HERRAMIENTAS PARA PROGRAMACION PARALELA

José Manuel García Carrasco

Departamento de Informática
Universidad de Castilla-La Mancha

1. CONCEPTOS PREVIOS.

Gracias a las progresivas mejoras tecnológicas desarrolladas durante las décadas pasadas, la capacidad de procesamiento de los ordenadores ha aumentado cada vez más. De todas formas, estas mejoras tecnológicas -integación VLSI, aumento de la frecuencia de reloj en los procesadores, etc- tienen un límite físico, por lo cual el aumento de capacidad de procesamiento no es ilimitado. Hay una serie de problemas que no se pueden resolver con este tipo de ordenadores: la circulación atmosférica y el tiempo, la evolución de las galaxias, problemas de sistemas expertos e inteligencia artificial, etc. Para la resolución de estos problemas se está investigando intensamente desde hace dos décadas en el procesamiento paralelo. Hoy en día ya tenemos ordenadores comerciales que trabajan en paralelo, y esta parece ser la principal línea de investigación para mejorar la potencia de los ordenadores. De hecho, esta técnica ya ha llegado a los ordenadores personales gracias a la posibilidad de añadir placas que contienen varios procesadores más. Esta línea de investigación ha dado lugar a los multiprocesadores y los multicomputadores. Sin embargo, que un ordenador tenga una mayor capacidad de procesamiento no se consigue simplemente colocando un número mayor de procesadores, sino que supone una gran complejidad, tanto en el hardware como en el software. Vamos a describir brevemente esta complejidad, centrándonos en los aspectos de programación.

En primer lugar, no todo problema puede ser paralelizado. Hay problemas que son inherentemente secuenciales y por lo tanto es difícil conseguir su ejecución en paralelo. De todas formas, la mayoría de los problemas científicos descritos anteriormente se pueden describir fácilmente en paralelo.

En segundo lugar, necesitamos una arquitectura paralela que permita ejecutar una determinada aplicación, pero también un lenguaje adecuado que permita expresar el paralelismo del problema. Este punto ha dado lugar a diversas líneas de trabajo e investigación que vamos a describir a continuación:

a) **Verificación de programas paralelos**. Un programa en paralelo no tiene nada que ver con un programa secuencial, pues su comportamiento es muy diferente. Las técnicas de verificación para programas secuenciales no sirven en el caso paralelo, por lo que se tiene que desarrollar toda la teoría formal de verificación de programas. Se puede citar los trabajos de [Kel76] y [Owi76] como dos trabajos pioneros que explican a fondo el problema de la verificación. Hay que tener en cuenta que en el caso paralelo nos aparecen problemas tales como el *deadlock, livelock, race conditions*, el *no determinismo,* etc., que complican abundantemente la verificación formal de un programa.

b) **Abstracción del hardware de la máquina.** Mientras está claro que los lenguajes secuenciales son independientes de la máquina en que se ejecutan, no se puede decir lo mismo de los lenguajes paralelos. Es fácil que una independencia de la máquina conduzca a una pérdida de potencia, por lo que habitualmente se encuentran ligados en mayor o menor medida a una arquitectura determinada. Un ejemplo de unión estrecha lo tenemos con el lenguaje Occam [Inm88], cuyo desarrollo está vinculado al del transputer [Inm89]. Lo habitual es conseguir un grado de abstracción media, es decir, abstraer los detalles de más bajo nivel (tipo de conexión entre los procesadores, potencia de cada procesador, mecanismo de acceso a memoria por los procesadores, etc.), para dejar sólo la referencia al modelo de procesamiento empleado (sistólicos, SIMD, MIMD, etc.).

c) **Lenguajes síncronos o asíncronos.** La mayoría de los lenguajes de programación desarrollados son asíncronos, es decir, son no deterministas y están basados en las nociones de procesos y comunicaciones entre procesos. Ejemplos de lenguajes de este tipo son el ADA, MODULA, Occam, etc. y las extensiones paralelas para Fortran, C, etc. Pero también se han desarrollado lenguajes síncronos, para aquellos problemas de tiempo real deterministas, tales como interfaces para teclado o ratón, video juegos, problemas de regulación automática, etc. Un ejemplo de este tipo de lenguaje es el ESTEREL [Ber84], que fue el primero que se desarrolló.

d) **Paralelización automática.** El problema que tiene el desarrollo de nuevos lenguajes de programación es que todo el software desarrollado hasta la fecha se queda obsoleto. Para evitar esto, una de las líneas de investigación más potenciadas en los últimos años es la paralelización automática. La idea es que sea el compilador el encargado de determinar qué zonas del programa se van a ejecutar en paralelo, a partir de un programa escrito en un lenguaje secuencial. Los primeros desarrollos en esta línea fueron para los casos de paralelismo de datos y en particular para operaciones que trataran con vectores, por lo que se denominó a dichos compiladores **vectorizadores.** Generalmente, el lenguaje secuencial mas empleado en este grupo es el Fortran, quizás porque la mayoría de los supercomputadores utilizan este lenguaje. Un *vectorizador* Fortran analiza la secuencia de instrucciones para una posible traducción a instrucciones vectoriales. Logicamente, es extremadamente *dependiente* de la máquina *objeto*, puesto que debe tener en cuenta las especiales características de su hardware. Los principales análisis que realiza un vectorizador son: determinación del flujo de procesamiento entre subprogramas, chequeo de las relaciones de dependencias entre los subprogramas, análisis de las variables y reemplazo de los bucles internos por instrucciones vectoriales. Un estudio de muchas de estas técnicas puede encontrarse en [Pad86]. Para máquinas de diseño MIMD esta técnica ha dado buenos resultados en sistemas de memoria compartida (multiprocesadores), habiendo poco trabajo aún desarrollado en el caso de memoria distribuída (multicomputadores), donde se puede citar el trabajo recogido en [Ter89].

Especiales problemas se presentan cuando aparecen estructuras de datos recursivas, así como en el manejo de los arrays y de los punteros, debido a la dependencia de unos datos con otros y a no poder determinar exactamente esa relación entre ellos, conduciendo a veces a situaciones de ambigüedad. Asimismo, otro de los poblemas es el análisis interprocedural y el estudio del paralelismo anidado. Diversos estudios y resoluciones de estos problemas pueden encontrarse en [Nic89], [Li88] y [Hen90].

e) **Estilos de programación en paralelo.** Tenemos dos estilos de programación en paralelo: el paralelismo en los datos (*data parallelism*) o el paralelismo en el flujo de control del programa (*control parallelism*).

El paralelismo en los datos se basa en realizar múltiples operaciones en

paralelo sobre un gran conjunto de datos, mientras que el paralelismo en el control del programa se basa en tener simultaneamente varios flujos de control ejecutandose en paralelo. Ultimamente se está prestando una mayor atención al paralelismo dirigido por los datos. El lenguaje C* [Ros86] desarrollado por Thinking Machines Corporation es un ejemplo de la importancia que está tomando este estilo de paralelismo.

No hay que confundir esta dicotomía de estilos de programación como sinónimos a la oposición que hay en el diseño del hardware de una máquina entre SIMD y MIMD. Si bien es cierto que los computadores SIMD son más adecuados para ser programados en un estilo de paralelismo de datos, también pueden ejecutar un paralelismo de control por interpretación, dependiendo el que tal interpretación sea práctica de los detalles de coste de esa máquina determinada. De la misma forma, aunque un diseño MIMD es más adecuado para ser programado en un estilo de paralelismo con varios flujos de control, también se puede programar para un algoritmo con paralelismo en los datos.

En el caso de máquinas con diseño MIMD se utilizan 2 estilos de programación: el estilo *fork-join* y el estilo *SPMD* (Single Program Multiple Data) [Kar87]. En el estilo *fork-join* un proceso se divide en varios sub-procesos mediante la instrucción *fork*, y espera hasta que finalizan, reflejado mediante la instrucción *join* que sirve ademas para sincronizar los procesos. Este estilo se utiliza solo en máquinas MIMD de memoria compartida. En el estilo *SPMD* todos los procesos ejecutan el mismo programa, pudiendo discriminar algunos segmentos de código en función del identificador de ese proceso. Aunque este estilo es muy cómodo para el desarrollo de programas en paralelo, presenta la dificultad de conseguir una carga balanceada entre los diversos procesadores, ya que la carga para cada proceso puede ser muy diferente especialmente en aquellos problemas con estructuras irregulares. Como podemos observar, el estilo de programación *SPMD* es un híbrido entre los dos estilos anteriores para el caso de máquinas MIMD. Este estilo de programación se utiliza tanto para MIMD de memoria compartida como MIMD de paso de mensajes.

Un estudio detallado de los conceptos más importantes que nos aparecen en el campo del paralelismo se pueden encontrar en [Gar89].

2. REVISION DE LAS HERRAMIENTAS ACTUALES.

Inicialmente, los ordenadores se programaban en código máquina (tiras de unos y ceros), dedicándose un ordenador a un solo usuario. El desarrollo del primer compilador de Fortran al principio de los años 60, junto con toda la teoría de lenguajes formales desarrollada en aquella época por Chomsky dió lugar al desarrollo de los lenguajes de alto nivel, siendo la tendencia actual el desarrollo de lenguajes de programación cercanos al lenguaje natural.

Junto al desarrollo de lenguajes y compiladores, pronto se empezaron a desarrollar *herramientas* que facilitaran la tarea de programar una aplicación determinada. Estas herramientas recibieron el nombre de **depuradores**, pues ayudaban al programador a *depurar* un programa de los errores que tuviera y hacerlo por tanto mas eficaz. Posteriormente vino el desarrollo de interfaz gráficas, pues de manera visual se mejora notablemente todo el proceso de programar y depurar una aplicación. La facilidad de interfaz gráficas ha permitido que haya muchas mas personas que se acerquen al apasionante mundo de la programación, ya que la principal característica de estos entornos es que son entornos agradables al usuario, permitiéndole desarrollar programas de una forma cómoda y eficaz.

En el caso del paralelismo sucede de forma similar, aunque con una complejidad añadida por el hecho de tener que *pensar* en paralelo. Aunque en el mundo real las cosas suceden en paralelo, el programador está acostumbrado a programar unas acciones despues de otras, es decir, secuencialmente. La programación en paralelo no es una cosa sencilla. El paso de un programa secuencial a un programa paralelo es una tarea habitualmente laboriosa, complicada y necesitada de varios intentos antes de llegar a una solución con éxito. El principal problema que se encuentra el programador es que la *intuición* le falla, es decir, no es fácil *imaginar* cómo se va a desarrollar la ejecución de un programa paralelo. Para solventar este problema, se han desarrollado diversas herramientas que ayudan en la tarea de programar en paralelo. Como viene recogido en [Kar87] es necesario que aún aparezcan mas facilidades de programación para conseguir que mejore la productividad de los programadores.

El resto del artículo se va a centrar en presentar las principales herramientas que hay en la actualidad para los **multicomputadores**. En primer lugar presentaremos una revisión de las herramientas de que disponemos hasta la fecha para posteriormente introducir el FDP, un entorno

de programación para multicomputadores que hemos desarrollado [Gar91].

En el caso de los multicomputadores (máquinas MIMD con memoria distribuída) el desarrollo de un programa paralelo no se limita tan solo a dividir el programa secuencial en procesos, sino -y esto lo díficil del asunto- hay que coordinar (sincronizar) y comunicar estos procesos de cara a la correcto resolución del problema, conseguir una carga balanceada en cada procesador y determinar de que forma arrancan y finalizan los diversos procesos. Las ayudas para resolver esta falta de *intuición* en los programadores se ha dirigido desde una doble perspectiva:

a) El desarrollo de compiladores que a partir de código secuencial (o cuasi-secuencial) se encargen de paralelizar la aplicación.

b) El desarrollo de herramientas que permitan simular o emular un programa paralelo para una determinada máquina y ayuden al programador a ir adquiriendo una forma de pensar en paralelo.

2.1 Paralelizadores de lenguaje secuencial.

Dentro de la primera linea de investigación se han desarrollado varios trabajos. Uno de los mas conocidos es el proyecto PTRAN desarrollado en el IBM Watson Research Center para automaticamente reestructurar programas secuenciales escritos en Fortran para su ejecución en arquitecturas paralelas. La actual versión del compilador PTRAN-A (ver [All88a]) incorpora información interprocedural a su análisis de dependencia de datos. El sistema se organiza alrededor de una base de datos construida con información a partir del programa, incorporando rápidos algoritmos en un diseño cómodo y práctico.

Otro trabajo mas reciente se puede encontrar en [Ter89]. En él se presenta un compilador reconfigurable para multicomputadores que automaticamente realiza la partición del programa en procesos, el mapeo y la inserción de primitivas de comunicación para la correcta comunicación de los procesos bajo la guía de unas directivas añadidas por el programador. El compilador es capaz de compilar C estándar, generando código para varias máquinas objeto, estando implementado el compilador en Prolog. Para hacer que el compilador sea independiente de la máquina objeto la información sobre su arquitectura está contenida en unas reglas específicas en un fichero distinto. En primer lugar el compilador genera código intermedio para un multicomputador genérico (sin tener en cuenta la arquitectura específica), y en segundo lugar a partir de ese código intermedio y teniendo en cuenta la información sobre la arquitectura de la máquina objeto se genera el código

final.

Un último trabajo todavía mas reciente es el presentado en la *2 Conferencia de Computación Europea sobre Memoria Distribuída* [Ung91] para programación de sistemas debilmente acoplados. Se trata de un entorno que acepta un programa secuencial escrito en un subconjunto de C++ y es trasladado (tambien en dos fases) a un programa paralelo. En una primera fase se genera una partición del programa y un grafo de comunicaciones independiente de la máquina, para a partir de éste generar código para una arquitectura determinada. El programador usa las técnicas de programación orientada a objetos (definiciones de objetos e invocaciones) en un estilo secuencial en vez de definir un conjunto de procesos paralelos con primitivas de comunicación explicitas. En la actualidad se generan programas para el hipercubo iPSC/2 y tambien para sistemas de transputers con el sistema operativo HELIOS.

2.2 Entornos de simulación.

La otra linea de trabajo está formada por sistemas desarrollados para simular o emular el comportamiento de un multicomputador y adquirir una *familiaridad* con el complejo mundo paralelo. Habitualmente estas herramientas simulan mas que emulan los multicomputadores, debido a que los emuladores son programas complejos de desarrollar, permitiendo solo la emulación de una arquitectura determinada, siendo dificil añadirle nuevos tipos de arquitecturas. Aunque los emuladores dan unos resultados mas precisos sobre el hardware de la máquina, no son apropiados para comparar un gran número de arquitecturas alternativas. De hecho los desarrollos de emuladores se han limitado a los realizados por las diversas marcas comerciales con el fin de facilitar el manejo fácil de sus sistemas. Por contra, la simulación reduce tanto el coste como la precisión en comparación con la emulación. Sus dos grandes aplicaciones han sido como simuladores de lenguajes de alto nivel para arquitecturas genéricas (usados por programadores y diseñadores de algoritmos) y como simuladores de redes de interconexión para el estudio de su eficiencia. El uso de los simuladores de lenguajes permite a los programadores *aprender* a pensar en paralelo. Sin embargo, el efecto de una arquitectura específica en la ejecución de estos programas es impredecible.

Un cierto número de proyectos se han desarrollado para modelar los multicom-putadores, centrándose sobre todo en desarrollar entornos de

programación y herramientas para los programadores, mas que herramientas para trabajar con la arquitectura del sistema. Un ejemplo es PIE [Seg85], un entorno de programación e instrumentación para procesamiento paralelo. Aunque en un principio está diseñado para sistemas con memoria compartida, se espera que sea extendible a otros sistemas. PIE ofrece una representación gráfica de los procesos y sus relaciones.

Otro desarrollo es el Poker [Sny84], originalmente diseñado para emular una arquitectura específica, el *Configurable Highly Parallel Computer*. Poker presenta un entorno integrado y permite al usuario controlar varios niveles de la arquitectura objeto del sistema. Un entorno multi-ventanas permite controlar diferentes funciones del sistema, tal como crear una determinada topología de interconexión, asignar procesos a procesadores o escribir el código para un proceso particular. Aunque Poker ofrece un excelente modelizado del multicomputador, no es fácil añadirle nuevas arquitecturas; ultimamente se ha extendido para el Cosmic Cube.

Dos recientes desarrollos son el PARET y el Hypertool. PARET [Nic88a] ofrece un entorno gráfico interactivo para que el usuario practique los diversos modelos de multicomputadores, ofreciendo diversos resultados de la ejecución. PARET intenta simular el comportamiento que tendría un determinado programa en una arquitectura concreta mas que realizar una emulación detallada de la ejecución de dicho programa. El objetivo de PARET es el diseño y estudio de los multicomputadores como sistemas, mas que como componentes aislados. El modelo solo trata con las características mas importantes del software y hardware paralelo, representando los subsistemas del multicomputador como grafos de flujo dirijidos. Los algoritmos y las arquitecturas se representan graficamente, controlandose su estado de forma interactiva durante la simulación. Los resultados de la simulación acerca de diversos parámetros se ofrecen al terminar la simulación, pudiéndose almacenar en un fichero.

Mas novedoso aún que PARET es el entorno Hypertool [Wu90]. Hypertool está pensado para incrementar la productividad de los programadores. El desarrollo de un programa paralelo usando Hypertool es el siguiente: un programador diseña un algoritmo, realiza una partición en procesos escribiendo un programa secuencial como un conjunto de procedimientos (los diversos procesos). Gracias a que el programa es secuencial, se puede ejecutar en una máquina secuencial para una correcta depuración. Hypertool automaticamente convierte el programa a un programa paralelo para una máquina objeto con paso de mensajes, asigna cada proceso

a un procesador, simulando su ejecución e informando adecuadamente de los resultados obtenidos en cuanto a potencia, tiempo de ejecución, tiempo de comunicación, tiempo de inactividad en cada procesador y retraso introducido en la red en cada canal de comunicación. Este informe tambien muestra las dependencias de datos entre procesadores, así como el grado de paralelismo y la distribución de la carga en cada procesador. Si el diseñador del algoritmo no está satisfecho con el resultado, debe redefinir la estrategía de partición del programa secuencial y volver a realizar la simulación. El tiempo de transmisión de un mensaje se calcula mediante un clásico modelo lineal en función de la longitud del mensaje y un umbral de comienzo; el modelo asume que el tráfico de la red no es muy denso y por tanto ignora el retraso producido por la saturación de mensajes en un nodo.

3. UN ENTORNO AVANZADO: EL FDP.

Tomando como punto de partida las ideas reflejadas anteriormente acerca de los diversos entornos de programación paralela desarrollados hasta la fecha, nosotros hemos desarrollado un nuevo entorno de programación: el FDP (**F**riendly **D**istributed **P**ascal) que permite la programación y evaluación de un programa sobre un multicomputador genérico en el lenguaje Distributed Pascal.

3.1 Características básicas del FDP.

El FDP es un entorno de programación que se ha desarrollado para un IBM PC, AT, 386 o compatible, aunque es fácilmente trasladable a otras máquinas, por ejemplo workstations. Nos permite editar un programa en Distributed Pascal, compilarlo y detectar los posibles errores, asi como ejecutarlo de forma simulada para un multicomputador genérico. La herramienta desarrollada pertenece al segundo tipo de la clasificación dada anteriormente: los entornos de simulación de programas paralelos. El entorno se ha desarrollado de forma interactiva con una interfaz gráfica. En la figura 1 se muestra el aspecto del FDP con un programa editado con varias ventanas desplegadas.

El menú principal consta de lo siguiente:

a) **Ficheros**: Diversas funciones para manejo de ficheros, tales como

crear un fichero, cargar ó grabar un fichero, enviar a imprimir un fichero, acceder al sistema operativo sin dejar el programa y por último abondonar el entorno.

b) **Editar**: Se ha desarrollado un editor para crear nuevos ficheros ó modificarlos ficheros ya existentes. Al editor se le han incluido diversas facilidades para manejar adecuadamente el texto de un programa, tales como *Página Arriba* o *Página Abajo*, Inicio y Fin de linea, etc.

c) **Compilar**: Compila un fichero escrito en Distributed Pascal a un lenguaje intermedio. En caso de error, informa con un mensaje adecuado acerca de la clase del error y su localización.

d) **Run**: Ejecuta un programa previamente compilado.

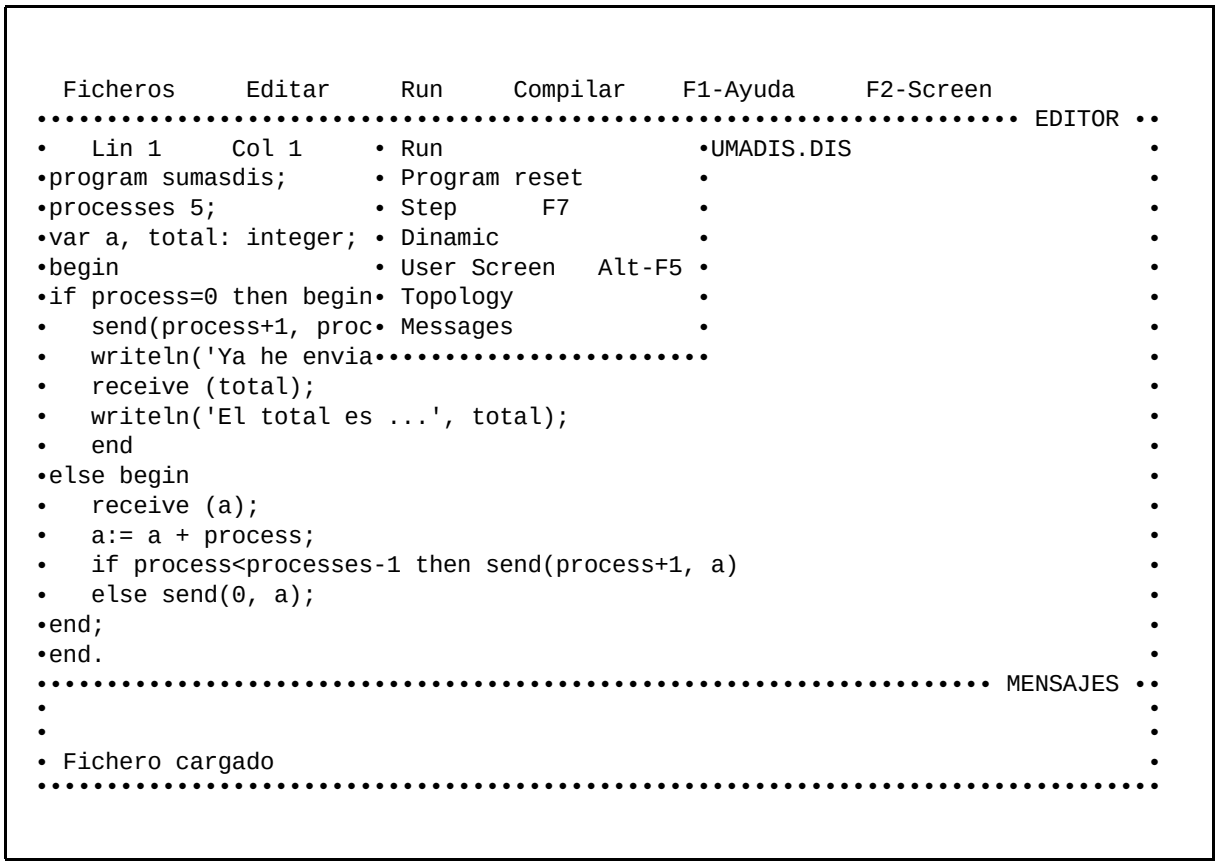

Fig. 1. Vista general del FDP, con un programa cargado y mostrándo las opciones de Run.

El entorno presenta tambien una ayuda al usuario (pulsando la tecla F1) que es *sensible* al contexto, dependiendo por tanto de donde nos encontremos para mostrarnos una informa-ción u otra. Asimismo presenta una ventana de comunicación con el usuario, donde va informando mediante adecuados mensajes de las funciones que está realizando: grabar un fichero, compilar un programa, etc. En el caso de estar editando un programa esta

ventana se puede suprimir (mediante la tecla F2) para permitir visualizar un mayor número de lineas del programa. Otra característica interesante del FDP es que permite en cualquier momento acceder a la pantalla de ejecución o pantalla de usuario (donde se muestran los resultados de la ejecución de un programa) mediante las teclas Alt-F5 o por medio de una opción del menú de ejecución.

El FDP permite simular cómo sería la ejecución de un programa en Distributed Pascal para una arquitectura determinada. Para ello, en la opción de Run hay un sub-menú (de estilo persiana - pull-down -) que permite seleccionar el número de procesadores físicos del multicomputador simulado asi como la topología de la red de interconexión. En la actualidad hay 3 topologías implementadas: el hipercubo, la malla 2D y el anillo.

Durante la ejecución se toma el código intermedio generado por el compilador y se interpreta, simulando el comportamiento de la máquina real con las características dadas. Una de las ventajas de este modo de proceder es la detección de los errores de ejecución debidos al procesamiento paralelo. Así en el caso de un *deadlock* se informa de dónde se quedaron bloqueados (a nivel de instrucción de código fuente) cada uno de los procesos. De esta forma se puede aprender cómodamente a pasar de la programación secuencial a la paralela, detectando los errores más frecuentes que se producen al programar los sistemas paralelos.

Otra de las características que incorpora el FDP es un depurador simbólico con tres funciones. Por una parte permite la ejecución paso a paso del programa, ejecutando de forma concurrente una línea de programa para cada uno de los procesos; esta linea será, en general, diferente para cada uno de los procesos,ya que la ejecución es asíncrona. Con esta opción un procedimiento ó función también se ejecuta paso a paso. La segunda función del depurador es similar a ésta, pero tratando los procedimientos ó funciones como una unidad, y ejecutándolos por lo tanto en un sólo paso. La tercera función permite observar el contenido de cada variable para cada proceso.

A continuación vamos a describir el lenguaje que acepta el FDP: el Distributed Pascal. Una mayor explicación del lenguaje así como del FDP puede encontrarse en [Gar91].

3.2. Modelo de máquina que soporta el lenguaje.

El lenguaje **Distributed Pascal** es un nuevo lenguaje de programación para arquitecturas avanzadas con paso de mensajes que también hemos

desarrollado. Se ha denominado Distributed Pascal debido a que es una extensión del Pascal estándar para multicomputadores. Gracias al conjunto de extensiones que se han añadido se puede programar de una forma cómoda y elegante los algoritmos paralelos consistentes en procesos que comunican mediante paso de mensajes. Dichas extensiones proporcionan una gran flexibilidad al permitir la particularización de código para cada proceso o para grupos de procesos.

El lenguaje base elegido ha sido el Pascal. Por una parte es un lenguaje muy educativo y tomado como base en muchas universidades para la enseñanza de la programación. Por ello, nos parece que el Distributed pascal puede ser un lenguaje muy adecuado para la enseñanza de la programación paralela. Por otra parte, tiene la suficiente potencia y versatilidad como parapermitir abordar un gran número de problemas complejos, en concreto los algortimos matriciales se programan y se resuelven perfectamente.

Como modelos paralelos para el desarrollo del lenguaje se han tomado ideas del CSP [Hoa78] y del Occam [Inm89], teniendo diversas diferencias prácticas que nos parecen que otorgan una mayor flexibilidad al lenguaje. Uno de los principales objetivos es que el lenguaje no sea específico para una arquitectura y microprocesador determinado, como ocurre en el caso del Occam.

Para hacer el lenguaje independiente de la máquina, el número de procesos no está limitado al número de procesadores. De cara a la programación de un algoritmo se emplea el concepto de **procesador virtual**, es decir, se supone que hay tantos procesadores como procesos se declaren. Le corresponde al kernel del run-time determinar qué proceso se carga en cada procesador (mapeo), pudiendo un procesador físico ejecutar mas de un proceso. Este mapeo es estático, es decir, una vez determinado en qué procesador se ejecuta cada proceso no cambia a lo largo de la ejecución del programa.

El modelo de comunicación se basa en el paso de mensajes. Para la programación se emplea el concepto de **red virtual**, la cual posee una topología totalmente conectada. De cara al usuario, este concepto permite comunicar directamente un proceso con cualquier otro, olvidándose de la red de interconexión existente en un multicomputador concreto. Incluso puede implementarse el concepto de red virtual totalmente conectada sobre un multiprocesador con memoria compartida. Nuevamente le corresponde al kernel el determinar si los procesos origen y destino de un mensaje están en el mismo procesador y la comunicación es interna, o bien ejecutar el algoritmo

de encaminamiento en cada procesador por el que pasa un mensaje si los procesos origen y destino están en procesadores diferentes, multiplexando en el tiempo los canales físicos de un procesador para dar servicio a todos los procesos que se están ejecutando en el mismo. Esto es posible gracias a que el mapeo es estático y conocido en todo el sistema.

El estilo de programación adoptado es el **SPMD** (Single Program Multiple Data) [Kar87], es decir, todos los procesos eje-cutan el mismo programa. En el caso de que haya varios procesos en un procesador, existe solo una copia del código en la me-moria, teniendo cada proceso sus propias variables. De este modo se reduce la ocupación de memoria; sin embargo, esto no implica que todos los procesos ejecuten las mismas instrucciones. Para poder discriminar algunas zonas de código y que sean ejecutadas tan sólo por algun proceso el lenguaje posee la variable *process* que identifica cada uno de los procesos. Este identificativo es único para cada proceso. Así para el proceso 0 la variable *process* valdrá 0, para el proceso 1 *process* será 1, etc.. En la figura 2 se puede ver cómo se podría discriminar el código de un programa si quisiéramos que una parte del mismo fuese ejecutada tan sólo por el proceso 0.

```
        (...)
    if process=0 then begin
        (...)  {ejecutado tan sólo por el proceso 0}
    end;
        (...)
```

Fig. 2. Particularización de código para un proceso.

Comentar dos detalles. Por una parte, decir que tambien se podía usar la sentencia CASE para discriminar código entre procesos, al igual que la sentencia IF. Por otra, decir que tambien se pueden calcular funciones donde una de sus variables sea *process*. En este caso todos los procesos ejecutan las mismas instrucciones pero el resultado es diferente en cada proceso.

Con respecto a las variables de cada proceso todas se declaran en el programa principal, no pudiendo particularizar algunas variables para algún proceso determinado. Ello implica que en todos los procesos las variables tienen el mismo nombre, tomando ya un valor u otro en función del proceso de que se trate. Esta forma de manejar las variables es adecuada para algoritmos *regulares*. Por algoritmos *regulares* queremos expresar algoritmos en donde todos los procesos tengan un comportamiento similar de tal forma que usen unas estructuras de datos muy parecidas. En caso contrario, donde cada proceso tuviera variables distintas a los demas procesos (distintas en

cuanto a su estructura, no en cuanto a su valor que eso por supuesto que si lo van a tener), sucedería que al no poder particularizar las variables todos los procesos tendrían todas las variables lo que supondría un desperdicio de memoria considerable. Por ello, el Distributed Pascal es sobre todo adecuado para algoritmos regulares en el sentido referido.

Para mantener la total independencia del lenguaje con respecto a la máquina, el programador nunca debe definir donde se va a ejecutar cada proceso. Esta es otra de las ventajas que tiene con respecto al Occam, ya que este lenguaje exige del usuario que explicitamente asigne procesos a procesadores y canales de comunicación virtuales a los enlaces físicos. Este es tambien el caso del ACLAN [Pla88], un lenguaje síncrono que para su ejecución necesita de un fichero auxiliar donde se muestre cómo está formada la red de interconexión. En el caso del Distributed Pascal no es necesario, encargándose el sistema operativo de asegurar esa independencia. Es importante que el programador tenga esto en cuenta, pues el correcto funcionamiento de un programa nunca puede depender de la configuración de la red de interconexión.

3.3. Características paralelas del lenguaje.

Una vez definido el modelo de máquina para el cual está pensado el lenguaje, en este apartado vamos a detallar de que forma el lenguaje maneja todo lo concerniente al paralelismo, tal como la definición de procesos, la comunicación entre ellos, etc.

3.3.1 **Creación y finalización de procesos.**

Como se ha mostrado arriba, la abstracción de procesador virtual dá un modelo de procesos independiente del número de procesadores físicos que se tengan en el multicomputador. A cada proceso se le va a identificar mediante un identificador de proceso (*process_id*). La creación de procesos va a ser sólo de tipo estático (al inicio de la ejecución), definiendose el número de procesos al inicio del programa. Un programa en Distributed Pascal adopta el siguiente formato:

<programa>::= **program** <identificador> **;** <declaración_procesos>
.

<declaración_procesos>::= **[processes** <entero_sin_signo> **;]**

<bloque>

expresado en reglas BNF extendidas. Como se puede ver, la declaración del número de procesos se realiza inmediatamente después que la declaración del nombre del programa. En caso de que no aparezca se toma por defecto que sólo existe un proceso y el programa es secuencial. El no terminal *entero_sin_signo* es un número entero sin signo que indica cuántos procesos se van a ejecutar en paralelo.

El no terminal *bloque* representa un bloque de un programa en Pascal estándar, al que se le han añadido en nuestro caso una serie de primitivas para manejar las comunicaciones.

Con respecto a la finalización de los diversos procesos, no se provee ningún mecanismo explícito para manejar su terminación. En esto es semejante a Occam y diferente a CSP. Las tareas finalizan cuando terminan de ejecutar el código que les corresponde.

Algunos lenguajes similares al Distributed Pascal permiten que a su vez un proceso se pueda dividir en un número de tareas que se ejecutan de forma concurrente (este es el caso del EDAM [Cou90] entre otros) y permitir con ello un mayor grado de paralelismo. En nuestro caso no es así; un programa se compone de un número de procesos que se ejecutan en paralelo, siendo cada uno de estos procesos una parte de código secuencial.

3.3.2 **Comunicación de procesos.**

En Distributed Pascal se ha tomado el modelo de comunicación mediante **buffer**, con un tamaño de buffer de 20 elementos. Dicho tamaño se ha tomado exageradamente grande para impedir errores de ejecución. Mas adelante se piensa optimizar este tamaño para minimizar el gasto de memoria que supone.

El buffer para la recepción de mensajes se ha implementado por medio de una cola con estrategia FIFO. Como esta primitiva no permite aceptar mensajes dependiendo de qué proceso es el que envía el mensaje, hay que asegurar que los diversos mensajes a una tarea son enviados en el mismo orden en que dicha tarea espera recibirlos. Cualquier implementación del lenguaje debe asegurar que todos los mensajes transferidos entre un par dado de procesos sean recibidos en el mismo orden en que ellos son envíados. Sin embargo, nada se puede asegurar acerca de diferentes procesos que envían mensajes a un mismo destino. Entonces el comportamiento de un algorítmo

debe ser independiente del orden de llegada de los mensajes, cuando estos vienen de fuentes diferentes. De lo contrario, habitualmente se producirá un error de ejecución (se realiza una comprobación entre el tipo de datos enviado y el tipo de datos recibido) o bien se producirán resultados incorrectos. Otra solución alternativa consiste en poner una cabecera en el mensaje indicando el proceso origen y el número de orden si hace falta.

Se ha elegido este modelo (a diferencia por ejemplo del Occam) para dar una mayor libertad a las procesos y conseguir asi un mayor grado de paralelismo. Si en algún algoritmo se necesitara la sincronización, el usuario la puede implementar por medio de mensajes que sean de reconocimiento, es decir, con la pareja send-receive cada vez que un proceso envie un mensaje se queda esperando a recibir una señal de que dicho mensaje ha llegado.

La comunicación entre los diversos procesos, se hace mediante las siguientes primitivas: *send, receive, received y broadcast.* Las comunicaciones se establecen a nivel de procesos, no necesitando ningún medio de enlace entre los procesos, como los *canales* en el caso del Occam o los *puertos* en el caso del EDAM. Esto simplifica el formato de las primitivas y libera al programador del engorro que supone estar definiendo canales entre todos los procesos que deseen comunicar.

La primitiva *send* sirve para enviar un mensaje de un proceso a otro proceso. Adopta el siguiente formato:

```
send ( node_dest , message)
```

siendo *node_dest* el proceso destino del mensaje y *message* el mensaje a enviar, el cual puede ser de cualquier tipo simple, estructurado o definido por el usuario.

El compilador admite que la variable *node_dest* no sea una constante, ni tan siquiera una expresión cuyo valor se pueda calcular en tiempo de compilación. Es decir, que el proceso receptor puede quedar indeterminado hasta la ejecución del programa, en donde se elegirá en función de los datos de entrada. Debido a ello, en muchos casos no es posible determinar en tiempo de compilación como va a ser el flujo de comunicaciones entre los procesos, ni a partir de este dato realizar el mapeo de procesos a procesadores, como es el caso descrito en [Cou90].

Debido al modelo de comunicación, el proceso que envía nunca se queda a la espera (bloqueado) del proceso receptor. En el caso de que el buffer esté lleno, ocurrirá un error de ejecución. Nos ha parecido mejor esta opción

que el hacer que el proceso que envia se quedara suspendido esperando hasta que el buffer admitiera el mensaje que quiere enviar, aunque ciertamente se podía haber elegido esta última alternativa.

La primitiva *receive* sirve para recibir un mensaje de otro proceso. Adopta el siguiente formato:

```
receive (var_dest)
```

donde *var_dest* indica la variable donde se guarda el mensaje recibido, la cual debe ser del mismo tipo que la variable *message* en el correspondiente *send* (se chequea por el compilador).

Debido al modelo de comunicación, si un proceso ejecuta la anterior instrucción y aún no ha recibido ningún mensaje, dicho proceso se queda bloqueado esperando hasta que llegue algún mensaje. Por tanto, un algoritmo podría caer en el *deadlock*, pues podría darse el caso de que todos los procesos estuvieran esperando un mensaje y por tanto todos se quedaran bloqueados. Por ello, cuando se diseñe un algoritmo hay que tener en cuenta esto para no caer en este error. En el simulador que hemos desarrollado se tiene en cuenta esta posibilidad detectando este error de ejecución e informando adecuadamente.

El lenguaje tiene también la función booleana *received*. Dicha función al ser ejecutada por un proceso me dará un resultado TRUE si ya ha llegado algún mensaje, o FALSE en caso contrario. Por medio de esta función, un proceso puede elegir entre tomar un mensaje de su cola FIFO de entrada o ejecutar otras sentencias, evitando por tanto el bloqueo producido cuando se ejecuta *receive* y no hay mensajes en la cola de entrada (suponiendo que haya algunas instrucciones alternativas a ejecutar). Asimismo se puede simular la construcción ALT del Occam entre la recepción de un mensaje y la ejecución de un conjunto de instrucciones. Por último, en algunos algoritmos es necesario que una tarea deba enviar un mensaje a todas las demás procesos. Para ello se ha implementado otra primitiva denominada *broadcast.* Adopta el siguiente formato:

```
broadcast (message)
```

donde *message* es el mensaje que se envía a todos los demás procesos.

Gracias a haber desarrollado el binomio lenguaje-compilador las primitivas adoptan un formato compacto y se evitan engorrosas llamadas a

librerías del sistema donde el usuario debe añadir la longitud y dirección del mensaje, su tipo, etc. Un ejemplo de esto se tiene para el iPSC y sus primitivas de comunicación para Fortran y C [Mol86]; para enviar un mensaje entre nodos la primitiva send adopta el siguiente formato

send (cid, type, buf, len, node, pid)

donde

cid: identificativo del canal a enviar el mensaje.
type: tipo del mensaje.
buf: buffer donde está almacenado el mensaje.
len: longitud (en bytes) del mensaje.
node: nodo destino del mensaje.
pid: identificativo del proceso.

Como podemos ver, el formato de estas primitivas en Distributed Pascal es mucho mas compacto y adecuado.

Resaltar que en estas primitivas las comunicaciones se establecen entre procesos, sin tener en cuenta la localización de dichos procesos en los procesadores físicos. De esta forma se consigue uno de los requisistos del lenguaje que es su independencia respecto de una arquitectura determinada.

3.3.3 **Expresión del no-determinismo.**

La mayoría de los actuales lenguajes incluyen características que permiten expresar situaciones en las cuales varios hechos pueden suceder al mismo tiempo; la forma de elegir uno de esos hechos es dejada a la implementación concreta del lenguaje. Por ejemplo, en Occam y CSP permiten que los procesos vayan precedidos de condiciones (*guardas*) que se deben verificar para que se realice ese proceso; en el caso de que haya varias condiciones que se cumplan, se elige arbitrariamente (de forma **no determinista**) uno de los procesos.

Esta característica es debida a que estos lenguajes se usan para tratar con aplicaciones de tiempo real, siendo inherente para dichas aplicaciones el no determinismo. En Distributed Pascal no se ha implementado esta característica, pues para el tipo de algoritmos que está pensado no se necesita.

3.3.4. **Un sencillo ejemplo.**

Una vez descrito las características mas importantes del lenguaje, se va a mostrar un ejemplo de aplicación para la correcta comprensión de las características citadas.

En este ejemplo vamos a presentar como se programaría la suma cíclica. Este sencillo programa consiste en calcular la suma de los identificativos de cada proceso. Cada proceso (empezando por el proceso 0) recibe la suma parcial y le añade el valor de su identificativo, enviando esta cantidad al siguiente proceso. El programa sirve sobre todo para ilustrar el envío y la recepción de mensajes. El programa principal puede verse en la figura 3.

```
begin
if process=0 then begin
   send(process+1, process);
   receive (total);
   writeln('El total es ...', total);
 end
else begin
   receive (a);
   a:= a + process;
   if process<processes-1 then send (process+1, a)
                          else send(0, a);
end;
end.
```

Fig. 3. Programa para una suma cíclica.

Se pueden realizar los siguientes comentarios:

a) Las variables *total* y *a* están replicadas en todos los procesos y se declararían al principio del programa. Aunque en todos los procesos se llamam igual, no tienen nin-guna relación las variables de procesos distintos. Debido a que el lenguaje no permite particularizar las variables a un proceso concreto sino que se replican en todos los procesos, el proceso 0 posee una variable *a* aunque no la use, y de igual forma el resto de procesos poseen la variable *total*.

b) El estilo de programación es SPMD. Para particu-larizar el código a algunos procesos se ha empleado la va-riable *process*. Así se tienen cuatro bloques de código: lineas 3-5 son ejecutadas sólo por el proceso 0, lineas 8-11 son ejecutadas por todos los procesos menos el 0, linea 10 es ejecutada por todos los procesos menos el proceso 0 y el último proceso (con identificativo processes-1), y la linea 11 es ejecutada tan sólo por el último pro-ceso.

c) Como vemos el programa es independiente del número de procesos que se ejecuten y del número de procesadores que tenga el multicomputador.

Para la ejecución del programa hay que añadirle estos dos parámetros: el número de procesos se añade en el propio texto del programa por medio de la constante *processes* (tal como se detalló anteriormente), mientras que el número de procesadores (nodos) en el multicomputador simulado se elige por medio de un menú en el FDP.

d) El proceso 0 se encarga de imprimir el resultado. Realmente esto sería un envío de ese dato al host para que lo imprimiera, ya que el host es el único que habitualmente tiene acceso a los periféricos. Gracias a la optimización que realiza el compilador en las sentencias de lectura/escritura, es posible escribirlo de esta manera y que sea el compilador el que ya se encargue de cambiar a una instrucción de envio de mensajes. Esta optimización hace que el programa sea mas *leible* y *comprensible* por un programador distinto al que realizó el programa.

```
begin
if (process=0) or (process = processes div 2) then begin
   send(process+1, process);
   if process=0 then begin receive (total);
      receive(a);
      total:= total + a;
      writeln('El total es ...', total);
   end;
   end
else begin
   receive (a);
   a:= a + process;
   if (process=processes-1) or (process=(processes-1) div 2) then send(0, a)
   else send(process+1, a);
end;
end.
```

Fig. 4. Explotando el paralelismo del programa.

e) Aunque el programa es paralelo, explota muy poco el paralelismo, debido a que la primera instrucción de los procesos es un *receive* del proceso anterior, y cada proceso envía su mensaje al final de su código. Ello hace que el comportamiento del programa sea practicamente secuencial. Para aumentar el rendimiento del algoritmo anterior, se podrían dividir los procesos en dos grupos y realizaran los anteriores calculos en paralelo. El programa podría adoptar la forma que se muestra en la figura 3. La dife-rencia entre este programa y el anterior es que el número de procesos se ha dividido en dos grupos: del proceso 0 hasta la mitad, y de la mitad hasta el proceso final. En cada grupo cada proceso realiza una operación igual que antes: sumar su identificativo del proceso a la suma parcial y enviarlo al siguiente proceso en

la cadena. La diferencia estriba en que *ahora* las operaciones en cada grupo se ejecutan en *paralelo*.

Como se puede ver en este sencillo ejemplo, para que un algoritmo paralelo funcione eficientemente no solo se debe disponer de un lenguaje que permita programar en paralelo, sino que ademas se debe diseñar el algoritmo de tal manera que explote adecuadamente el paralelismo de la máquina en la que se va a ejecutar dicho programa.

4. CARACTERISTICAS ESPECIALES DEL FDP.

Como se ha descrito anteriormente, cada uno de los entornos de programación que hay desarrollados en la actualidad permiten la simulación de un programa en un multicomputador resaltando alguna de las características que poseen estas máquinas. Este también ha sido el caso del entorno que hemos desarrollado.

En el FDP la atención se centra en una característica novedosa que presentan este tipo de máquinas: la **reconfiguración dinámica de la red de interconexión**. Esta característica se ha desarrollado debido a que para este tipo de máquinas su cuello de botella son las comunicaciones entre nodos, resultando que para un programa que presente una alta tasa de comunicaciones se degrada el rendimiento de esta arquitectura. Para un gran número de nodos es impracticable una interconexión total, por lo que se deben adoptar diversas topologías para la red de interconexión. En las máquinas actuales la comunicación entre nodos vecinos es más rápida que entre nodos no vecinos, necesitándo en este último caso implementar algún algoritmo de encaminamiento de mensajes a traves de los nodos intermedios.

Especialmente con algoritmos de encaminamiento del tipo store-and-forward se deben emplear técnicas de mapeo refinadas para lograr que los procesos que comunican con una gran frecuencia se encuentren situados en nodos vecinos de la red de interconexión. Pero en el caso de algoritmos donde no se pueda predecir *a priori* cómo va a ser el flujo de comunicaciones, es dificil lograr un adecuado mapeo de los procesos en los procesadores.

En este caso, y tambien en el caso de aplicaciones cuyo flujo de comunicaciones cambia a lo largo de la ejecución del programa, una alternativa que se está desarrollando ultimamente para resolver este problema es la *reconfiguración dinámica de la red*. Ya hay varios proyectos que se están

llevando a cabo, como por ejemplo el Esprit P1085 Supernode [Nic90] o el DAMP System [Bau91].

El FDP permite la reconfiguración dinámica de la red de interconexión de cara a evaluar la ganancia que obtenemos mediante este técnica avanzada de los multicomputadores. Para ello se ha desarrollado un algoritmo que depende basicamente de 2 umbrales y que permite la reconfiguración local de la red. Para una explicación mas detallada del algoritmo y de los resultados obtenidos hasta la fecha puede verse [Gar91a].

5. CONCLUSIONES.

En este artículo se ha presentado el estado actual de las herramientas para programación paralela para el caso de multicomputadores. Se han presentado las dos tendencias que hay en el desarrollo de facilidades de programación: la paralelización automática de código a partir de un programa secuencial (quizás con ayuda de algunas directivas que indique la arquitectura de la máquina), y el uso de entornos de programación que permiten la simulación de un programa paralelo, permitiendo de esta forma que el programador se *acostumbre* a pensar en paralelo.

Por último, se ha expuesto un entorno novedoso de programación que hemos desarrollado: el FDP. Este entorno permite la simulación de un programa paralelo escrito en Distributed Pascal para un multicomputador genérico, fijándose sobre todo en una característica de estas máquinas: la reconfiguración dinámica de la red de interconexión.

REFERENCIAS

[All88a] Allen, F.; Burke, M.; Charles, P.; Cytron, R. and Ferrante, J. "An overview of the PTRAN analysis system for multiprocessing". *Journal of Parallel and Distributed Computing* 5, 617-640 (1988).

[Bau91] Bauch, A.; Braam, R. and Maehle, E. "DAMP: A dynamic reconfigurate multi-processor system with a distributed switching network".

2nd European Distributed Memory Computing Conference, Munich, April 1991.

[Ber84] Berry, G. and Cosserat, L. The synchronous programming language esterel and mathematical semantics. *Seminar on Concurrency,* Springer-Verlag LNCS 197 (1984)

[Cou90] Counilh, M.C. and Roman, J. Expression for massively parallel algorithms -description and illustrative example. *Parallel Computing 16*, (1990), 239-251, North-Holland.

[Gar89] García Carrasco, José Manuel. "Modelos de arquitecturas, lenguajes y compiladores para procesamiento en paralelo". Dept. de Ingeniería de Sistemas, Computación y Automática. Univ. Politécnica de Valencia. 1989.

[Gar91] García, J.M. and Duato, J. "FDP: An environment for a MIMD programming with message-passing". Technical Report GCP #1/91, Departamento de Ingeniería de Sistemas, Computación y Automática, Universidad Politécnica de Valencia, 1991.

[Gar91a] García J.M. and Duato, J. "An algorithm for dynamic reconfiguration of a multicomputer network". Technical Report GCP #2/91, Departamento de Ingeniería de Sistemas, Computación y Automática, Universidad Politécnica de Valencia, 1991.

[Hen90] Hendren, L.J. and Nicolau, A. "Parallelizing programs with recursive data structures". *IEEE Transaccions on Parallel and Distributed Systems,* Jan. 1990, pp. 35-47

[Hoa78] Hoare, C.A.R. "Communicating sequential processes". *Communications of the ACM 21,* no.8, pp. 666-677, aug. 1978

[Inm88] Inmos Limited. *Occam-2 Reference Manual.* Prentice-Hall, 1988

[Inm89] Inmos Corporation. *The Transputer Databook*. Inmos Ltd., England 1989

[Kar87] Karp, Alan. "Programming for parallelism". *Computer,* May 1987, pp. 43-57

[Kel76] Keller, R.M. "Formal verification of parallel programs". *Comm. ACM,* vol. 19, no. 7, july 1976, pp. 371-384

[Li88] Li, Z. and Yew, Pen-Chung Efficient interprocedural analisys for program parallelization and restructuring. Procc. SIGPLAN '88 Conference on Programming, Language Design and Implementation, pp. 85-99

[Mol86] Moler, C. and Scott, D.S. Communication utilities for the iPSC. Technical Report n. 2 Intel Scientific Computers, Aug. 1986.

[Nic88a] Nichols, K.M. and Edmark, J.T. "Modeling multicomputer systems with PARET". *IEEE Computer*, May 1988, pp. 39-48.

[Nic89] Nicolau, Alexandru. "Run-time disambiguation: Coping with statically unpredictable dependencies". *IEEE Trans. on Computers,* Vol. 38, n. 5, May 1989

[Nic90] Nicole, Denis A. Reconfigurable Transputer Processor Architectures. in *Multi-processor Computer Architectures*, T.J. Mountain and M.J. Shute Editors, North-Holland, 1990

[Owi76] Owicki, Susan and Gries, David Verifying Properties of Parallel Programs: An axiomatic Approach. Comm. ACM, vol. 19, no. 5, may 1976, pp. 279-285

[Pad86] Padua, David and Wolfe, Michael. Advanced Compiler Optimizations for Supercomputers. Comm. of the ACM, Vol. 29, n. 12, December 1986

[Pla88] Plata, O.G. and Zapata, E.L. "ACLAN: Manual de programación". Dept. de Electrónica. Universidad de Santiago de Compostela, 1988.

[Ros86] Rose, J.R. and Steele, G.L. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1986

[Seg85] Segall, Z. and Rudolph, L. "PIE: A programming and instrumentation environment for parallel processing". *IEEE Software*, Nov. 1985, pp. 22-37.

[Sny84] Snyder, L. Parallel Programming and the POKER programming environment. IEEE Comp. Mag., July 1984, pp. 27-36

[Ter89] Terrano, A.; Dunn, S. and Peters, J.E. Using an Architectural knowledge base to Generate Code for Parallel Computers. CAMC september 1989, vol. 32, n.9, pp. 1065-1072

[Ung91] Ungerer, T. and Bic, L. "An object-oriented interface for parallel programming of loosely-coupled multiprocessors systems". *2nd European Distributed Memory Computing Conference*, Munich, April 1991.

[Wu90] Wu, Min-You and Gajski, D.D. Hypertool: A programming aid for message-passing systems. IEEE Trans. on Parallel and Distributed Systems, vol. 1, no. 3, July 1990, pp. 330-343.