.

Las características que nombramos en la sección anterior han sido incorporadas al sistema de programación esqueletal, permitiendo al usuario realizar una mejor configuración del esqueleto para resolver su programa paralelo. En una primera instancia de desarrollo se ha implementado el paradigma Divide y Vencerás.

3.1 Esqueleto Divide y Vencerás

El diseño y construcción del esqueleto Divide y Vencerás presentado, tiene sus sustentos en [8], y se ha modificado para soportar las características discutidas y otros rasgos que agregan mayor flexibilidad.

El esqueleto soporta un modo de anidamiento transitorio a través de invocaciones a otros esqueletos en cualquier momento de la ejecución del código interno de una actividad. Por ejemplo la actividad de división podría utilizar un esqueleto con interacciones centralizadas-centralizadas que lo resuelva. La opción de anidamiento persistente no es soportada aún por el esqueleto. Como ejemplos de sistemas que soportan ambos modos de anidamiento podemos nombrar: Skel [6], Kuchen's Skeleton Library [11] y Eden [3].

El modo de interacción es asociado con cada actividad en el esqueleto, instanciando el parámetro correspondiente en la llamada al mismo. Al contrario de otros esqueletos, donde debe definirse el modo de interacción por cada actividad, para el esqueleto $D\&V$ solo es necesario definir si este permitirá que las interacciones puedan realizarse en forma explícita. Para soportar las interacciones explicitas, el sistema provee una función *Interactuar()* que podrá ser invocada en cualquiera de las actividades (generalmente en las actividad de división y combinación).

Cuando el esqueleto es configurado en modo explícito su ejecución se asemeja a un código secuencial, en donde no existe comunicación entre procesos, la llamada a la función *Interactuar()* permitirá que las actividades socias interactúen en modo sincrónico. En este caso, el usuario del esqueleto debe llevar el control de la sincronización entre las actividades.

La definición del esqueleto es la siguiente:

```
void DC_Call(tipoDC Type,int Weight,mInteraction MI,TPF_trivial Itrivial,TPF_conquer Iconquer,
             TPF_divide Idivide,TPF_combine Icombine,TPF_secuencial Isecuencial,
             TypeN *In,int SizeBufferIn,int SizeDataTypeIn,TypeN *Out,int SizeBufferOut,
             int SizeDataTypeOut,MPI_Comm comm)
```

El prototipo del esqueleto $D\&V$ permite configurar un conjunto de parámetros, no sólo con el objetivo de proveer versatilidad en su aplicación a distintos problemas, sino también para que sea capaz de representar algunos de los conceptos esenciales mencionados anteriormente:

· **tipoDC Type**

Su valor determina la versión del algoritmo que se desea utilizar y la cual puede depender del tipo de problema específico a resolver. Es un tipo enumerado definido como:

```
typedef enum {DCC,DCH,DCED,DCCT) tipoDC;
```

1. DCC, Divide y Vencerás Clásico: en este algoritmo un sólo procesador dispone de los datos de entrada (entrada centralizada). Cuando el procesador divide el problema, comunicará parte de los subproblemas a procesadores libres y seguirá trabajando con el subproblema restante.

2. DCH, Divide y Vencerás Hipercúbico: en este tipo de algoritmo, el esqueleto mantiene una estructura recursiva y genera un árbol binario de grupos de procesos, cuyas hojas contienen un único proceso. En cada recursión, el grupo de procesos es dividido en dos subgrupos donde cada uno soluciona un subproblema. Esta configuración es ideal para algoritmos $D\&V$ binarios con interacciones RR.

3. $DCED$, Divide y Vencerás Completamente Paralelo (Divide and Conquer with Embarrassing Divisibility): este algoritmo pertenece a la clase de $D\&V$ en el cual no es necesario (o es mínima) la comunicación. En la práctica estos algoritmos son aquellos en el cual el problema puede ser tratado inmediatamente como dos o más subproblemas; en estos algoritmos no es necesario movimientos de datos extras en la etapa de división. Ejemplos de este tipo de algoritmo son el producto escalar y la multiplicación de matrices balanceadas.

4. $DCCT$, Divide y Vencerás con Combinación Trivial: estos algoritmos son aquellos en los que una vez que el problema es dividido a su máxima expresión, el mismo queda resuelto y su solución queda distribuida en todos los procesos de último nivel. En estos algoritmos la fase de división, no sólo divide los datos, sino que los modifica en busca del resultado. Como ejemplo de este tipo de algoritmo se puede mencionar el mergesort.

· **int Weight**

Este parámetro es también conocido como factor de ramificación (branching factor). Es el número de actividades (subproblemas) que se generarán en la fase de división. En cualquier caso $Weight$ debe ser mayor que 1. Para el tipo DCH, $Weight$ debe ser 2. En otro caso $Weight$ debe ser escogido teniendo en cuenta las características del problema y los procesadores disponibles.

Problemas como el quicksort tienen un factor de ramificación de 2, por más que el problema inicial sea dividido en 3 listas, sólo dos de ellas constituyen los subproblemas a resolver. Algunos algoritmos que trabajan sobre un espacio bidimensional (como el algoritmo $n-body$ sobre un plano) tienen un factor de ramificación igual a 4. La multiplicación de matrices de Strassen [12] tiene un factor de ramificación igual a 7.

En los casos en los cuales el factor de ramificación no es naturalmente mapeado sobre una arquitectura, alguna técnica de agrupación de procesos o balance de carga es necesaria. El esqueleto soporta un factor de ramificación constante. Una vez definido, es imposible cambiar este valor dinámicamente.

· **mInteraction MI**

Es un tipo enumerado definido como:

```
typedef enum {IMPL,EXPL) mInteraction;
```

Cada valor especifica el modo de interacción que se desea utilizar. El valor $IMPL$ indica que el esqueleto trabaja en modo implícito. En caso de configurarse en modo $EXPL$, las interacciones entre las actividades serán responsabilidad del programador.

· **TPF_trivial Itrivial**

Este parámetro es el puntero a la función $trivial()$ que indica sobre su retorno si se ha alcanzado el punto final de la recursión en donde no es posible seguir dividiendo los subproblemas. En tal caso, el esqueleto llama automáticamente a la función $conquer()$. Su prototipo es:

```
int trivial(TypeN *Input);
```

· **TPF_conquer Iconquer**

Apunta a la actividad encargada de resolver el caso base (trivial). El resultado debe ser almacenado en el parámetro $Output$. Su definición es la siguiente:

```
void conquer(TypeN *Input,TypeN *Output)
```

· **TPF_divide Idivide**

Representa el puntero a la actividad de división. Esta actividad divide los datos de entrada ($Input$) y genera el conjunto de subdatos a resolver. Es responsabilidad de la actividad de división generar una entrada (sub-problema) para cada ramificación, y asignar el espacio necesario. La estructura $IntraN$ contiene tantas entradas como número de ramificaciones se generen ($Weight$). El prototipo de la función es:

```
void divide(TypeN *Input, IntraN InputIntra)
```

· **TPF_combine Icombine**

Es la función encargada de combinar las soluciones parciales que se encuentran en $OutputIntra$, y de generar la solución general. Es responsabilidad de esta función asignar el resultado a $Output$. El prototipo de la función es:

```
void combine(TypeN *Output,IntraN *OutputIntra);
```

· **TPF_secuencial Isecuencial**

Es el puntero al algoritmo secuencial que (mejor) resuelve el problema en una arquitectura secuencial (arquitectura monoprocesador). El prototipo de la función es:

```
void secuencial(TypeN *Input,TypeN *Output)
```

· **TypeN *Input, int SizeBufferInput, int SizeDataTypeInput**

Este parámetro indica la estructura de entrada al esqueleto. La estructura $TypeN$ esta definida como un buffer genérico que puede representar cualquier tipo de dato elemental o definido por el usuario. El esqueleto necesita conocer el tamaño de su tipo de dato elemental ($SizeDataTypeInput$), y la cantidad de elementos ($SizeBufferInput$) requeridos. A continuación se muestra la definición de la estructura $TypeN$:

```
typedef struct
{
void * _Buffer;
long _Size;
}TypeN;
```

Para mayor facilidad en el uso de esta estructura se han implementado algunas de las operaciones más requeridas sobre ella. La función $DC\_Alloc()$ permite asignar espacio de memoria. Si $DC\_Alloc()$ no puede asignar la memoria requerida, asigna a $buff$ el valor $NULL$. El prototipo de la función es:

```
void DC_Alloc(TypeN *buff,int LongBuffer, int SizeDataType);
```

La función $DC\_Copy()$ permite copiar un buffer de datos. Antes de la invocación a $DC\_Copy()$, el usuario debe haber realizado la asignación pertinente sobre la estructura dst. En caso de no realizar la copia, asigna a dst el valor $NULL$. El prototito de la función es el siguiente:

```
void DC_Copy(TypeN *dst,TypeN *src);
```

· **TypeN *Output,int SizeBufferOutput, int SizeDataTypeOutput**

Estos tres últimos parámetros le indican al esqueleto el buffer de salida. Esta estructura también debe ser provista de un espacio de memoria asignada antes de llamar al esqueleto. El tamaño del tipo de dato elemental $SizeDataTypeOutput$ y la cantidad de elementos del buffer de salida $SizeBufferOutput$ deben ser especificados. No necesariamente un problema $D\&V$ debe tener el mismo tipo de dato elemental para el problema y la solución, ni tampoco el mismo tamaño de datos de entrada que de salida.

· **MPI_Comm comm**

Representa al comunicador de MPI. Permite al esqueleto delimitar cual es el grupo de procesadores involucrados en una comunicación, así también como la integración del esqueleto con código paralelo (Ad-Hoc) necesario para resolver un problema irregular que no se ajusta a ningún patrón soportado por el sistema.

3.2 Ejemplos

Para mostrar como el esqueleto resuelve distintos tipos de problemas Divide y Vencerás se han elegido ejemplos simples (”toys examples”). La idea de mostrar estos ejemplos es evidenciar como el esqueleto debidamente configurado, nos ayuda a resolver de forma práctica muchos problemas de la clase $D\&V$.

Reducción de Vectores El algoritmo de reducción de vectores es un ejemplo clásico que consiste en aplicar una operación asociativa y conmutativa a los valores correspondientes a cada una de las componentes de los vectores de entrada y de esa manera obtener un único vector resultado. Por ejemplo, dados dos vectores $A = (2, 4, 3)$ y $B = (6, 3, 5)$ y la operación suma $(+)$, el vector reducción es: $R = A + B = (2+6, 4+3, 3+5) = (8, 7, 8)$. Una solución eficiente usa el esqueleto configurado con los siguientes valores:

```
void DC_Call(DCED, 2,IMPL, &trivialRV, &conquerRV, NULL, &combineRV, &secuencialRV,
             &VectorInput,N, sizeof(vector), &VectorOutput, 1, sizeof(vector), Comm)
```

El primer parámetro indica que el esqueleto debe resolver un tipo de $D\&V$ completamente paralelo. Este tipo de $D\&V$ no necesita de una función de división. El esqueleto divide el conjunto de datos por el factor de ramificación hasta alcanzar el punto de parada ($\&trivialRV$), luego comienza su fase de conquista ($\&conquerRV$) y reducción ($\&combineRV$).

Otra configuración del esqueleto podría usar interacción explícita entre las actividades:

```
void DC_Call(DCED,2,EXPL,&trivialRV, &conquerRV, NULL, &combineRV, &secuencialRV,
        &VectorInput, N, sizeof(vector), &VectorOutput, 1, sizeof(vector), Comm)
```

En este caso el usuario debe realizar las comunicaciones necesarias de forma explícita en la actividad de combinación. Para realizar estas comunicaciones, el programador debe hacer uso de las funciones dispuestas por el sistema. Para el esqueleto Divide y Vencerás el sistema dispone de la función $Interactuar()$. La función se encarga de intercambiar el contenido de los bufferes de actividades socias.

```
void combine(TypeN *Output,IntraN OutputPartner)
{
 Interactuar(Output, OutputPartner, sizeof(Datatype))
 /* tareas de la combinación */
}
```

Quicksort Quicksort es un algoritmo Divide y Vencerás que ordena una secuencia de números, dividiéndola recursivamente en sub-secuencias más pequeñas [4]. El ordenamiento de las secuencias mas pequeñas representa dos subproblemas completamente independientes que pueden ser solucionados en paralelo.

```
DC_Call(DCCT, 2, IMPL, NULL, NULL, &divideqs, NULL, &secuencialqs, &VectorInput,
        N, sizeof(int), &VectorOutput, N, sizeof(int), comm)
```

Para la correcta instanciación del problema, se instancia el tipo $D\&V$ con combinación trivial. En este caso el esqueleto hace caso omiso de la función trivial, conquer y combine. La implementación del esqueleto resuelve el caso simple al llegar a un tamaño de entrada de datos de 1 elemento y luego se encarga de ir comunicando las soluciones parciales en el árbol y dejar la solución en $VectorOutput$.

FFT La Transformada Rápida de Fourier (FFT de Fourier Fast Transform) es uno de los métodos númericos más altamente usado en las ciencias e ingenierías, especialmente en el area de procesamiento de imagenes y señales, análisis espectrales, comunicación, teléfonos celulares y hasta sistemas de control digital. Existen varios algoritmos secuenciales que resuelven la FFT, uno de los primeros fue introducido por Cooley y Tukey (1966), ver [10] para más detalles.

```
DC_Call(DCH,2,IMPL,&Trivialfft,&conquerfft,&dividefft,&combinefft,&secuencialfft,&VectorInput,
        N, sizeof(Complex),&VectorOutput,N,sizeof(complex),comm);
```

Este ejemplo se adapta perfectamente al paradigma $D\&V$ con comunicaciones hipercúbicas. Por otro lado, esta aplicación permite testear el esqueleto en su máxima expresión y mostrar como funcionan todas sus componentes. La actividad $dividefft$ se encarga de dividir la secuencia de entrada ($VectorInput$) en los componentes pares e impares hasta llegar a secuencias de 1 elemento, donde la FFT de un elemento es el mismo elemento ($trivialfft$ y $conquerfft$). Por último $combinefft$ realiza las computaciones necesarias y combina las secuencias transformadas resultantes en $VectorOutput$.

4 CONCLUSIONES Y TRABAJO FUTURO

En este trabajo, se revisaron algunos conceptos fundamentales de diseño que deben soportar los sistemas esqueletales (conjunto de constructores de alto nivel) si se desea obtener un entorno de programación esqueletal flexible que ayude al programador paralelo a resolver una amplia gama de problemas. También se ha introducido y utilizado una terminología fácil de comprender y a la vez ágil para definir estos conceptos y establecer su relación con un sistema de programación esqueletal. El soporte de anidamiento es esencial en la programación estructurada para poder combinar e integrar los distintos esqueletos y encarar soluciones a problemas irregulares y complejos. Un sistema esqueletal que soporte ambos tipos de anidamiento ofrece al usuario mayor flexibilidad al momento de diseñar una solución. Si bien, en la actualidad, el sistema implementado soporta únicamente el anidamiento transitorio, se ha podido observar como esta caracterítica es útil para representar problemas variados. El modo de interacción es otro aspecto que debe tenerse en cuenta si lo que se requiere es flexibilidad. El modo de interacción explícito elimina las restricciones temporales entre las actividades, pero incrementa la responsabilidad del usuario. Nuestro sistema soporta ambos modos de interacción e introduce una función de comunicación colectiva definida para el esqueleto divide y vencerás.

Si bien el esqueleto puede resultar complejo por su número de parámetros, se ha mostrado, por medio de la instanciación de distintos ejemplos clásicos, que la interfase es conceptualmente manejable y que su correcta configuración facilita la resolución de problemas típicos pertenecientes a la clase de problemas, en este caso, $D\&V$.

En la actualidad se esta trabajando en la generación de otros esqueletos, en el soporte de características avanzadas (balance de carga, anidamiento persistente, etc), la portabilidad y otimización de esqueletos en diferentes arquitecturas (incluidos Multiclusters y Grid). Todos estos temas son relevates si queremos que un sistema de programación esqueletal alcance los objetivos de programabilidad, portabilidad y performance, y demuestre obtener beneficios con su uso.

AGRADECIMIENTOS

Los autores deseamos agradecer a la Universidad Nacional de San Luis, la ANPCYT y el CONICET por su contínuo soporte en el desarrollo de nuestras investigaciones.

BIBLIOGRAFÍA

[1] Message passing interface forum. available at http://www.mpi-forum.org/docs/docs.html.

[2] M. I. Cole. A. Benoit. *Two Fundamental Concepts in Skelletal Parallel Programming.* ICCS 2005, LNCS 3515, 2005.

[3] Loogen R. Ortega-Mall´en Y. PeÜna R. Breitinger, S. *Eden: Language Definition and Operational Semantics.* Technical Report 10, Philipps-University of Marburg, 1996.

[4] Hoare C.A.R. *Algorithm 64: Quicksort*. Communications of the ACM, 4, p. 321, 1961.

[5] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing.* Pitman, London, UK., 1989.

[6] M. I. Cole. *eSkel: The edinburgh Skeleton library Version 2.0*. Draft API reference manual. Internal Paper, School of Informatics, University of Edinburgh, 2003.

[7] C. Rodriguez León F. Piccoli, M. Printista. *Dynamic Hypercubic Parallel Computations*. Proceeding (466) Parallel and Distributed Computing and Systems, 2005.

[8] M. Printista F.D. Saez, R. Gallard. *Paradigms of Parallel Programming*. Workshop de Investigadores en Ciencias de la Computación (WICC 2003), Tandil, Argentina, May 2003.

[9] M. Printista J.G. Zanabria, F. Piccoli. *Hypercubic Comunications in MPI*. Tesis submited for UNSL, 2005.

[10] Cooley J.W. and Tukey J.W. *An algorithm for machine calculation of complex Fourier series*. 1966.

[11] H. Kuchen. *A skeleton library*. Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer-Verlag Pag. 620-629, 2002.

[12] Strassen V. *Gaussian elimination is not optimal*. Numerische Mathematik., 1969.