# El Procesamiento Paralelo, Enfoque Cualitativo y Simulación

Leslie Murray[†]

Escuela de Ingeniería Electrónica
Universidad Nacional de Rosario
Argentina

Mayo 26, 2000.

**Resumen**

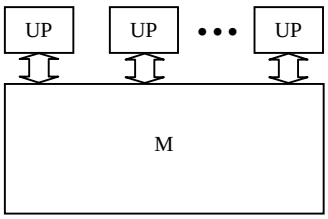
El presente trabajo constituye un material de apoyo en la introducción de ciertos conceptos básicos del procesamiento paralelo y afines. Es de interés que las ideas aquí vertidas resulten de aplicación práctica en proyectos interdisciplinarios relacionando fundamentalmente las áreas de Computación y Electrónica Digital. He concentrado la atención particularmente en dos de esos aspectos que resultan característicos y que constituyen parte de la esencia misma del mecanismo de ejecución simultánea de programas dentro de una misma máquina; son ellos: 1) La forma en que se organiza la memoria del sistema y 2) Los métodos a través de los cuales se garantiza la ejecución sincronizada de los diferentes procesos. Se abordan luego temas relacionados con los dos simuladores de arquitecturas de procesamiento paralelo (Limes y MulSim) utilizados para implementar y analizar algunas de las ideas anteriores, se comentan los fundamentos de su funcionamiento y su modo de uso con vistas a la realización de una serie de ensayos, cosa que se hace en último lugar.

# 1. Introducción

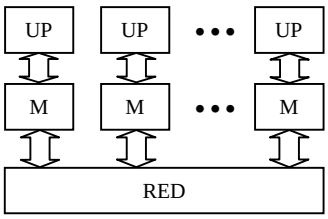
En 1966 según algunos, en 1972 según otros, Flynn estableció una clasificación según la cual todo sistema de cómputo pertenece a una de las siguientes cuatro categorías: **SISD** (*Single Instruction stream - Single Data stream*), **SIMD** (*Single Instruction stream - Multiple Data stream*), **MISD** (*Multiple Instruction stream - Single Data stream*) y **MIMD** (*Multiple Instruction stream - Multiple Data stream*).
A la categoría SISD corresponde el modelo de computador tradicional (Von Neumann), en el que más allá de las variantes, se lleva acabo un procesamiento secuencial mediante la ejecución de una única instrucción sobre un único dato por vez (ej. cargar el contenido de una dirección de memoria en un acumulador, sumar 1 al contenido de un registro, etc.).
Dentro del tipo SIMD se consideran los sistemas que a partir de una única instrucción son capaces de procesar un conjunto o array de datos en forma simultánea, con la particularidad de ejecutar la misma instrucción sobre cada uno de sus elementos. Estos sistemas se componen de varias unidades funcionales idénticas (Unidades Aritmético-Lógicas, Registros de Direcciones de Memoria, etc.) comandadas todas ellas desde una única unidad de control encargada de recibir y procesar cada instrucción a ejecutar. Podría considerarse que, aunque en forma restringida, esta arquitectura incorpora en cierta medida el paralelismo desde el momento que es capaz de ejecutar "algunas tareas" (no "cualquier tarea") en forma simultánea, concretamente puede realizar N operaciones O, sobre N datos D, al mismo tiempo. En rigor no se puede

[†] Este informe, realizado en el marco de mi programa de Maestría en Informática, PEDECIBA, Facultad de Ingeniería, Universidad de la República, Uruguay, fue aportado como material de apoyo a un curso de la Cátedra "Arquitectura y Diseño de Computadoras" de la Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina. Un especial agradecimiento a los Ingenieros Sergio Geninatti y Santiago E. Roatta, por la colaboración y orientación recibida.

concebir una máquina cuyo modo de operación sea exclusivamente SIMD dado que es imposible pensar en tareas sobre segmentos de datos tales que absolutamente todos tengan aspecto vectorial. Para que esta modalidad pueda ser implementada, la arquitectura correspondiente deberá combinar necesariamente las prestaciones SIMD con las SISD.
La categoría MISD corresponde a configuraciones en las que un tren de unidades de procesamiento se surte por uno de sus extremos de un dato extraído de memoria y devuelve un resultado a la misma memoria por el extremo opuesto, en tanto cada unidad del tren toma por entrada al dato procesado por la unidad anterior y da por salida el que será tomado como entrada por la unidad siguiente. La característica MI está en el hecho de que cada unidad de procesamiento va ejecutando una operación distinta sobre un único dato (SD) que va avanzando hasta convertirse en el resultado.
MIMD es un modelo en el cual el sistema puede considerarse de alguna forma particionado en unidades con capacidad de, 1) Desarrollar su propio cómputo y 2) Comunicarse con las demás; de modo que al ejecutar cada una de ellas un determinado flujo de instrucciones (SI) sobre su propio flujo de datos (SD) se logra que en conjunto, múltiples flujos de instrucciones (MI) actúen sobre otros tantos flujos de datos (MD).
SISD constituye, como ya se dijo, el modelo clásico de arquitectura de computadoras en el que se basaron los diseños a lo largo de muchos años. En procura de lograr aumentos de performance, la producción de máquinas comerciales ha ido incorporando elementos de los demás modelos. En mayor medida SIMD y en menor MISD sirven al diseño de computadores de propósito específico pero no de equipos de distribución masiva, en tanto la última categoría MIMD constituye no sólo la más ambiciosa combinación posible de las dos variables (*Instruction stream - Data stream*) sino el fundamento mismo del procesamiento paralelo en su expresión más completa. Para Kai Hwang [Advanced Computer Architecture - p.11] *"...sistemas de cómputo paralelo son aquellos capaces de ejecutar programas en el modo MIMD..."*. Estos sistemas a su vez pueden clasificarse en dos grandes categorías, los **Multiprocesadores** o sistemas de memoria compartida en los cuales la comunicación entre las distintas unidades se realiza, en forma implícita a través de variables compartidas dentro de una única memoria principal y los **Multicomputadores** o sistemas de memoria distribuida en los que la comunicación explícita entre unidades se realiza a través de una Red de paso de mensajes y donde cada unidad de procesamiento opera sobre su propia memoria.




MULTIPROCESADOR
(Memoria Compartida)




MULTICOMPUTADOR
(Memoria Distribuida)

El mecanismo de paso de mensajes es el que permite a los procesos de un Multicomputador correr en forma integrada, coherente y sincronizada a fin de que la ejecución de cada uno resulte una cooperación con la ejecución global de una aplicación que, en pos de obtener un mejor rendimiento, haya sido distribuida entre las distintas unidades. Este paradigma se apoya en dos operaciones básicas, una para enviar mensajes, SEND y otra para recibirlos, RECEIVE. La idea es que cualquiera sea la topología de la red que de soporte al paso de los mensajes se establezcan canales a través de los cuales dos procesos P y Q puedan intercambiar información, por ejemplo del siguiente modo: SEND(P,m); RECEIVE(Q,m). En este caso el proceso Q envía el mensaje m al proceso P mediante la emisión, a través del canal establecido, de la sentencia SEND dirigida explícitamente al proceso P el cual recibe el mensaje m del proceso Q y acusa recibo de él, a modo de confirmación, mediante la emisión a través del mismo canal de la sentencia RECEIVE en la que especifica el destinatario de la misma (no de m), o sea Q. Este tipo de comunicación implica el establecimiento de un canal (en tiempo y espacio) entre procesos para lo cual ambos deben estar simultáneamente en condiciones de comunicarse, de ahí el nombre de sincrónico. Su contrapartida es el modo asincrónico en el que la comunicación se realiza a través de buffers que actúan como buzones de correo. En este caso el proceso que envía el mensaje se despreocupa respecto a sí el destinatario está o no en condiciones de recibirlo. No obstante las demoras pueden sobrevenir mientras el proceso que hubiera enviado un mensaje no reciba la confirmación departe del proceso que lo hubiese recibido. Los mensajes pueden ser datos, información de sincronización, señales de interrupción, etc.

En los Multiprocesadores la implementación y ejecución de procesos paralelos difiere esencialmente del paso de mensajes ya que contempla una única forma de comunicación entre ellos, bastante más restringida pero por cierto mucho más rápida, materializada a través de lecturas/escrituras de posiciones compartidas de memoria en las que se alojen variables reconocidas por todos los procesos o sea de carácter global. Es éste modelo de computador paralelo el que vamos a abordar con mayor detalle, tanto que desde aquí y hasta el final del presente trabajo toda mención a sistema de cómputo/computador de procesamiento paralelo será implícitamente una referencia a un Multiprocesador.
He hecho alusión a memoria "compartida" y "distribuida" porque es la terminología habitualmente utilizada para clasificar los sistemas de memoria de Multiprocesadores y Multicomputadores, pero de hecho no se trata de calificativos que representen el opuesto uno del otro. Lo contrario de compartido es privado y esa es la clasificación que más apropiadamente describiría la situación. Que las CPUs direccionen todas el mismo espacio de memoria o que cada una tenga su propio juego de direcciones es lo que pretenden señalar las definiciones de Multiprocesador y Multicomputador y no si la memoria es distribuida o centralizada, porque ello tiene que ver con su ubicación física, aunque en general las compartidas estén centralizadas y las privadas distribuidas. Hago esta aclaración no sólo porque implica definir las cosas con propiedad sino porque además refuerza el concepto de lo que se debe entender por cada uno de los dos modelos.
Hay argumentos variados a favor de uno y otro; para los defensores de Multicomputadores los sistemas de memoria compartida admiten un tráfico limitado sin importar cuanta memoria caché utilicen (ver Jerarquía de Memoria) debiendo por lo tanto resultar acotado el número de procesadores. En favor de Multiprocesadores se pregona que sólo se justifica distribuir la memoria si el problema es suficientemente paralelizable como para demandar un gran número procesadores y también que la memoria distribuida incrementa la dificultad de programación ya que el programador o el compilador deben decidir cómo organizar los datos en los módulos de memoria de modo de reducir la comunicación. Algo que se puede afirmar sin temor a equivocarse es que el tipo de aplicación que se pretenda ejecutar tendrá una gravitación importante en el modelo de computador paralelo que pudiera resultar más apropiado y por otro lado que las dificultades que presenta la programación paralela son tales que se va a requerir no sólo gran entrenamiento y preparación de los futuros programadores en ese sentido sino una extraordinaria adaptación de los compiladores para poder explotar todas las ventajas del paralelismo.

# 2. Jerarquía de Memoria

## 2.1. Introducción

Según Hennessy & Patterson *"...el rendimiento de un programa en multiprocesador depende de la eficiencia del sistema para compartir los datos..."*. Vamos a abordar la metodología a través de la cual un equipo dotado de N procesadores (CPU), cada uno con su respectiva memoria cache (MC), y una única memoria principal (MP) administra esos recursos a fin de lograr que las aplicaciones corran "repartidas" entre las N unidades CPU-MC, compartiendo un único espacio de datos en la MP. Para ello analizaremos en primer término los distintos modos de operación de cada una de las N unidades CPU-MC-MP es decir las pautas que rigen ese modelo (jerarquía de memoria), tal como si se tratara de una unidad monoprocesadora.
Aunque este criterio en algunos casos pueda sufrir modificaciones, una MC guardará coherencia con su correspondiente MP siempre que, considerando el espacio de cualquiera de las dos memorias dividido en una cantidad entera bloques o líneas, toda línea que esté presente en la MC esté también en la MP. Los bloques (o líneas) se irán subiendo desde la MP hacia la MC a medida que vayan siendo requeridos por la CPU, explotándose así los principios de localidad temporal y espacial que dicen que los elementos referenciados alguna vez por un programa tenderán a ser referenciados nuevamente "pronto" y que al referenciarse un elemento determinado cobran alta probabilidad de ser referenciados los ubicados físicamente "próximos" al mismo. Así es que toda referencia a memoria realizada por la CPU será dirigida primeramente a la MC, sea esta una lectura o una escritura. Si se intenta leer un dato y la línea que lo contiene está en la MC de allí se toma y eso constituye un éxito de lectura (*read hit*), caso contrario se produce un fallo de lectura (*read miss*). (Veremos más adelante situaciones en las que se producen fallos de lectura aún cuando la línea esté en la MC). Ante un fallo de lectura el recurso inevitable y costoso, de ahí que deba evitarse, es llevar el dato a la CPU desde la MP, previa escritura de la línea que lo contiene en la MC (recordar principios de localidad). ¿Qué sucede en las escrituras?, cuando la CPU intenta escribir en una dirección de memoria (dato), también procura hacerlo primeramente en la MC y lo hace si la línea que contiene esa dirección ha sido previamente subida a la misma, eso sería un éxito de

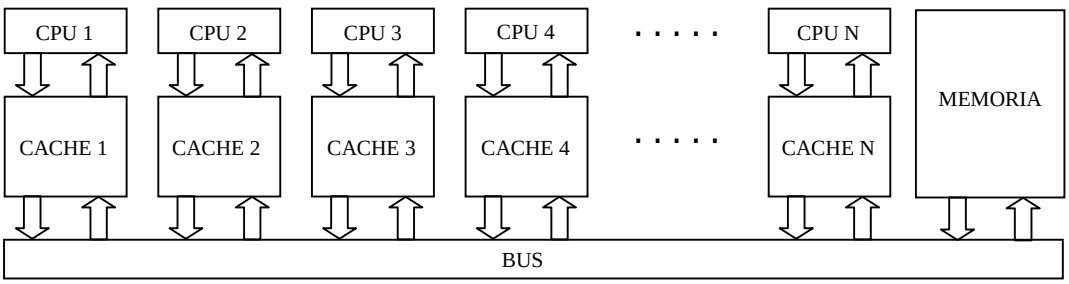
escritura (*write hit*). Acto seguido existen dos posibilidades, bajar inmediatamente la línea modificada a la MP, escritura directa (*write through*) o esperar y no bajarla hasta que algún otro proceso intente borrarla de la MC, post escritura (*write back*). ¿Cuándo y por qué habría de ser borrada una línea de la MC? en términos gruesos la MP es voluminosa y lenta mientras que la MC es reducida y rápida, se procura entonces que la MC albergue solamente los datos frecuentemente referenciados y si la política de ubicación de líneas en ella es buena, esos datos permanecerán en MC durante un tiempo razonablemente prolongado, pero la importante diferencia de tamaño entre ambas memorias hace que permanentemente estén subiendo líneas desde la MP y que consecuentemente algunas de la MC vayan siendo reemplazadas. Básicamente hay tres políticas de reemplazo para cuando una línea subida desde MP debe alojarse en MC, *totalmente asociativa*, si la línea puede ubicarse en cualquier posición de la MC, *correspondencia directa*, si la línea subida sólo puede ir a una única posición (por ejemplo a través de una función de hashing aplicada a la dirección original) y *asociativa por conjuntos*, que sería una situación intermedia en la que la línea subida puede ir a cualquier posición de un conjunto perfectamente determinado. Recordando que estábamos intentando escribir sobre una posición de memoria, queda por último la posibilidad de que la línea sobre la que se intente escribir no esté en la MC, eso es un fallo de escritura (*write miss*). En este caso la CPU puede proceder de dos formas, bajar hasta la MP y escribir directamente sobre ella, acto seguido o bien subir inmediatamente la línea modificada a la MC, modalidad conocida como "ubicar en escritura" o bien no hacerlo, "no ubicar en escritura"; o bien la MC fallada puede pedir primeramente la línea a la MP y luego escribirla (ya se verá la utilidad de esta opción en el protocolo de Berkeley en el que la línea solicitada podría obtenerse de otra MC y no necesariamente de MP).

| | **Memoria Caché** | **Memoria Principal** |
|---|---|---|
| Tamaño Total | 1 a 256 Kb | 4 a 2.048 Mb |
| Tamaño de Bloque | 4 a 128 bytes | 512 a 8.192 bytes |
| Tiempo de Acierto | 1 a 4 ciclos de reloj | 1 a 10 ciclos de reloj |
| Penalización de Fallo [TA]+[TT] | 8 a 32 ciclos de reloj | 100.000 a 600.000 ciclos de reloj |
| Tiempo de Acceso [TA] | 6 a 10 ciclos de reloj | 100.000 a 500.000 ciclos de reloj |
| Tiempo de Transferencia [TT] | 2 a 22 ciclos de reloj | 10.000 a 100.000 ciclos de reloj |
| Frecuencia de Fallos | 1% a 20% | 0,00001% a 0,001% |

Rangos de valores típicos de parámetros clave de una Jerarquía de memoria

# 2.2. Multiprocesadores

En un esquema monoprocesador con sólo una CPU, una MC y una MP, el problema de coherencia se resuelve entonces estableciendo reglas de acuerdo a las pautas que se acaban de comentar. La política a aplicar resulta simple y permite pocas variantes. Pero en una arquitectura multiprocesador, digamos de N unidades CPU-MC con sólo una MP, el problema de los accesos a memoria se complica y surge una enorme variedad de soluciones. Téngase en cuenta que en general sobre cada unidad estará corriendo un proceso distinto, aún cuando entre todos conformen una única aplicación, generándose entonces una competencia por el acceso a las posiciones de memoria que habrá que manejar estableciendo políticas que garanticen tanto la sincronización como la validez y coherencia de los datos leídos y escritos.
Una de las posibles arquitecturas para implementar la jerarquía de memoria en un esquema multiprocesador es la que se conoce como Memoria Compartida sobre Bus Común. Cada unidad CPU-MC avanzará en su cómputo tanto como le sea posible prescindiendo de las demás. Una unidad de control que no se muestra en la figura se encargará de la administración del bus permitiendo a cada MC volcar información cuando así lo solicite, atendiendo también los pedidos de acceso a memoria, mientras que un puerto especial de cada MC estará leyendo el bus permanentemente.

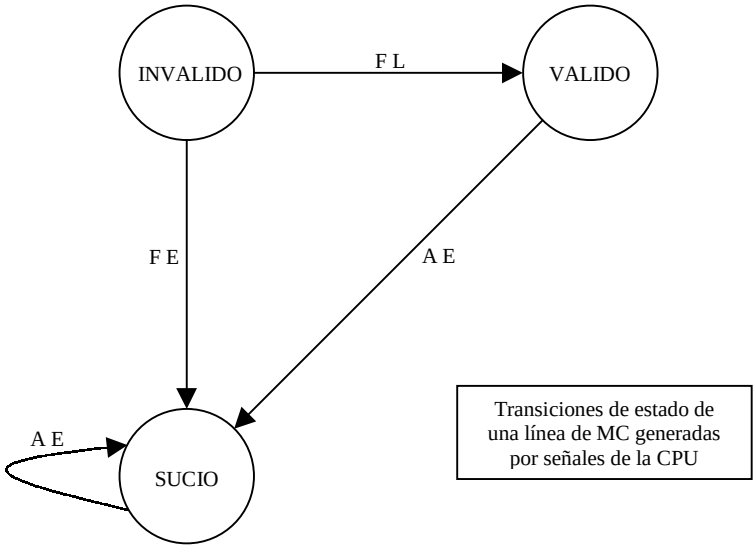


A diferencia de un sistema monoprocesador en el que la MC no tiene otro objetivo que acelerar el cómputo gracias a su tecnología de construcción, en sistemas multiprocesadores también tiene por finalidad distribuir, en la medida de lo posible, tanto datos como instrucciones entre las correspondientes unidades para evitar que la MP sea solicitada en forma alternada y continua por todas las CPUs, debiendo esperar cada una su turno mientras las demás van consiguiendo el acceso. Veamos esto con mas detalle en el siguiente ejemplo: imaginemos una MP (ideal) que permitiera accesos completos en sólo un ciclo de reloj. Esta solución neutralizaría la necesidad de MC en un sistema monoprocesador, nada podría haber más rápido que esta MP, pero ¿sucederá lo mismo en un sistema multiprocesador? Decididamente no, si se prescinde de las MC cada unidad verá reducida su velocidad de cómputo en un orden igual al número de unidades del sistema con relación a la velocidad a la que cada unidad monoprocesadora CPU-MC-MP es capaz de correr, velocidad a la cual una adecuada elección de MC podría acercar razonablemente al multiprocesador. (La no existencia de MC en un sistema multiprocesador debería independizar la velocidad de cómputo respecto a la cantidad de procesadores acercándola a la velocidad lograda con un único procesador o sea al sistema multitarea equivalente ya que las unidades no podrían trabajar simultáneamente. En la realidad esta tendencia se ve atenuada por la proporción de operaciones que la CPU realiza a nivel de registros, es decir independizada de la memoria).
Existen básicamente dos tipos de protocolo para mantener la coherencia entre los datos de MC y MP de Multiprocesadores (protocolos de coherencia cache), los basados en directorio, en los que cierta información sobre cada línea de la memoria se almacena en un espacio único (directorio o tabla) y los de espionaje (*snoopy*) en los que cada línea alojada en la MC contiene, junto con los datos, cierta información sobre esa línea (su validez, etc.) siendo esa información generada a partir de otra recogida mediante controladores de cada MC que al estar "espiando" (snooping) permanentemente el bus levantan información de interés cuando la detectan. Resumiendo, en los protocolos de directorio, el mismo alberga información del tipo: todas las cache que tienen copia de cada línea, la relación que existe entre el contenido de una línea de MC y su correspondiente en MP, etc., en tanto en los de espionaje toda esa información acompaña cada línea presente en la MC. A propósito, sería conveniente comentar la terminología empleada habitualmente para describir el estado de una línea de MC, VALIDA significa que tiene exactamente el mismo contenido que su correspondiente en MP, SUCIA que la información que contiene difiere de la que existe en su correspondiente en MP no obstante lo cual es válida (caso de post escritura en el que la línea de MC ha sufrido modificaciones por parte de la CPU pero todavía no ha sido desalojada) e INVALIDA cuando el contenido además de diferir del de su correspondiente en MP carece de validez.
Se puede percibir ya el objetivo de estos protocolos en el hecho de permitir a una CPU que en los accesos a memoria tenga exclusividad para escribir un objeto y que pueda disponer de la copia más actualizada al leer un objeto. Vamos a concentrarnos de aquí en más en los de tipo *snoopy* aclarando que las ideas son igualmente aplicables a los de directorio, salvo en la forma de gestionar la información sobre el estado de cada línea. Los problemas a resolver son del tipo: ¿qué deberán hacer todas las cache que tengan copia de una determinada línea cuando sobre alguna de ellas se haya protagonizado un fallo y haya sido necesario actualizarla?, ¿deberán actualizarla también?, ¿deberán invalidarla?, ... ¿qué deberá suceder cuando se escriba sobre una MC?, ... De aquí en más pasa a jugar un papel importante el modo de operación que tenga cada unidad CPU-MC-MP en cuanto a sí aplica escritura directa o post escritura, si ubica en escritura o si no lo hace, etc. y también todo lo que tenga que ver con las respuestas a las preguntas formuladas más arriba, generándose a partir de allí dos tipos de protocolo de espionaje: los de *invalidación en escritura* en los que cuando una CPU se dispone a escribir sobre su MC anuncia sobre el bus qué línea va a modificar para que todas las cache restantes chequeen si tienen o no esa línea y en caso afirmativo invaliden su contenido luego de lo cual se produce la escritura y los de *difusión o actualización en escritura*, en los que la CPU que va a escribir difunde sobre el bus no sólo la identificación de la línea a modificar sino también el nuevo dato, permitiendo a todas las MC que tengan esa línea, actualizarla. Habitualmente en la difusión en escritura los bloques se identifican como compartidos o privados según puedan existir copias en varias MC o en una sola; se suele aplicar en cada caso una modalidad distinta para difundir la información hacia las MC, en el caso de datos compartidos la mecánica es similar a la de escritura directa (*write through*) es decir la información se difunde ni bien es escrita en una de las MC mientras que para los datos privados se procede como en post escritura (*write back*), produciéndose la difusión sólo cuando una línea con información actualizada va a ser desalojada.
Como es de suponer la invalidación provocará una elevación en la tasa de fallos pero es un proceso muy rápido, en tanto la difusión es lenta pero tiende a mantener los fallos en un nivel bajo. También cabe observar que los datos fuertemente afectados de localidad para una unidad CPU-MC podrían no serlo para las demás por lo que de ningún modo está garantizada la utilidad de las actualizaciones en todos los casos.
Existe una infinita variedad de combinaciones de: capacidad de MC y de MP, tamaño de línea, latencia de las memorias, etc., y además todos los programas tienen distinto comportamiento y por ende diferente

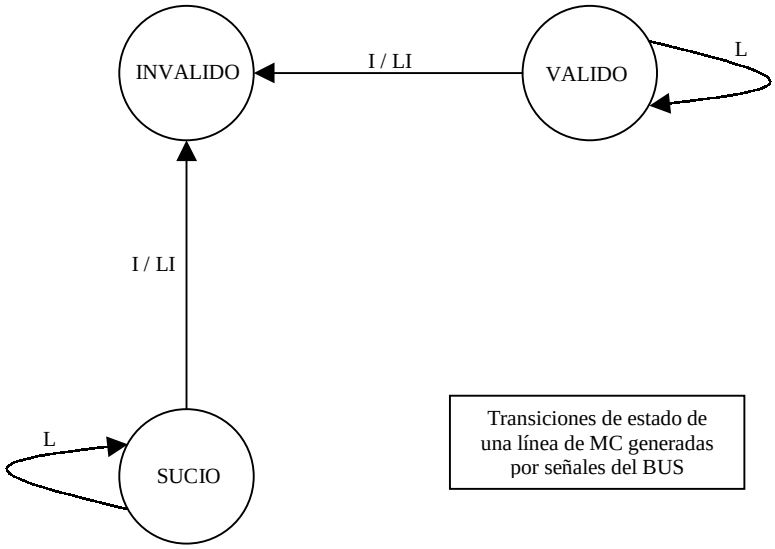
demanda de recursos. Como si todo esto fuera poco, en programas paralelos existe el componente adicional de la distribución de tareas (*scheduling*) entre las distintas unidades CPU-MC-MP lo que puede llegar a ocasionar variantes de performance para una misma aplicación en una misma arquitectura. Todos estos planteos contribuyen a señalar la simulación como casi único método viable para extraer conclusiones comparativas a partir de esta inmensa cantidad de variantes.

# 2.3. Protocolo de Invalidación basado en Post Escritura

Vamos a presentar seguidamente un sencillo protocolo de coherencia cache de invalidación en escritura basado en post escritura:



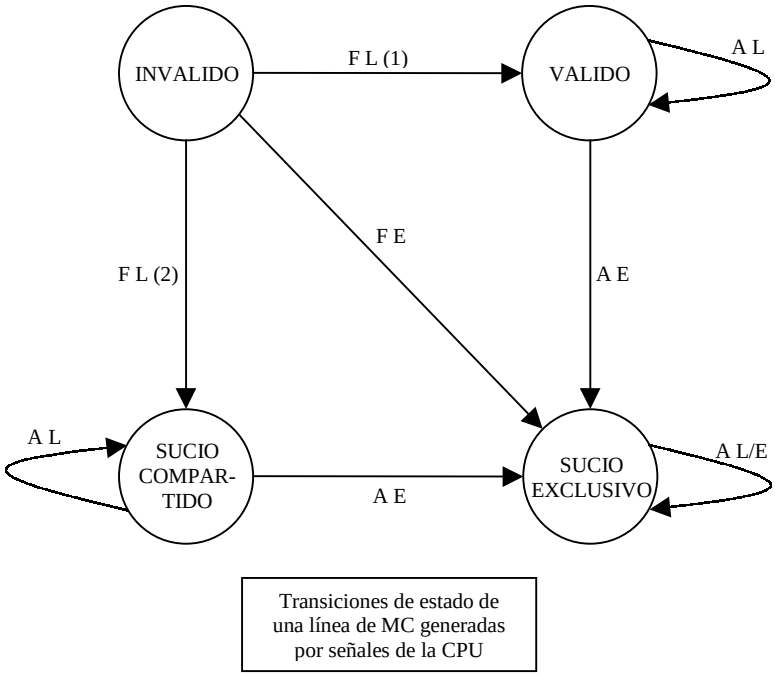

El diagrama de transición de estados finitos se explica por sí solo. Representa todas las posibles evoluciones del estado de una línea de MC. F y A indican fallo o acierto mientras que L y E, lectura y escritura. No hemos tomado en cuenta la entrada A L porque nunca produce cambio de estado.
Un rápido análisis muestra un hecho evidente, que todas las escrituras sean éstas provenientes de F o A llevan la línea al estado SUCIO mientras que las lecturas (recordemos que sólo consideramos fallos) la conducen al estado VALIDO. Esta mecánica guarda estrecha relación con el modo de post escritura de este protocolo. Al estado VALIDO se lo suele llamar también estado de SOLO LECTURA porque en él se permanece ante sucesivas lecturas, mientras que una escritura hace que se salga de él. Del mismo modo se acostumbra llamar LECTURA/ESCRITURA al estado SUCIO porque ante una sucesión de A sean éstos de lectura o de escritura no se produce conmutación.
Como se puede observar las señales de entrada al diagrama, es decir las que provocan los cambios de estado, son en todos los casos acciones de la CPU. Pero el estado de una línea de MC puede conmutar no sólo en respuesta a una acción de la CPU a la que pertenece, sino también a partir de señales de otras MC que reciba a través del bus, por ejemplo una orden de invalidación (recuérdese que hay un controlador por cada MC "espiando" el bus en procura de detectar señales que la tengan por destinataria). Es por eso que se acostumbra acompañar otro diagrama de transición de estados vinculado a estas señales simplemente por una cuestión de claridad porque en rigor un protocolo quedará completamente determinado mediante un único diagrama producto de la superposición de los dos que estamos presentando.
Hay tres posibles señales que una línea de MC puede detectar que otra MC le dirige a través del bus; L: lectura, cuando otra MC intenta leer su copia de esa línea, I: invalidación, cuando otra MC ordena la invalidación su copia de la línea y LI: lectura+invalidación, cuando se ordena llevar a cabo las dos tareas (las ordenes de invalidar son volcadas al bus sólo cuando una línea es escrita). En este protocolo informar que una copia ha sido leída en otra MC no presenta utilidad, lo que sucede es que todos los accesos son notificados al bus (sean estos lecturas o escrituras) siendo indispensable identificar el tipo de operación realizada.
A diferencia de este protocolo, uno de invalidación pero basado en escritura directa tendría sólo dos estados (VALIDO e INVALIDO), la aparición del tercer estado (SUCIO) es producto de la política de post escritura. El análisis sería similar, de hecho más simple.




En los protocolos de tipo *snoopy* el estado de cada línea, como ya se dijo, está representado por algunos bits que a ese fin se adicionan a la misma. Adecuando la lógica de un procesador es posible dar cualquier significado a esos bits haciendo que la coherencia sea garantizada por la política que cada uno considere más adecuada, generándose así una diversidad infinita de protocolos.

# 2.4. Protocolo de Berkeley

El protocolo de Berkeley, al igual que el protocolo elemental que se acaba de presentar, opera mediante invalidación en escritura y post escritura pero implementa dos mejoras importantes, producto de la descomposición del estado SUCIO en dos, capaces de representar cualidades muy útiles: la **propiedad** y la **exclusividad** sobre las líneas. La MC que tenga la propiedad sobre una línea será la encargada de asistir los pedidos de actualización de otras MC detectados sobre el bus.
En el arranque todas las líneas serán propiedad de la MP y a medida que la CPU las vaya escribiendo, la MC escrita irá adquiriendo la propiedad sobre la o las líneas nuevas y así sucesivamente, es decir la MC más recientemente actualizada será la propietaria de la línea modificada, resultando así no más de una propietaria por línea (recordar que se opera por invalidación).



Ante un fallo en cualquier MC, la propietaria de la línea sobre la que se produzca el fallo (si existe) se encargará de proveer la línea a la MC fallada, ahorrándole el costoso trabajo de ir a buscarla a MP. Obviamente esta asistencia entre cachés reduce notablemente el trafico MP-MC haciendo decrecer la penalización en aproximadamente un orden de magnitud.
Las líneas en estado VALIDO o INVALIDO no tienen "dueño", solamente determinan alguna forma de propiedad los estados SUCIOS; en ellos una línea puede pertenecer a una MC en forma exclusiva o compartida. Si sólo una MC cuenta con una línea, la propiedad será exclusiva, si alguna otra la posee la propiedad será compartida. La ventaja de que el estado indique el tipo de propiedad está en el hecho que cuando se escriben líneas de propiedad exclusiva no se vuelca al bus la señal de invalidación para las demás MC lográndose una disminución de tráfico en el bus.
Como se dijo, una línea de MC en estado SUCIO EXCLUSIVO aloja información actualizada, que no está presente en ninguna línea de ninguna otra MC y que difiere de su correspondiente línea en MP. El estado SUCIO COMPARTIDO indica que la información es actualizada y diferente de la correspondiente en MP pero que está en más de una MC.
Veamos dos cosas para cada uno de los cuatro estados: (a) por qué vía puede ser alcanzado y (b) cuales son las posibles alternativas para salir del mismo.

INVALIDO: (a) cuando la línea existe en la MC y detecta una señal de invalidación con su dirección en el bus. (b) ante un F L la línea o bien es subida desde MP y el estado pasado a VALIDO o bien es traída desde otra MC (la "dueña") y el estado pasado a SUCIO COMPARTIDO. Ante un F E lo más adecuado en este protocolo es primeramente pedir la línea, modificarla y luego escribirla en MC ¿por qué hacer esto en lugar de bajar hasta MP, escribirla y luego subirla hasta la MC invalidando las demás copias? por la sencilla razón que es muy factible, aunque no seguro, que exista otra MC en condiciones de asistir la lectura ahorrándose muy probablemente un acceso a MP. Por eso un F E lleva la línea a estado SUCIO EXCLUSIVO.

VALIDO: (a) toda vez que se produzca un F L (lo cual únicamente puede suceder en el estado INVALIDO) y que éste sea asistido desde MP y no desde otra MC. (b) Por supuesto que en este estado no se conciben fallos, los A L no afectarán el estado de la línea mientras que los A E la llevarán a SUCIO EXCLUSIVO (recordar criterio de post escritura).

SUCIO EXCLUSIVO: (a) desde los estados VALIDO e INVALIDO según lo visto en los ítems -b- de los dos puntos anteriores. (b) Sucesivos aciertos de L y/o E no harán cambiar de estado a la línea, mientras que fallos no son posibles en este estado. Luego es imposible salir de este estado, la única posibilidad de cambio está en el reemplazo de la línea.

SUCIO COMPARTIDO: (a) cuando hay un F L en una línea en estado INVALIDO y ésta es asistida por otra MC, luego de la asistencia tanto la asistida como la asistente van a quedar con la misma línea, por ende en el mismo estado, es decir el estado de la asistente, SUCIO COMPARTIDO. (b) En este estado no puede haber fallos, A L no lo modifican, sólo un A E porque le quita el carácter de COMPARTIDO para pasarlo a EXCLUSIVO.

Esta es la forma de analizar el diagrama que de por sí es autoexplicativo y sobre el que también cabe la descomposición en dos según el origen de las señales consideradas, por lo que se muestra seguidamente la parte correspondiente a señales de entrada provenientes del bus. Aquí también hay tres posibles señales que una línea de MC puede detectar que otra MC le dirige a través del bus, recordemos; L: lectura, cuando otra MC intenta leer esa línea, I: invalidación, cuando otra MC ordena la invalidación de la línea y LI: lectura+invalidación, cuando se ordena llevar a cabo las dos tareas (las ordenes de invalidar son volcadas al bus sólo cuando una línea es escrita). Cabe recalcar una vez más que no estamos frente a un segundo diagrama de transición de estados sino a uno único descompuesto en dos para facilitar su interpretación, la forma de arribar a cada estado en este esquema es, obviamente, la misma que en el anterior. Sólo cabe comentar las transiciones posibles desde cada uno.

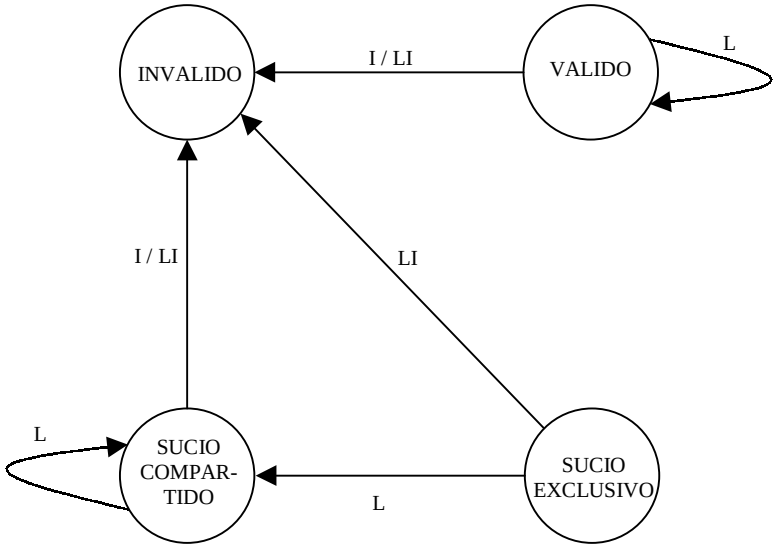
INVALIDO: en este estado una línea no puede ser destinataria de ninguna señal por parte de otra MC pues la información que contiene no es de utilidad para ser leída por nadie ni tiene sentido de ser invalidada puesto que ya lo está.

VALIDO: si otra MC protagoniza un F L, pide a través del bus la línea y al existir la misma en estado VALIDO la lee quedando luego ambas en este estado. Pero si la otra MC ha sufrido un F E, también va a

pedir la línea al lugar dónde está en estado VALIDO, pero acto seguido tendrá que invalidar la línea leída porque la va a escribir dejándola en estado SUCIO EXCLUSIVO.

SUCIO EXCLUSIVO: en este estado nunca se puede recibir un pedido de invalidación porque SUCIO EXCLUSIVO implica la línea más actualizada de todo el sistema. Sólo tienen sentidos pedidos de L o LI.

SUCIO COMPARTIDO: Al igual que VALIDO, puede recibir cualquiera de las tres señales, permaneciendo sin cambio si se le hace una lectura y pasando a INVALIDO en cualquiera de las dos variantes de invalidación.




Transiciones de estado de una línea de MC generadas por señales del BUS

# 2.5. Otros protocolos

Como se dijo, existe una infinita diversidad de variantes para implementar las ideas expuestas hasta aquí bajo la forma de protocolos de coherencia caché. Vamos a comentar brevemente las ideas fundamentales de los más conocidos.

*Protocolo de Illinois*. Se basa en los mismos principios que el de Berkeley (invalidación en escritura basado en post escritura), extendiéndose algunos de los conceptos, por ejemplo el de propiedad, permitiendo la existencia de mas de un dueño para una única línea y estableciendo un orden de prioridad para que respetando ese orden cualquiera de los dueños asista un pedido de actualización de otra MC. También se procede no difundiendo señal de invalidación si la línea afectada no es compartida.

*Protocolo Firefly*. A diferencia de Berkeley e Illinois, Firefly es un protocolo de difusión en escritura por lo que no cuenta con estado INVALIDO ya que, prescindiendo del arranque, nunca una línea puede alcanzar ese estado. En cuanto a la asistencia entre MC maneja un criterio similar al de Illinois sólo que no se designa la unidad asistente de acuerdo a un orden de prioridad sino que permite que todas las MC en condiciones de asistir lo hagan simultáneamente. Si bien esto último redunda en una simplificación, ocasiona una elevación en el número de bloqueos de procesadores.

*Protocolo Dragon*. Así como Firefly lleva las ideas de Illinois al terreno de la difusión en escritura, Dragon hace lo propio con Berkeley. No tiene estado INVALIDO y retoma la idea de propiedad haciendo que sólo el dueño de una línea pueda asistir a otra MC y eventualmente la MP cuando no haya otro dueño.

No existe una definición terminante en cuanto a cual de las modalidades es mejor, sí invalidación o difusión. Esta cuestión está fuertemente influenciada por: la aplicación que se quiera ejecutar, la cantidad de procesadores, el criterio para la distribución de tareas entre los mismos, etc. Se han desarrollado por

eso modificaciones en la idea de recoger lo mejor de cada modalidad. Vamos a comentar dos políticas que pueden ser aplicadas para modificar las versiones clásicas de los protocolos presentados hasta ahora dando origen a sus versiones híbridas.

*Read Broadcast (difusión de lectura)*. El punto débil de los protocolos de invalidación está en la alta tasa de fallos por invalidación, es decir fallos en los que incurre una CPU cuando al acceder a la línea buscada la encuentra pero ve que ha sido previamente invalidada por otra MC. El objetivo de este método es reducir esa tasa de fallos. Para ello procede de la siguiente manera: cuando una MC "detecta" que otra MC está leyendo una línea cuya copia posee en estado inválido, aprovecha para actualizarla, aún cuando de momento no la necesite. Esto constituye una especie de "difusión o actualización" no de escritura sino de lectura (*read Broadcast*) por lo que la aplicación de este método a protocolos de invalidación como Berkeley o Illinois da origen a sus respectivas versiones híbridas.

*Snooping Competitivo*. El punto débil de los protocolos de difusión es la excesiva utilización del bus y los consecuentes bloqueos de CPU debidos a actualizaciones innecesarias. Para solucionar este problema se propone lo siguiente: una línea de MC será invalidada luego de que haya sido actualizada un número "grande" de veces sin haber sido solicitada por la CPU. La forma de implementar esta idea es mediante el agregado de un contador por cada línea de MC de manera que cuando la CPU la referencie, el contador sea seteado a su máximo valor, con cada actualización el contador se decremente en uno y al llegar a cero la línea sea invalidada. En teoría Firefly y Dragon deberían tornarse más flexibles mediante este agregado que da origen a sus versiones híbridas. No obstante, al igual que en Read Broadcast, existe siempre una gran dependencia de factores externos que torna imposible determinar el mejor modelo, lo adecuado es ensayarlos, y como ya se anticipó nada mejor que simularlos, para determinar las condiciones bajo las cuales cada uno ofrece la mejor solución.

# 3. Sincronización

## 3.1. Introducción

Cuando un programa paralelo arranca, lo hace en una de las unidades CPU-MC-MP de la máquina multiprocesadora, tal como lo hacen los programas secuenciales en máquinas monoprocesadoras. De ahí en más los procesos que se van lanzando en distintas unidades quedan corriendo simultáneamente en tanto no se produzcan situaciones de conflicto. Mientras el cómputo de cada uno pueda evolucionar sólo mediante accesos a posiciones de memoria que no sean requeridas por más de una unidad a la vez, y mientras ningún tramo de proceso requiera la condición de no ser interrumpido, podríamos decir que los procesos no se "molestarán" entre sí y podrán evolucionar incluso hasta completarse. Al igual que en máquinas monoprocesadoras con Sistemas Operativos (SO) multitarea, en máquinas multiprocesadoras también existe la posibilidad de que se quieran correr más procesos que la cantidad total de CPUs, en cuyo caso aun cuando determinados procesos no se afecten mutuamente ni tengan problemas de sincronización, también podría darse la necesidad de interrumpirlos e incluso de removerlos (transitoriamente) de sus respectivas unidades. De manera que la "manipulación" de procesos en un sistema multiprocesador (lanzamiento, remoción, bloqueo, etc.) es muy similar a la de una máquina monoprocesadora con SO multitarea, con la única diferencia (en cuanto a sincronización se refiere) en la disponibilidad de CPUs para distribuir procesos y con la consecuente posibilidad, por supuesto, de que varios de ellos puedan efectivamente estar en ejecución simultáneamente.
El tema que vamos a abordar tiene que ver con la forma de evitar que uno o más procesos (en general corriendo en distintas unidades) se solapen en la escala de tiempo precisamente en segmentos que contengan demandas sobre los mismos recursos (por ejemplo la o las mismas variables globales, la misma unidad de Entrada/Salida, etc.). Un caso elemental de este tipo de conflicto lo constituye el intento simultáneo, por parte de más de una CPU, de acceder a la misma posición de MP.
En general, mediante mecanismos de hardware, cada vez que una CPU consigue el acceso a MP a través del bus, todas las demás ven bloqueada esa posibilidad. El bus actúa entonces como uno de los medios para a arbitrar sobre los múltiples intentos de acceder a la MP, al menos durante los ciclos que ese acceso lo mantiene ocupado. Esta es una facilidad que el hardware debe ofrecer inevitablemente ya que por ejemplo una memoria, del tipo que sea, ¡no puede ser direccionada por más de una dirección a la vez!. Ilustremos estas ideas mediante un ejemplo suponiendo que una CPU intenta incrementar en 1 el valor de una variable global residente en la MP. Hay procesadores cuyo lenguaje ensamblador cuenta con una única instrucción para esa tarea, lo cual ni remotamente significa que puede ser ejecutada en un ciclo, ni

en dos... Digamos que podría ser necesario para ello, por ejemplo, leer el valor almacenado, subirlo hasta un registro de la CPU, sumarle 1 y recién entonces procurar escribirlo en la misma posición desde la que fue leído. ¿Qué sucedería si otro proceso que hubiera conseguido el acceso a MP desde otra CPU leyera el valor a incrementar inmediatamente después de que lo haya hecho el proceso que lo iba a incrementar?, al decir "inmediatamente después" me refiero a "antes de que el incremento haya sido realizado", porque la CPU que intentaba incrementar tendrá la exclusividad sobre el bus para leer, perdiéndola luego al iniciar los cómputos siguientes e intentando recuperarla recién para escribir el valor incrementado. Aquí hay una cuestión importantísima a destacar, si el segundo proceso, el que sólo iba a leer, accede a la variable en un instante cronológicamente posterior a aquel en el que lo hizo el que intentaba incrementar, eso significa que precisamente en ese orden fueron "pensadas" ambas tareas por el programador (o por el compilador) y que por lo tanto el segundo proceso debería leer el valor incrementado de la variable. ¡No sucederá así por supuesto! a menos que se implemente algún mecanismo corrector.
Este ejemplo que pone de manifiesto un problema sobre una única instrucción, puede generalizarse a tramos completos de código. Estos tramos se denominan *Zona Crítica* (ZC) y pueden definirse de la siguiente manera: una ZC es un segmento de código que debe ser ejecutado por una única unidad a la vez y cuya ejecución una vez iniciada debe completarse sin interrupciones. Por supuesto y aunque no se mencione en la definición, queda claro que el carácter crítico va indivisiblemente ligado al hecho de que ese segmento de código ha de operar sobre recursos de carácter global, aunque ésta condición no sea excluyente; un programador podrá decidir proteger un tramo de programa dándole el carácter de ZC cualesquiera sean las características de los recursos requeridos por el mismo. Lo que debe quedar claro es que, en multiprocesadores, proteger una ZC no es sino darle a la CPU que la va a ejecutar la exclusividad sobre la MP durante los ciclos que le tome esa ejecución.
Este tipo de problema conduce a la necesidad de implementar algo así como mecanismos de protección y dado que para proteger una ZC de un proceso a veces es necesario detener otros (ya se verá cómo) hasta que haya sido ejecutada la ZC protegida, la resultante en la escala de tiempo es una especie de reacomodamiento relativo entre los procesos tendiente a garantizar el orden previsto por la programación. En esto consiste la sincronización.

# 3.2. Spin Lock

### 3.2.1. Spin Lock Binario

Este mecanismo (netamente de software) se basa en la utilización de una única variable global compartida por todos los procesos que tengan ZCs que proteger. Inicialmente esta variable se setea en el valor 0 que representa desbloqueo (**unlock**). El proceso que logra leer ese 0 es autorizado a entrar en su ZC no sin que antes setee el valor 1 en la variable, lo que representará bloqueo (**lock**) para todos los demás procesos que la estén leyendo o intentando leer. Esas lecturas son realizadas por los procesos interesados desde loops que los mantienen "dando vueltas" (**spinning**) hasta que consiguen leer el valor 0.

```
var x:(0,1)                    /Se declara la variable global x en el dominio (0,1)/
x:=0                           /Se inicializa en 0/
Proceso Pi i=1,2,...,n
   repetir
     :                         /Se inserta el loop con espera ocupada justo antes de la ZC/
   hasta x:=0
   x:=1                        /Se bloquea el acceso a su ZC a los demás Procesos/
     :                         /ZC/
   x:=0                        /Se libera el acceso al próximo proceso en leer la variable x/
```

Este mecanismo se apoya en lo que se conoce como protocolo de espera ocupada (*busy-wait protocol*) porque los procesos que aguardan para ingresar a su ZC permanecen corriendo, manteniendo ocupadas sus respectivas CPUs, intentando superar el bloqueo continuamente. Este protocolo tiene la ventaja de que el proceso que aguarda por el objeto compartido (en este caso la MP para poder ejecutar su ZC) tiene una respuesta inmediata ni bien el mismo queda disponible, pero el inconveniente es que sigue consumiendo recursos del sistema mientras los loops (*spinning*) están corriendo.
Su contrapartida es el protocolo de espera durmiendo (*sleep-wait protocol*). En este modo de operación los procesos son removidos de sus respectivas unidades y colocados en colas de espera (a dormir). Los procesos suspendidos deben ser notificados de los eventos por los que están aguardando, pero mientras están durmiendo liberan una cantidad importante de recursos, por ejemplo ¡la unidad completa! para que

comience a correr en ella otro proceso. En sistemas multiprocesadores el incremento de complejidad del hardware para implementar protocolos de espera durmiendo es importante frente a los de espera ocupada. En particular el mecanismo **spin lock** se aplica preferentemente con protocolos de espera ocupada.

### 3.2.2. Spin Lock generalizado con n posibles valores

El Spin Lock binario deja en manos del mecanismo arbitrador del bus la prioridad para que el recurso compartido sea asignado. Cada vez que la variable x toma el valor 0 es probable que muchas unidades pretendan leerla simultáneamente. La concesión del bus para poder llevar acabo la lectura es resuelta a favor de una determinada unidad y en detrimento de las demás mediante un mecanismo propio del sistema multiprocesador, que por supuesto deberá contemplar el orden cronológico de las solicitudes para garantizar la sincronización, pero que no podrá ser forzado por el programador. Si la intención es conceder los bloqueos según un orden preestablecido, el programador deberá apelar a una solución como la siguiente,

```
var x:(1,2,...,n)              /Se declara la variable global x en el dominio (1,2,...,n)/
x:=1                           /Se concede el bloqueo al Proceso 1/
Proceso Pi i=1,2,...,n-1
   repetir
     :                         /Se inserta el loop con espera ocupada justo antes de la ZC/
   hasta x:=i
     :                         /Mientras Proceso i esté dentro de su ZC los demás Procesos
     :                          ven bloqueado el acceso a la suya/
   x:=i+1                      /Se libera el acceso al Proceso siguiente/

Proceso Pn
   repetir
     :
   hasta x:=n
     :
     :
   x:=1                        /Se libera el acceso al primer Proceso/
```

Este mecanismo garantiza exclusión mutua al costo de un incremento en los tiempos de espera mientras el bloqueo se va desplazando entre los n procesos ordenados circularmente. Los procesos deberán esperar ("su turno") aún en caso de no haber conflictos por requerimientos simultáneos y si uno quedara bloqueado todos los demás se bloquearían también.

### 3.2.3. Test&Set

Para que los algoritmos mostrados más arriba tengan un funcionamiento efectivo deberán ser adecuados al hardware del sistema entre otras cosas en lo siguiente: desde que en la variable de bloqueo x es leído el valor que autoriza el ingreso a la ZC y hasta que en esa variable es seteado el valor que impide ese ingreso al resto de los procesos, la unidad a la que en principio le ha sido concedido el bloqueo debe ejecutar varias operaciones (ver Introducción) pudiendo suceder que la variable de bloqueo sea nuevamente leída antes de que le haya sido seteado el valor de protección. Una alternativa para ese problema es la operación (de hardware) Test&Set que permite ejecutar en forma atómica estas acciones,

```
Test&Set(x,yi):
   <yi:=x ; x:=1>
```

$y_i$ es un variable local de cada proceso que ha de quedar con el mismo valor de la variable x al cabo de la ejecución de la primitiva Test&Set. Paralelamente y sin que ningún otro proceso la pueda acceder, la variable x tomará el valor 1 antes de completarse la ejecución de Test&Set. La resultante es una ejecución atómica de una secuencia de dos operaciones que, de otro modo, puede resultar interferida por otro proceso con graves consecuencias para la sincronización. Será necesario entonces reformular los algoritmos anteriores incorporando la mejora provista por este mecanismo. El caso de n procesos entre los que sólo se pretende exclusión mutua no presenta inconvenientes y la aplicación de la primitiva Test&Set es directa. En el caso que los n procesos se quieren sincronizar forzando un orden circular preestablecido aparece un pequeño inconveniente en el hecho que la primitiva Test&Set tal como se presentó fuerza a 1

el valor de x una vez ejecutada. El valor 1 significaba bloqueo del acceso a su ZC para todos los procesos pero ahora, en caso de mantenerse la convención del problema resuelto mediante spin lock, significaría desbloqueo para el proceso $P_1$. La solución propuesta pasa por: reservar el valor x=1 para bloqueo a todos los procesos y dar el acceso al proceso $P_i$ cuando x=i+1 (en lugar de x=i).

```
var x:(0,1)
x:=0

Proceso Pi i=1,2,...,n
   var yi:(0,1)
   repetir
     Test&Set(x,yi)
   hasta yi:=0
     :
     :
   x:=0
```

```
var x:(1,2,...,n,n+1)
x:=2

Proceso Pi i=1,2,...,n-1
   var yi:(0,1)
   repetir
      Test&Set(x,yi)
   hasta yi:=i+1
     :
     :
   x:=yi+1

Proceso Pn
   var yn:(0,1)
   repetir
      Test&Set(x,yn)
   hasta yn:=n+1
     :
     :
   x:=2
```

Distintos fabricantes dan a este tipo de operación nombres como: Fetch&Add, Compare&Swap,... Y además otros mecanismos similares son provistos para operaciones de carácter atómico tanto en *lecturas*, *escrituras*, *lectura-modificación-escritura*, etc. Veamos entonces los algoritmos,

# 3.3. Semáforos

Los semáforos son un medio para implementar sincronización aplicado en sistemas con protocolos de espera durmiendo. Los hay de dos tipos, binario y de conteo.

### 3.3.1. Semáforos binarios

Un semáforo binario es una variable booleana (s) capaz de tomar los valores 0 o 1. Cada recurso compartido (en particular las ZCs) puede asociarse a un semáforo. Luego de la declaración e inicialización sólo dos operaciones primitivas (de carácter atómico) pueden ejecutarse sobre s,

- P(s) resta 1 al valor de s en tanto y en cuanto el valor de s no sea 0.
- V(s) suma 1 al valor de s en tanto y en cuanto el valor de s no sea 1.

A través de una adecuada convención, los semáforos binarios se corresponden en esencia con las variables de bloqueo (lock) empleadas en protocolos de espera ocupada para resolver problemas de exclusión mutua. Veamos entonces la aplicación de este mecanismo al problema resuelto anteriormente mediante spin lock, aceptando para ello una convención adicional: un intento de P(s) cuando s=0 pondrá a dormir al proceso que haya hecho el intento, mientras que toda operación V(s) despertará al proceso que esté al tope de la cola de procesos durmiendo en tanto ésta no esté vacía.

```
semáforo s:                    /Se declara la variable global s de tipo semáforo/
s:=1                           /Se inicializa en 1/
Proceso Pi i=1,2,...,n
   P(s)
    :                          /ZC/
   V(s)
```

El primer proceso (digamos $P_k$) que consigue ejecutar P(s) accede inmediatamente a su ZC. Todos los demás procesos que intenten hacer P(s) mientras está en ejecución la ZC de $P_k$ se encuentran con s=0 y se ponen a dormir. Cuando $P_k$ ejecuta V(s) despierta al primer proceso de la cola (si lo hubiese) y lo pone en estado *listo*, o sea en condiciones de ejecutar inmediatamente su P(s).

### 3.3.2. Semáforos de conteo

En el caso de semáforos de conteo la variable s ya no es booleana sino entera no-negativa y capaz de tomar los (n+1) valores: 0,1,2,....,n. Un semáforo de conteo se comporta como una colección de n permisos que actúan sobre otras tantas copias de un recurso compartido (un dato, una variable, un array, una ZC, etc.) de modo que permite atender cada pedido del recurso concediendo permiso sobre el mismo hasta tanto todas sus copias hayan sido concedidas. Formalmente se definen también dos operaciones primitivas y de carácter atómico,

- P(s) resta 1 al valor de s en tanto y en cuanto s>0, caso contrario coloca el proceso a dormir.
- V(s) suma 1 al valor de s y eventualmente despierta un proceso dormido a la espera de ejecutar P(s).

Intuitivamente P(s) corresponde a un pedido de permiso, mientras que V(s) representa la devolución del permiso recibido una vez que se ha hecho uso de una copia del recurso compartido asociado con el semáforo. Todo segmento de código delimitado por el par de primitivas (P,V) resulta así imposible de ser interrumpido por otro proceso.
Supongamos por ejemplo que quisiéramos limitar a 3 el número máximo de procesos corriendo simultáneamente en nuestro equipo multiprocesador mediante el uso de un semáforo de conteo.

```
semáforo s:
s:=3
```

| T | Proceso i | Proceso j | Proceso k | Proceso l | s: |
|---|---|---|---|---|---|
| $t_0$ | | | | | s=3 |
| $t_1$ | | P(s) | | | s=2 |
| $t_2$ | P(s) | : | | | s=1 |
| $t_3$ | : | : | P(s) | | s=0 |
| $t_4$ | : | : | : | | |
| $t_5$ | : | : | : | | |
| $t_6$ | : | : | : | | |
| $t_7$ | : | : | : | P(s) | (a) |
| $t_8$ | : | : | : | : | |
| $t_9$ | V(s) | : | : | : | (b) |
| $t_{10}$ | | : | V(s) | : | |
| $t_{11}$ | | V(s) | | : | |

(a) en $t_7$ se intenta lanzar el Proceso l pero éste no puede ejecutar P(s) porque s=0 por lo tanto es puesto a dormir.
(b) en $t_9$ el Proceso i ejecuta su V(s) con dos consecuencias: se lleva el valor de s de 0 a 1 y se despierta al Proceso l para que ejecute su P(s) y comience a correr.

Veamos ahora una aplicación de semáforos sobre el clásico problema PRODUCTOR-CONSUMIDOR. Proceso PRODUCTOR es aquel que produce items que son demandados por un proceso CONSUMIDOR. Los items producidos son alojados en una cola o buffer en espera de ser requeridos de modo que no puede alojarse más que un número máximo de items y no pueden extraerse items si el buffer/cola está vacío. En general un conjunto de PRODUCTORES y otro de CONSUMIDORES (vinculados por el mismo tipo de item) estarán corriendo simultáneamente, en el mejor de los casos, en distintas unidades de procesamiento. Tanto la colocación como la extracción de items se compone de una cantidad de pasos que por motivos similares a los expuestos hasta aquí (ver Introducción) conviene que sean ejecutados sin ser interrumpidos por otro pedido de colocación/extracción, es decir merecen ser tratados como procesos de exclusión mutua y una forma de protegerlos es mediante un semáforo binario. Por otro lado cuando el buffer/cola está lleno se deben impedir intentos de colocación de items, para lo cual no es mala idea poner a dormir a los PRODUCTORES que soliciten colocar, hasta que se produzcan vacantes en el buffer/cola. Asimismo al vaciarse el buffer/cola, poner a dormir a los CONSUMIDORES que quieran extraer puede resultar una decisión muy útil. Dos semáforos de conteo pueden manejar esta última implementación.

```
semáforo s:,lleno:,vacío:
s:=1,lleno:=0,vacío:=N
```

/Se declaran e inicializan variables globales de tipo semáforo, s para la exclusión mutua de accesos al buffer/cola, lleno y vacío para limitar colocaciones y extracciones de items a la capacidad del buffer/cola/

```
Proceso Pi i=1,2,...,j          /j procesos PRODUCTORES/
   repetir
      producir(item)            /se crea un nuevo item/
      P(vacío)
      P(s)
      colocar(item)             /se coloca en el buffer/cola/
      V(s)
      V(lleno)
   siempre
Proceso Ci i=1,2,...,k          /k procesos CONSUMIDORES/
   repetir
      P(lleno)
      P(s)
      extraer(item)             /se extrae un item del buffer/cola/
      V(s)
      V(vacío)
      utilizar(item)            /se procesa el item/
   siempre
```

Colocaciones y extracciones están protegidas por los pares (P,V) sobre el semáforo s y por fuera de esos bloques (brindando un nivel adicional de protección) hay pares (P,V) sobre los semáforos lleno y vacío pero cruzados. La resultante de este mecanismo es la siguiente: el primer proceso que intente colocar llevará al semáforo vacío de (N) a (N-1) ejecutando la primitiva P(vacío) y consiguiendo así alcanzar la línea P(s). Como s está en 1, la llevará a cero y pasará a ejecutar la rutina colocar(item). Ningún proceso podrá mientras tanto intentar extraer porque al estar lleno en 0 P(lleno) es imposible y aún en el caso de lleno>0 la extracción sólo podrá tener lugar luego de ejecutado V(s) del proceso que está colocando. Por otro lado cuando se intente extraer habrá que superar la barrera P(lleno); como lleno se inicializó en 0, P(lleno) solo podrá ejecutarse cuando hubiese sucedido antes un V(lleno) es decir cuando algún item hubiese sido colocado con anterioridad.

# 3.4. Monitores

Los monitores son programas de alto nivel encargados de administrar los recursos compartidos. Son propios de los ambientes donde la programación es estructurada y se componen de una colección de variables compartidas y procedimientos asociados bajo una única estructura que da acceso los procesos para que ejecuten los procedimientos cuando los necesiten. Un monitor típico se compone de tres partes:

- Definición de su nombre y declaración de todas sus variables locales.
- Colección de procedimientos que harán uso de las variables locales declaradas (y de las variables globales de los procesos bajo la forma de argumentos).
- Inicialización de las variables locales declaradas.

La idea es que las ZCs de los procesos sean removidas de los mismos e implementadas a través de los procedimientos del monitor. La exclusión mutua se garantiza en el hecho que sólo un proceso por vez puede acceder a dichos procedimientos. La sincronización queda en manos de determinadas *primitivas de condición* que permiten la comunicación entre procedimientos dentro del mismo monitor. Por ejemplo:

- **wait(**condición**)** pone al proceso que activó el monitor a dormir y así lo deja hasta que sea despertado por una señal (*signal*) que le indique que la condición ha sido satisfecha.
- **signal(**condición**)** despierta a un proceso que hubiese estado durmiendo en espera que se satisfaga la condición y no hace nada en caso de no haber proceso durmiendo a la espera de condición.

El monitor no resulta entonces un proceso en sí mismo sino un módulo estático de datos y declaraciones de procedimientos que sólo se activa a solicitud de los verdaderos procesos, los que deben ser programados por separado. Veamos el problema PRODUCTOR-CONSUMIDOR resuelto mediante un monitor. suponiendo que se producen y se ofrecen al consumidor, números enteros.

```
Monitor PRODUCTOR-CONSUMUDOR  /Nombre del monitor/
   Buffer[1:N]:entero         /Declaración de variables. El Buffer es un array de enteros a
   No_lleno,No_vacío:bool      ser recorrido en forma circular, No_lleno y No_vacío indican
   In,Out:entero               el estado del Buffer, In y Out son punteros para colocar o
   Cont:índice                 extraer del Buffer y Cont un indicador de la cantidad de
                               items alojados en el Buffer/
```

```
   Procedimiento Colocar(item)
      comienzo
      if Cont=N then Wait(No_lleno)
      Buffer[In]:=item
      In:=(In+1) mod N
      Signal(No_vacío)
      fin


   Procedimiento Extraer(item)
      comienzo
      if Cont=0 then Wait(No_vacío)
      item:=Buffer[Out]
      Out:=(Out+1) mod N
      Signal(No_lleno)
      fin


   comienzo
      In:=0
      Out:=0
      Cont:0
   fin
```

No se explicita el modo de operación de la variable Cont pero debería ser valorizada mediante algún cálculo que contemple las posiciones relativas de los punteros In y Out. Tampoco se muestra la valorización de las variables No_lleno y No_vacío pero es muy claro su significado. Aceptando que el pseudo-código utilizado tiene connotaciones del Pascal, vale la pena remarcar el hecho que el programa monitor "no hace nada por sí mismo", atendiendo a que en el segmento donde iría el código del programa principal, es decir entre **comienzo** y **fin** (**begin** y **end** en Pascal) no hay más que inicialización de variables locales.
Hasta aquí, el monitor propiamente dicho y por último los procesos,

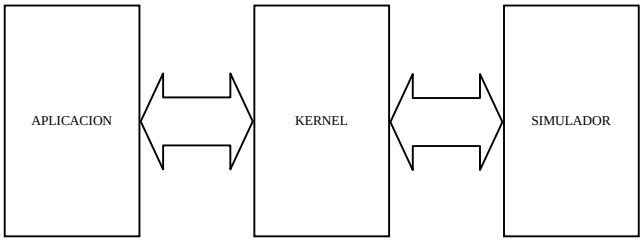
```
Productor
   item:entero
   comienzo
      Producir(item)
      Colocar(item)
   fin


Consumidor
   item:entero
   comienzo
      Extraer(item)
      Utilizar(item)
   fin
```

¿Cómo sería la operación de este mecanismo?, supongamos que el proceso Productor crea un item mediante el procedimiento Producir (que no se muestra) y luego decide colocarlo en el Buffer para que quede a disposición del proceso Consumidor. En primer lugar deberá esperar a que el monitor esté desocupado, quedando así garantizada la colocación en forma atómica. Una vez que obtiene el acceso al monitor ejecuta el procedimiento Colocar. Si el Buffer está lleno (Cont=N) el proceso Productor es puesto a dormir hasta que lo despierte una señal que le indique que ya no lo está (No_lleno) y el único lugar donde puede generarse esa señal (signal) es en el procedimiento Extraer luego de una extracción exitosa. Pero si Cont<N entonces el Buffer alojará el item en la posición señalada por el puntero In, luego de lo cual se desplazará ese puntero una unidad y finalmente enviará una señal de No_vacío por si hubiese algún proceso Consumidor durmiendo a la espera de items.

# 4. Simulación de arquitecturas paralelas

## 4.1. Introducción

Las aplicaciones escritas para sistemas de procesamiento paralelo son programas que requieren, para su ejecución, equipos dotados de N procesadores, de modo que en cada uno de ellos pueda correr simultáneamente un proceso creado en tiempo de ejecución, compartiendo eventualmente entre todos alguna zona de la memoria, con la esperanza de que el tiempo total que le tome correr a la aplicación completa se reduzca una cantidad de veces comprendida entre 1 y N con relación al que le tomaría correr en un equipo secuencial monoprocesador. ¿Cómo podría una aplicación escrita para su procesamiento paralelo correr en una máquina con un solo procesador? Bueno, al decir "...el tiempo que le tomaría correr en un equipo secuencial monoprocesador..." no me refiero al programa paralelo textual sino a su versión equivalente escrita para un computador monoprocesador, no obstante lo cual también es posible correr directamente el programa paralelo en un equipo con un sólo procesador y el artificio que lo permite es precisamente el que instrumenta la simulación que vamos a presentar a continuación y en alguna medida el mecanismo en el que se basan los sistemas multitarea/multiusuario para crear la ilusión de que dos o más procesos pueden ejecutarse simultáneamente cuando la arquitectura secuencial monoprocesadora (Von Neumann) sobre las que operan, ¡no permite más que uno a la vez!.
Un programa redactado para ser procesado en máquina paralela incluirá sentencias especiales (las que le darán precisamente el carácter paralelo), desde las encargadas de la creación de procesos y el lanzamiento de los mismos en distintas unidades de procesamiento, hasta los pedidos de acceso a zonas comunes, los pedidos de bloqueo y liberación de las zonas comunes, etc., insertadas dentro de código de programación convencional (no paralelo). Al menos esa es la filosofía de muchas de las herramientas de programación en este paradigma.
Vamos a introducir sistemas capaces de simular el comportamiento de computadoras de procesamiento paralelo en máquinas monoprocesadoras, de modo que resulte posible en estas correr programas diseñados para aquellas. Los vamos a presentar descompuestos en bloques uno de los cuales es precisamente la APLICACION.




Llamaremos por número, desde #0 hasta #N a cada una de las unidades de procesamiento de la computadora a simular y "procesador local" (*host processor*) al que reside física y realmente en nuestra máquina. En él va a comenzar a correr la aplicación, y a medida que la misma vaya arribando a puntos de interés global (primitivas de sincronización o referencias a memoria insertadas en el código) otro bloque llamado KERNEL, con el que la APLICACION estará intercambiando señales permanentemente, los detectará y ordenará las acciones a seguir. En definitiva la función del KERNEL no es otra que reordenar las tareas de modo de transformar un proceso paralelo en uno equivalente serie. Veámoslo desde el siguiente punto de vista: se inicia la ejecución de una APLICACION, lo cual en la máquina paralela sucedería en la unidad #0 y en la máquina real sucede en el procesador local (*host processor*). Hasta tanto no se arribe a una sentencia que ordene una acción de carácter global, no habrá diferencias.
Analicemos a partir de aquí dos situaciones posibles, (1) el programa ordena la creación y el lanzamiento de un proceso en otra unidad de procesamiento, el KERNEL lo detiene y lo deja en un estado de "listo para retomar su ejecución", como no tiene dónde lanzar el nuevo proceso, lo programa para ser lanzado en el mismo procesador local y para que una vez iniciada su ejecución corra hasta que se vea obligado a detenerlo, (2) en un instante determinado uno de los procesos ordena una referencia (lectura/escritura) a una zona compartida de la memoria, el KERNEL lo debe detener, tomando nota de qué tipo de operación ha sido solicitada así como del instante de tiempo en el que ha sucedido la solicitud, pudiendo así planificarla para un instante futuro. Esta planificación (*scheduling*) consiste en ir colocando en una cola todas las tareas que, producto de los eventos del tipo (1) y (2) vayan quedando pendientes, respetando el

orden cronológico cuando se trate de referencias a memoria y colocándolas al final cuando se trate de solicitudes de reinicio de ejecución de proceso. Una vez que comiencen a ejecutarse esas acciones programadas, cada vez que una referencia a memoria (tomada del tope de la cola) sea ejecutada, el proceso correspondiente pasará al estado de "listo para retomar su ejecución" y por ende planificado, por lo que llegará un momento en que la cola de planificación sólo va a contener solicitudes de reinicio, entonces el KERNEL comenzará a relanzar uno a uno todos los procesos hasta que comiencen nuevamente las detenciones producto de eventos de tipo (1) y/o (2).
Resumiendo: el KERNEL irá planificando (entendiendo por planificar colocar en la posición "adecuada" de una cola) referencias a memoria y solicitudes de reinicio de corrida de procesos; e irá ordenando las tareas a seguir tomándolas del tope de esa cola una vez que todos los procesos se hayan detenido. Como consecuencia, las operaciones que la APLICACION espera sean realizadas en forma simultánea, "una al lado de la otra" se transformarán en otras realizadas en forma secuencial "una a continuación de la otra", resultando una ejecución sincronizada de las referencias a memoria (tal como sucederían en la realidad), y corriendo uno tras otro los procesos en tanto sus corridas no se afecten mutuamente.
Por último existe un bloque que también intercambia señales con el KERNEL cuya misión es ejecutar las referencias a la memoria compartida. Este bloque, designado como SIMULADOR en el diagrama, se comporta de acuerdo a la arquitectura y al protocolo que para todo lo que tenga que ver con memoria se desee simular. O sea que para una misma APLICACION y un mismo KERNEL, un usuario podrá elegir las características principales de la máquina paralela en la que se desea "ver correr" la APLICACION modificando el bloque SIMULADOR o eventualmente creando uno nuevo.
El sistema completo ofrecerá al usuario dos tipos de resultado: por un lado la salida que produciría la APLICACION al correr sobre una máquina paralela real y por otro un paquete de información y estadísticas sobre cuestiones internas de la máquina simulada. El primer tipo de resultado es útil para aquellos interesados en ensayar algoritmos paralelos y cuestiones inherentes exclusivamente a software de ese tipo, programas, etc. El segundo lo es para quienes deseen ensayar variantes propias de la arquitectura paralela, como por ejemplo: el número de unidades de procesamiento, el tamaño de memoria cache, el tamaño de las líneas de memoria cache, el protocolo de coherencia, etc.
Este tipo de simulación es uno de los tantos que existen en su género y dado su modo de funcionamiento es conocido como "simulación conducida por ejecución" (*execution-driven simulation*) puesto que todos los eventos son generados precisamente ejecutando la aplicación en el procesador local, de manera que la ilusión creada es la de que el programa corre en una máquina paralela formada por "N máquinas locales".
Otra alternativa es la simulación "conducida por trazas" (*trace-driven simulation*), aquí los eventos son generados a partir de trazas producidas por otras herramientas (ajenas al simulador) y eventualmente en otras máquinas, distintas del procesador local. El método no es del todo apropiado para nuestro caso dado que trazas paralelas producidas para un determinado sistema de memoria podrían nunca suceder en otro; el método resulta por tanto inexacto aunque rápido, pero a costa del enorme espacio que ocupan las buenas trazas.
Otra modalidad pasa por "interpretar " el código de máquina en lugar de "ejecutarlo", puede resultar una opción interesante cuando se pretende un nivel extremo de detalle pero muy lento dado lo inútil de interpretar instrucciones que no tengan efecto global.

# 4.2. Limes

**Li**nux **me**mory **s**imulator: Limes, es un simulador (conducido por ejecución) que a diferencia de casi la mayoría de sus equivalentes corre en PC i486 (o superior) bajo sistema operativo Linux, por supuesto, originalmente diseñado para correr programas de la colección SPLASH-2 (Stanford Parallel Applications for Shared Memory) que tienen carácter de benchmark. Se trata de programas escritos en "C" con el apoyo del juego de macros **ANL** (Argonne National Laboratory) para resolver las cuestiones de carácter global/paralelo.
Para correr Limes cualquier versión de kernel de Linux es adecuada pero es requisito que el compilador sea GNU CC 2.6.3, si se cuenta con uno superior (actualmente la versión es 2.95) basta con copiar sólo los archivos que conforman el entorno de desarrollo (*development environment*) a un directorio determinado y retocar ligeramente algunos archivos previo a la compilación.
El paquete incluye simuladores basados en cuatro protocolos (*snoopy*) de coherencia cache, Berkeley, Dragon, WIN (Word Invalidate, un protocolo basado en el método de invalidación parcial de palabra) y WTI (Write-Through Invalidate un protocolo elemental que exhibe la peor performance de todos). También permite implementar lo que llama Simulador Ideal y que no es otra cosa que la simulación de un sistema de memoria perfecta en el que todas las referencias se completan en un ciclo y además se admiten accesos simultáneos, es decir que se completan en un solo ciclo todos los accesos que pudieran coincidir para un instante determinado (ver Ensayos).

### 4.2.1. Macros ANL

A modo de ilustrar la utilización de los macros veamos un breve ejemplo de su aplicación para resolver operaciones de carácter global en un programa que incrementa en cuatro el valor de una variable (incrementándola en uno cuatro veces), para lo cual lanza tres procesos paralelos, realizando un incremento cada proceso, y uno el Master (se conoce como Master al proceso principal desde el que se lanzan los restantes). Veamos previamente una breve descripción de los macros que serán necesarios,

**LOCKDEC**(x) Declara la variable de bloqueo x.
**LOCKINIT**(x) Inicializa la variable de bloqueo x en su valor *free* ó cero, que representa no-bloqueo. Utilizado por el Master antes de que comience el procesamiento paralelo.
**LOCK**(x) Lleva la variable de bloqueo x a su valor *busy* ó uno, que representa bloqueo. Utilizado al ingresar a Zonas Críticas. Es una operación de tipo spin lock en la que cada verificación se hace en forma atómica.
**UNLOCK**(x) Lleva la variable de bloqueo x a su valor *free* ó cero, que representa no-bloqueo. Utilizado al salir de Zonas Críticas.
**CREATE**(p) Crea un proceso p que comienza su ejecución en paralelo con el padre que lo creó. Cuando el proceso p alcanza su fin deja de existir.
**WAIT_FOR_END**(n) El programa se detiene al llegar a esta sentencia hasta que completen su ejecución los n procesos que éste hubiese creado. Al momento de ejecutarse es muy probable que se ignore cuantos procesos quedan por terminar, no obstante se deben mencionar todos los creados.

```
int cont;
LOCKDEC(bloqueo);
void(Proceso);

main() {
    int i;
    cont=0;
    LOCKINIT(bloqueo);
    for(i=1;i<4;++i)                /* 1 */
        CREATE(Proceso);
    Proceso();                      /* 2 */
    WAIT_FOR_END(3);                /* 3 */
    printf("cont=%d\n",cont);       /* 4 */
    }

void Proceso() {
    LOCK(bloqueo);
    ++cont;
    UNLOCK(bloqueo);
    }
```

Teniendo en cuenta las explicaciones sobre cada macro utilizado, el programa resulta simple de entender. Sólo algunas líneas merecen comentario,
`/* 1 */` La función Proceso es lanzada tres veces y comienza a correr, en el mejor de los casos, en tres procesadores distintos.
`/* 2 */` Por cuarta vez es lanzada la función Proceso pero ahora queda corriendo en el mismo procesador en el que lo está haciendo main().
`/* 3 */` main() puede estar seguro de que se completó la función Proceso que está corriendo en su mismo procesador (la de la línea /* 2 */) pero como ignora si se han completado o no los otros tres, se detiene a la espera de ese evento.
`/* 4 */` Se imprime el valor de cont que en este punto es 4.

### 4.2.2. Limes-Guía de Usuario

Cada uno de los tres benchmarks cuenta con su propio juego de opciones para cada corrida (similares, salvo en los casos de parámetros específicos del problema resuelto por la aplicación). Se transcriben textualmente las partes del código fuente de cada uno en las que se comentan esas opciones.

```
/*************************************************************************/
/*                                                                       */
/*  Perform 1D fast Fourier transform using six-step FFT method          */
/*                                                                       */
/*  1) Performs staggered, blocked transposes for cache-line reuse       */
/*  2) Roots of unity rearranged and distributed for only local          */
/*     accesses during application of roots of unity                     */
/*  3) Small set of roots of unity elements replicated locally for       */
/*     1D FFTs (less than root N elements replicated at each node)       */
/*  4) Matrix data structures are padded to reduce cache mapping         */
/*     conflicts                                                         */
/*                                                                       */
/*  Command line options:                                                */
/*                                                                       */
/*  -mM : M = even integer; 2**M total complex data points transformed.  */
/*  -pP : P = number of processors; Must be a power of 2.                */
/*  -nN : N = number of cache lines.                                     */
/*  -lL : L = Log base 2 of cache line length in bytes.                  */
/*  -s  : Print individual processor timing statistics.                  */
/*  -t  : Perform FFT and inverse FFT.  Test output by comparing the     */
/*        integral of the original data to the integral of the data      */
/*        that results from performing the FFT and inverse FFT.          */
/*  -o  : Print out complex data points.                                 */
/*  -h  : Print out command line options.                                */
/*                                                                       */
/*  Note: This version works under both the FORK and SPROC models        */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                                                                       */
/*  Parallel dense blocked LU factorization (no pivoting)                */
/*                                                                       */
/*  This version contains one dimensional arrays in which the matrix     */
/*  to be factored is stored.                                            */
/*                                                                       */
/*  Command line options:                                                */
/*                                                                       */
/*  -nN : Decompose NxN matrix.                                          */
/*  -pP : P = number of processors.                                      */
/*  -bB : Use a block size of B. BxB elements should fit in cache for    */
/*        good performance. Small block sizes (B=8, B=16) work well.     */
/*  -s  : Print individual processor timing statistics.                  */
/*  -t  : Test output.                                                   */
/*  -o  : Print out matrix values.                                       */
/*  -h  : Print out command line options.                                */
/*                                                                       */
/*  Note: This version works under both the FORK and SPROC models        */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                                                                       */
/*  SPLASH Ocean Code                                                    */
/*                                                                       */
/*  This application studies the role of eddy and boundary currents in   */
/*  influencing large-scale ocean movements.  This implementation uses   */
/*  dynamically allocated four-dimensional arrays for grid data storage. */
/*                                                                       */
/*  Command line options:                                                */
/*                                                                       */
/*     -nN : Simulate NxN ocean.  N must be (power of 2)+2.              */
/*     -pP : P = number of processors.  P must be power of 2.            */
/*     -eE : E = error tolerance for iterative relaxation.               */
/*     -rR : R = distance between grid points in meters.                 */
/*     -tT : T = timestep in seconds.                                    */
/*     -s  : Print timing statistics.                                    */
/*     -o  : Print out relaxation residual values.                       */
/*     -h  : Print out command line options.                             */
/*                                                                       */
/*  Default: OCEAN -n130 -p1 -e1e-7 -r20000.0 -t28800.0                  */
/*                                                                       */
/*  NOTE: This code works under both the FORK and SPROC models.          */
/*                                                                       */
/*************************************************************************/
```

Una simulación completa involucra a los tres conjuntos de programas: APLICACION+KERNEL+SIMULADOR, linkeados sobre un único ejecutable. Ese proceso de compilación se maneja íntegramente desde los archivos makefile de cada aplicación. La única tarea del usuario es ejecutar "make" en el directorio en el que se encuentra el benchmark que va a constituir la aplicación, revisando previamente su makefile y adecuándolo a los propios intereses, comentando, descomentando o valorizando las variables que se detallan a continuación.

**APPCS**: define la lista de módulos de la forma "nombre**.c**" que conforman la aplicación.
**APPCCFLAGS**: define banderas a ser pasadas al compilador. Casi invariablemente se utilizará **-O2**. Otras opciones cono por ejemplo **-g** podrían resultar de interés para indicar al compilador que se requiere información para debugging, fundamentalmente si se trabaja con aplicaciones creadas por el usuario.
**APPLDFLAGS**: especifica banderas a ser pasadas al linker (como por ejemplo **-lm** para que se incluya la librería matemática).
**AUGFLAG**: por medio de esta variable es posible decidir que tipos de evento, de los generados por un proceso, deben ser instrumentados por Limes. Existen tres posibles niveles: **-0** indica que sólo se atenderán instrucciones de sincronización e instrucciones de máquina definidas por el usuario; es particularmente útil si sólo se pretende ensayar algoritmos con uso de locks para proteger secciones críticas. **-1** ordena implementar lo mismo que -0 más lecturas y escrituras "compartidas", es el nivel por default y **-2** ordena instrumentar todo, incluidas referencias a la pila, accesos a datos locales, etc., es la opción adecuada si se quiere simular el sistema con buen nivel de detalle (caches, etc.).
**SYNC**: por medio de esta variable se especifica la política de sincronización. Puede tomar los valores ´realistic´ (default), ´abstract´ o el de una implementación propia. Para el caso ´realistic´ todo pedido de LOCK o UNLOCK detectado por el kernel es pasado directamente al simulador, recibiendo el procesador que hizo el pedido una orden de quedar bloqueado en el caso que el LOCK solicitado hubiese estado ocupado. Ninguna acción es ejecutada sin el conocimiento del simulador, en tanto con ´abstract´ un pedido de LOCK por parte de un proceso procede del siguiente modo: si el LOCK esta libre continúa, si está ocupado el proceso se pone a sí mismo "a dormir" marcando su estado como BLOCKED y colocando su número ID en la cola asociada con ese LOCK; de ahí en más el kernel lo ignorará y el simulador no verá ningún pedido hasta que otro procesador ejecute UNLOCK sobre ese lock, entonces el kernel tomará el primer proceso de la cola pasando su estado a READY procediendo a planificar el proceso recientemente "despertado" que sabrá que ahora el LOCK es suyo.
**TARGET**: especifica el nombre del ejecutable.
**SIM**, **SIMHOME**: especifica nombre y ubicación del SIMULADOR (relativa al directorio limes/simulator). En caso de omisión se asume el simulador de memoria ideal. ´ber´, ´dra´, ´win´ y ´wti´ son las opciones para SIM que permiten implementar simuladores bajo los protocolos de Berkeley, Dragon, Word Invalidate y Write-Through Invalidate respectivamente, mientras que ´snoopy´ el valor de SIMHOME que aloja a cualquiera esos cuatro protocolos.
**ALGEVALONLY**: valorizada en ´1´ permite evitar la simulación de las referencias a memoria conservando no obstante el correcto orden de las mismas, mientras que si se comenta o elimina del makefile las referencias quedan en manos de los respectivos simuladores. La ejecución sin referencias es apropiada cuando sólo interesa ensayar algoritmos. La diferencia entre este modo y el simulador de memoria ideal está en el hecho que aquí los pedidos no son fragmentados en partes cuyo tamaño es fijado por la arquitectura simulada. Independientemente que un READ tome uno, dos, cuatro u ocho bytes se completará inmediatamente (en un ciclo) a diferencia del modo con memoria ideal en el que los pedidos mayores de cuatro bytes se fragmentarán en trozos precisamente de cuatro palabras; así un pedido de lectura de ocho bytes tomará dos ciclos. Luego, aunque alejado de la realidad física pero sin perder sincronización, el valor ´1´ para esta variable acelerará la simulación como máximo cuatro veces.

Cuando se ejecuta make por primera vez, el proceso de compilación se compone de las siguientes cuatro tareas:

1. Compilar la aplicación lo cual consiste en las siguientes sub-tareas,
   (a) Convertir los programas fuente en archivos asembler.
   (b) Llamar a un programa de aumentado para que instrumente nuevos archivos asembler a partir de los recientemente generados. El aumentado es realizado por un programa (aug) que es un analizador sintáctico y que mediante un barrido (*parsing*) del código asembler recientemente generado por la primera intervención del compilador, detecta las referencias a memoria y las sentencias de sincronización a fin de convertirlas al contexto de Limes (recordar que se está compilando un programa para multiprocesador, de modo que producto de esa compilación van a aparecer, entre otras cosas, referencias a una arquitectura que no existe en nuestra máquina y

que habrá que convertir en eventos interpretables por Limes o sea eventos capaces de dar intervención al KERNEL y al SIMULADOR). Toda esta conversión redunda en un importante agregado de código, de ahí el nombre de aumentado.

(c) Compilar los archivos aumentados guardando el código objeto en un archivo de nombre $(TARGET).a

2. Llamar al makefile del kernel para que genere el archivo kernel.a con el código objeto del kernel.
3. Llamar al makefile del simulador para que genere el archivo $(SIM).a con el código objeto del simulador.
4. Linkear todos los archivos de extensión .a en un único ejecutable de nombre $(TARGET).

Durante el proceso los archivos .a van siendo guardados, a medida que se van generando, en el directorio limes/lib. Es útil ver los comandos implementados por Limes para ir limpiando y borrando archivos de utilidad sólo temporaria:

**make dep**: genera las dependencias para el simulador y para el kernel.
**make clean**: limpia los archivos de la aplicación y del simulador (objeto y asembler).
**make simclean**: limpia los archivos del simulador (objeto y asembler). Deberá ejecutarse simclean si se desea compilar con Dragon una aplicación que acaba de ser compilada con Berkeley. No es necesario si el nuevo protocolo tiene otro valor de SIMHOME.
**make appclean**: limpia los archivos de la aplicación (objeto y asembler).
**make appsclean**: borra los archivos .s (asembler no aumentado) de la aplicación.
**make appSclean**: borra los archivos .s (asembler aumentado) de la aplicación.
**make kerclean**: limpia los archivos temporarios del kernel.
**make syncclean**: limpia los archivos objeto de la implementación de sincronización.
**make libclean**: limpia el directorio limes/lib.
**make totalclean**: limpia todo. Es conveniente ejecutar esta opción cuando se sospecha que algo no funciona bien. También si se han cambiado los valores de SYNC y/o ALGEVALONLY.

Desde ya las ventajas de los makefile están completamente explotadas. Si se compila una simulación y luego se hace una modificación en el código fuente de la aplicación, sólo los archivos fuente modificados serán recompilados, no los otros, ni el kernel ni el simulador.
Para correr una aplicación una vez compilada no hay mas que lanzar el ejecutable indicando las opciones propias de acuerdo a lo requerido por cada aplicación. Una vez lanzada la ejecución de una APLICACION, pulsando **CTRL**-\ se podrá visualizar la evolución de la misma, se volcará al terminal el estado de cada proceso y la línea actual de código fuente si la opción **-g** fue especificada en la variable APPCCFLAGS. Si la salida del simulador fue entubada hacia un archivo no se deberá usar la opción CTRL-\ pues mataría el proceso en el otro extremo de la tubería, usar en su lugar **kill -12**.
Además de las opciones propias de cada aplicación es posible, en cada ejecución, pasar comandos directamente al kernel. Para ello luego de los argumentos de la aplicación se deberán colocar dos guiones (--) luego de los cuales podrán aparecer esos comandos. Indicando **-- -h**, es posible visualizar estas opciones, las cuales de momento son cinco:

**-t**: genera un archivo de trazas. Las referencias a memoria son volcadas a la standard output (o standard error) a medida que van sucediendo y de acuerdo al sistema que se esté simulando. La traza es producida bajo formato ASCII plano pudiendo ser direccionada o entubada hacia un archivo para su posterior reprocesamiento o análisis. Esta salida es producida por la función `produce_trace()` de `kernel.c` y puede ser adaptada a gusto o necesidad de cada usuario. En el formato por default aparece una línea por cada instante, en orden cronológico, en el que al menos un procesador genera un evento. Su aspecto y significado es,

**T=**<tiempo> {**P**<n>: <operación><área> <dirección>,<tamaño>} ...

`tiempo:` el instante (medido en ciclos) en el que se genera el pedido.
`n:` número de procesador que genera el evento (comenzando por cero).
`operación:` R, W, L o U para READ, WRITE, LOCK o UNLOCK.
`área:` la zona de memoria sobre la que se produce la referencia, pudiendo tratarse de c para el segmento en el que se aloja el código del programa (obviamente aquí habrá sólo lecturas aunque con probabilidad de que sean compartidas), d para el segmento de datos (lecturas y escrituras compartidas) y s para el segmento de pila (con carácter privado de cada proceso). Locks y unlocks ocurren sólo en el área de datos compartidos.
`dirección:` la dirección virtual de la referencia.
`tamaño:` el tamaño en bytes del dato leído o escrito.

Ejemplo:

```
   :
T=10240  P0: Rd 220402,4   P3: Wd 324816,4   P5: Rs 3290482312,4
T=10243  P0: Rd 220406,4   P4: Ld 402442,4
   :
```

**-s**: produce estadísticas sobre cada proceso al final de la simulación: el número de lecturas y escrituras (compartidas y privadas), locks, barriers, tiempo total de ejecución y porcentaje de tiempo consumido en tareas de sincronización.
**-e**: fuerza la salida de limes a la standard error en lugar de standard output.
**-i**: permite especificar otro archivo de inicialización distinto de limes.ini. Líneas del tipo **ITEM=VALOR** (donde ITEM es una cadena sin blancos y VALOR un entero) serán especificadas en estos archivos registrándose principalmente información concerniente a memorias cache. Por ejemplo limes.ini (o el archivo especificado en **-i**) podrán contener líneas como:

```
cache_size=16
cache_line_size=64
cache_way=8
```

Este archivo indica simular un tamaño de memoria cache de 16 Kb, con líneas de 64 bytes, asociativa de 8 vías (los valores por default para cuando no se use archivo de inicialización son 8 Kb-32b-2 vías).
**-dITEM=VALOR**: permite definir parámetros de inicialización sin necesidad de crear un archivo .ini o sbreescribiéndolo si existe. Ejemplo:

```
FFT -m12 -p4 -- -dcache_size=16 -dcache_line_size=16 -dcache_way=8.
```

# 4.3. MulSim

**Mul**tiprocessor **Sim**ulator: MulSim, es un simulador de sistemas multiprocesadores de memoria compartida cuya ventaja es su simplicidad e independencia de plataforma para correr. A diferencia de Limes resulta portable a casi cualquier versión de Unix incluso Linux. La arquitectura que simula (definida por el lenguaje ensamblador Mas) es similar a la de los procesadores SPARC de Sun Microsystems: las instrucciones se ejecutan en un solo ciclo, se comporta como máquina *load/store*, las operaciones aritméticas se realizan únicamente en el modo registro-a-registro y tiene un sistema de ventanas de registro usado para almacenar la "pila global" (*runtime stack*).
Permite elegir entre cuatro variantes de interconexión de memoria,

- **PRAM**: un mecanismo idealizado también conocido como "memoria perfecta" en el cual no sólo los accesos a memoria principal se completan en un solo ciclo sino que además se admiten accesos simultáneos, es decir que todos los accesos programados para un instante dado se completan en un único ciclo de ejecución (útil para ensayar programas pero no arquitecturas). Desde luego este mecanismo sólo es implementable en simuladores, no tanto por la rapidez sino por la simultaneidad y por supuesto que no utiliza memoria caché.
- **Bus**: un protocolo (también sin memoria caché) en el que sólo se idealiza la velocidad de respuesta de la memoria principal, considerándola un ciclo, pero se permite un único acceso por vez.
- **Snoopy-update**: un protocolo elemental que simplemente actualiza copias de líneas presentes en más de una caché cuando una de ellas es actualizada; no hay flujo de información entre cachés ni estados que requieran tratamiento especial. Se consideran cachés de tamaño infinito (ver Ensayos).
- **Snoopy-Invalidate**: un protocolo elemental que simplemente invalida copias de líneas presentes en más de una caché cuando una de ellas es actualizada; no hay flujo de información entre cachés ni estados que requieran tratamiento especial. Se consideran cachés de tamaño infinito (ver Ensayos).

MulSim permite además a los usuarios programar su propio sistema de interconexión de memoria.
A diferencia de Limes que es un simulador pensado para correr programas escritos en C con la inclusión de macros de la colección ANL para las operaciones de carácter global, MulSim tiene un compilador para programas escritos en el lenguaje ensamblador Mas, mencionado más arriba, siendo esa la forma más directa, aunque menos práctica, de crear aplicaciones. También se provee un compilador para programas escritos en C con la inclusión de algunas funciones y macros (también provistos) consistiendo el proceso en este caso en la compilación del programa en C a su equivalente en Mas y luego la obtención del ejecutable (para el equipo host o del usuario) mediante el compilador anterior.
Una vez compilada una aplicación, la línea de comando para su ejecución tiene en todos los casos la siguiente forma:

<nombre> <#CPUs> <i/n> <protocolo> <arg1> <arg2> <arg3> ...

<nombre> El nombre del ejecutable.
<#CPUs> La cantidad de procesadores a simular.
<i/n> i para el caso de una corrida interactiva que puede ser ejecutada paso a paso ingresando el usuario permanentemente comandos de una colección que se ofrece y que resulta particularmente útil para tareas de debugging y para examinar detalladamente el comportamiento de los programas. O bien n para forzar una corrida no-interactiva completándose la ejecución del programa sin pausas.
<protocolo> p, b, su ó si según se quiera simular un sistema de memoria perfecta, uno de tipo Bus, un Snoopy-Update o un Snoopy-Invalidate.
<arg1> <arg2> <arg3> ... argumentos propios de cada aplicación.

# 5. Metodología/Análisis de Objetivos

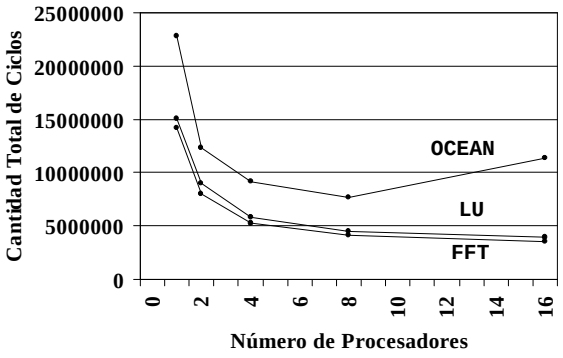
Entre los objetivos planteados para del presente proyecto figuraba el adquirir práctica en la búsqueda y puesta en marcha de simuladores, sean estos desarrollos académicos o productos comerciales, destinándolos a dar apoyo al dictado de cursos en general, abordando previamente las ideas fundamentales de los mismos con vistas a dar a la simulación un marco de verdadera utilidad.
En lo que hace a esta tarea y en particular al tiempo que para la misma había sido presupuestado, la realidad mostró que requería bastante más. Para la búsqueda en sí, el primer requisito era que los simuladores corrieran bajo alguno de los sistemas operativos disponibles, Linux o Windows (95/98). De la gran cantidad de estos productos que se ofrecen en la Web (casi todos desarrollados en universidades), absolutamente ninguno de los vistos corre bajo sistema operativo Windows, todos lo hacen bajo distintas versiones de Unix y sólo algunos bajo Linux. Por esta causa fue necesario descartar varios de los más conocidos como por ejemplo "Proteus" (Louisiana State University), "rsim" (Rice University) y "MINT" (University of Rochester). De los que corren bajo Linux pasaron por la lista, "Augmint" (University of Illinois at Urbana-Champaign) muy similar a "Limes" pero con menos facilidades provistas y por ende con mayor exigencia de programación para lograr las variantes a ensayar, "FAST" (University of Minnesota) descartado por proporcionar muchas alternativas para la simulación de sistemas secuenciales pero mínimas para la de sistemas paralelos además de escasa documentación para usuarios y también "DiST" (La Trobe University, Australia) un desarrollo sobre el que tuvimos un interés particular porque se presentaba además como herramienta para la enseñanza de conceptos de computación paralela y dado que ninguno de los links relativos a este simulador permitía bajarlo, procuré contactar directamente al responsable principal del proyecto (Dr. A. N. Pears), pero todos los intentos fueron infructuosos. La elección recayó finalmente en "Limes" y "MulSim". Cabe mencionar también una complicación importante que acompañó la puesta en funcionamiento de Limes. Tal se anticipaba en la documentación, era requisito compilarlo con GCC v2.6.3 o alguna versión anterior, a pesar de lo cual y dada la dificultad para conseguir este compilador, se intentó (sin éxito), gracias a la ayuda prestada por colegas de otras cátedras, hacer algunos cambios para lograr compilarlo con versiones actuales de GCC (que andan por la v2.95). Finalmente y ya próximos a desechar el simulador, a pesar de que se imponía como la mejor elección, se consiguió una vieja versión de la distribución Slackware [1994] de Linux a partir de la cual se obtuvieron los archivos correspondientes al entorno de desarrollo, una vez instalados los cuales, según instrucciones precisas de la guía de usuario de Limes, la compilación y consecuentemente su uso se pudieron concretar.
Una conclusión que vale la pena destacar es que indudablemente el auge de Internet torna muy rápidas y bastante sencillas estas búsquedas no obstante lo cual una cosa es ir en procura de productos de uso, y por ende de conocimiento, masivo y otra muy diferente es apuntar a programas desarrollados con objetivos específicos y principalmente académicos a los que hay que prestar especial atención para asegurarse, entre otras cosas, de que hayan sido concebidos para fines equivalentes al uso que se les pretende dar. Por otro lado, no siempre la documentación que los acompaña es tan completa como para cubrir las necesidades de cualquier usuario.
Con relación a la otra parte, es decir al estudio de los principios e ideas fundamentales del tema que se espera apoyar mediante simulación, lo realizado tuvo un alcance algo mayor. A medida que trabajaba, tomando una perspectiva desde luego más amplia y sobre mayor cantidad de temas que los que se comentan en este informe, me surgió la inquietud de cambiar relativamente el enfoque de esta tarea. Se había pensado y así se propuso, que el proyecto se concentrara en la elaboración de trabajos prácticos, no obstante lo cual consideré que dado que su última finalidad sería dar apoyo a una materia que está en plena elaboración, que la misma corresponde a una disciplina en la que muchos de los conceptos son absolutamente nuevos (al menos para la carrera) y siendo que la bibliografía disponible no es tan vasta y

que los temas siguen muchas veces otro tipo de orden o lineamiento, apoyados a menudo sobre menciones a máquinas reales de las que los estudiantes podrían no tener conocimiento ni tal vez tiempo de extraer sólo los conceptos, valía la pena hacer una presentación detallada de dos o tres temas fundamentales conformando algo que podría ser considerado como material de apoyo al dictado de la cátedra. Resolví entonces armar esta parte mediante un enfoque primeramente general y luego un poco más específico de temas relativos al curso, quedando finalmente este material ubicado en lo que va desde el comienzo del trabajo hasta este punto.
Lo que sigue es una serie de pruebas de carácter eminentemente práctico, donde se verifican mediante la simulación muchas de las ideas y puntos de vista expuestos hasta aquí y que aunque no tienen "formato de trabajo práctico", son ensayos que sientan las bases para poder elaborarlos fácilmente tanto a partir del tipo de experimento realizado en cada caso como de la forma de interpretarlos y analizarlos.

# 6. Ensayos

**CANTIDAD TOTAL DE CICLOS** -vs- **NÚMERO DE PROCESADORES**
SIMULADOR=Limes / PROTOCOLO=Berkeley / TAMAÑO DE MEMORIA CACHÉ (KB)=8 / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2

| | FFT | LU | OCEAN |
|---|---|---|---|
| **1** | 14192533 | 15059419 | 22789690 |
| **2** | 7993183 | 8981203 | 12292548 |
| **4** | 5294810 | 5796788 | 9197400 |
| **8** | 4131112 | 4463533 | 7653975 |
| **16** | 3556432 | 3968768 | 11348426 |




Para un sistema de cómputo paralelo, y para una aplicación dada, Escalabilidad es la propiedad de responder con un incremento lineal de la velocidad de ejecución al incremento del número de procesadores. Los gráficos muestran la inversa de la velocidad medida a través del número total de ciclos de máquina que demanda al sistema simulado correr cada uno de los programas indicados. Así presentada la Escalabilidad constituye una idealización ya que para todo sistema y para toda aplicación, existe un límite más allá del cual el incremento del número de procesadores no sólo no tiene sentido sino que hasta podría tornarse perjudicial. Un factor determinante para este problema es, por supuesto, el tipo de aplicación; algunas estarán programadas para lanzar no más que un número máximo de procesos paralelos, en cuyo caso un incremento del número de procesadores no influirá más allá de ese máximo, otras podrían intentar lanzar tantos procesos como les sea posible, en ese caso, al crecer el sistema, podrían originarse una cantidad tal de tareas de comunicación entre procesadores (en nuestro caso a través de la memoria compartida) capaz de entorpecer los cómputos y de empeorar el rendimiento.

**CANTIDAD TOTAL DE CICLOS** -vs- **NÚMERO DE PROCESADORES**
SIMULADOR=Limes / PROTOCOLO=Dragon / TAMAÑO DE MEMORIA CACHÉ (KB)=8 / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2

Estos resultados permiten ratificar los conceptos anteriores sobre Escalabilidad (independizándolos del Protocolo de Coherencia Caché utilizado) y observar además otro fenómeno precisamente relacionado con el tipo de protocolo.
Lo que hace que los protocolos de Invalidación y Actualización tengan rendimientos similares es la relación:

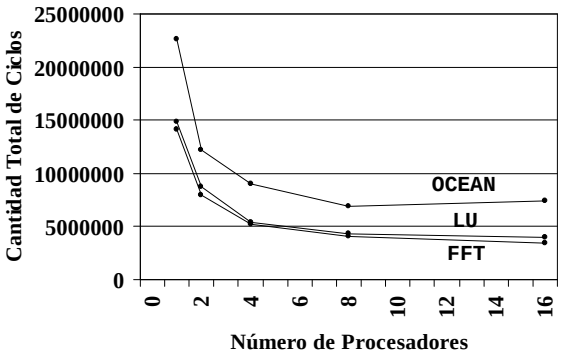
Costo de (Actualización) ● Costo de (Fallo por Invalidación+Acceso a MP)

Pero en general al aumentar el número de procesadores resulta:

Costo de (Actualización) < Costo de (Fallo por Invalidación+Acceso a MP)

Por lo que protocolos de invalidación como Berkeley tienen mejor performance con "pocos" procesadores mientras que para "muchos" procesadores se comportan mejor los de actualización como Dragon.

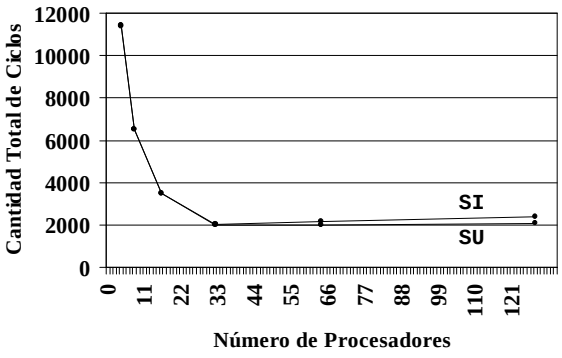
| | FFT | LU | OCEAN |
|---|---|---|---|
| **1** | 14151829 | 14866531 | 22611880 |
| **2** | 7938798 | 8744480 | 12159348 |
| **4** | 5214165 | 5349422 | 8972726 |
| **8** | 4028735 | 4293603 | 6862247 |
| **16** | 3450847 | 3972101 | 7394183 |

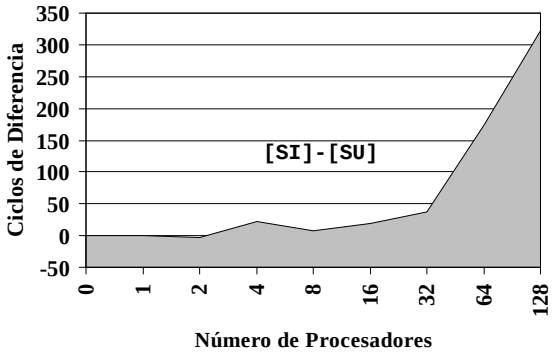



**CANTIDAD TOTAL DE CICLOS** -vs- **NÚMERO DE PROCESADORES**

SIMULADOR=MulSim / PROGRAMA=KWQSort / TAMAÑO DE MEMORIA CACHÉ (KB)=∞ / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=1 / NÚMERO TOTAL DE ARREGLOS=100 / LONGITUD DE CADA ARREGLO=8 / NÚMERO DE ARREGLOS ASIGNADOS POR PROCESADOR=4

| | SU | SI |
|---|---|---|
| **1** | 41149 | 41149 |
| **2** | 21222 | 21219 |
| **4** | 11387 | 11409 |
| **8** | 6525 | 6530 |
| **16** | 3501 | 3520 |
| **32** | 2010 | 2047 |
| **64** | 1988 | 2162 |
| **128** | 2085 | 2407 |



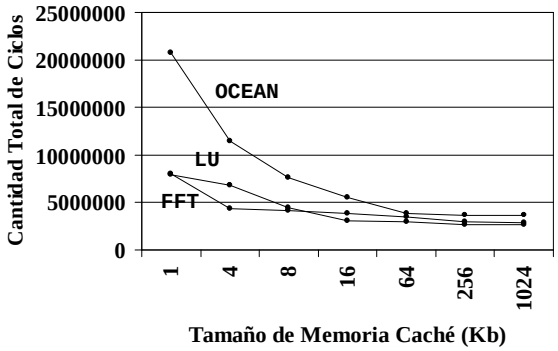




Los dos aspectos comentados en los ensayos anteriores son puestos en evidencia nuevamente en el presente. SI significa protocolo tipo Snoopy de Invalidación mientas que SU, Snoopy de Actualización. Se trata de protocolos elementales que simplemente invalidan/actualizan copias de líneas presentes en más de una caché cuando en una de ellas son actualizadas; no hay flujo de información entre cachés ni estados que requieran tratamiento especial. Los resultados son similares a los anteriores sólo que además se graficó (a partir de la misma tabla) la diferencia entre el Número Total de Ciclos de ejecución para cada protocolo ([SI] es la Cantidad Total de Ciclos para el programa ejecutado con el protocolo de Invalidación y [SU] lo mismo para Actualización). Se aprecia entonces la diferencia de performance a favor de Actualización con el incremento del número de procesadores.

**CANTIDAD TOTAL DE CICLOS** -vs- **TAMAÑO DE MEMORIA CACHÉ (KB)**
SIMULADOR=Limes / PROTOCOLO=Berkeley / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2 / NÚMERO DE PROCESADORES= 8

Habíamos dicho que en sistemas multiprocesadores la memoria caché cumple dos funciones, una "destrabar" los accesos a la memoria principal "repartiendo" en la medida de lo posible la información en ella alojada a fin de evitar múltiples detenciones de procesadores a la espera de poder accederla y otra producir accesos mucho más rápidos en caso de éxitos (*hits*) ya que su tecnología de construcción así lo permite. Tanto una situación como la otra se ven decididamente favorecidas por el incremento de tamaño de la memoria caché y consecuentemente reducida la tasa de fallos.

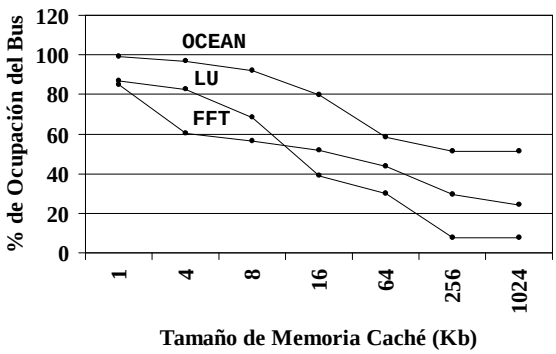
| | FFT | LU | OCEAN |
|---|---|---|---|
| **1** | 7962260 | 7918539 | 20730193 |
| **4** | 4359608 | 6842757 | 11436907 |
| **8** | 4131112 | 4463533 | 7653975 |
| **16** | 3819369 | 3051220 | 5513527 |
| **64** | 3462973 | 2967719 | 3869975 |
| **256** | 2988337 | 2694781 | 3659380 |
| **1024** | 2819247 | 2694781 | 3652562 |




Cuanto mayor sea la porción de memoria principal alojada en cada caché tanto menor será la probabilidad de fallos y por ende la necesidad de accesos a memoria principal (independientemente del protocolo). Además cabe otra observación importante, hay un límite más allá del cual no tiene sentido hacer crecer las memorias caché y para reconocer esto debemos aceptar que a partir de determinado tamaño de caché, ya no crecerá la tasa de fallos. Imaginemos, como caso general, una asociativa por conjuntos de k vías y que tal lo hecho en el ensayo, la hacemos crecer de tamaño dejando constante la cantidad de vías. Lo que se consigue entonces es que las líneas de memoria principal destinadas a cada conjunto se reduzcan en cantidad y se distancien entre sí (debido al crecimiento de la cantidad de conjuntos), ¿cuál es la consecuencia de esto? Puede llegar un punto en que las líneas destinadas a cada conjunto sean "tan pocas" y estén "tan lejos" una de otra que, en un caso extremo, al subir puedan no tener necesidad de desplazar a ninguna. De ahí en más un crecimiento de tamaño de caché no resulta perceptible en el rendimiento. Un ejemplo claro de esto se ve en las dos últimas posiciones de la columna de la tabla de LU donde para un crecimiento del tamaño de caché de 256 Kb a 1024 Kb (1 Mb) no hay mejora alguna de rendimiento. Por esa causa algunos simuladores (como MulSim) consideran una memoria caché de tamaño infinito de manera que permiten subir a ellas todas las líneas que sea necesario sin poner límites.

**PORCENTAJE DE OCUPACIÓN DEL BUS** -vs- **TAMAÑO DE MEMORIA CACHÉ (KB)**
SIMULADOR=Limes / PROTOCOLO=Berkeley / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2 / NÚMERO DE PROCESADORES= 8

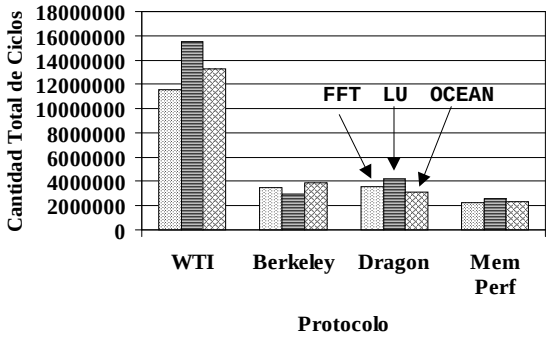
| | FFT | LU | OCEAN |
|---|---|---|---|
| **1** | 85,04 | 86,73 | 99,32 |
| **4** | 60,39 | 82,57 | 96,89 |
| **8** | 56,33 | 68,25 | 91,81 |
| **16** | 51,74 | 39,02 | 79,79 |
| **64** | 43,84 | 29,86 | 58,21 |
| **256** | 29,25 | 7,57 | 51,22 |
| **1024** | 24,32 | 7,57 | 51,00 |



Cuando las caché pueden asistirse entre sí (ej. Berkeley o Dragon) la necesidad de acceder a memoria principal se reduce. Por lo expuesto en el ensayo anterior, esa necesidad se reduce tanto más cuanto mayor es el tamaño de la caché. Reconociendo como Bus sólo a aquella parte del mismo que canaliza el flujo desde y hacia la memoria principal, un coeficiente interesante es el porcentaje de uso del mismo, evaluado como el cociente entre el número de ciclos en los que el Bus haya sido utilizado y la cantidad total de ciclos que la ejecución del programa hubiere demandado. Queda claro entonces el por qué de la reducción de uso del Bus para incrementos de tamaño de caché.

**CANTIDAD TOTAL DE CICLOS** -vs- **PROTOCOLO**
SIMULADOR=Limes / TAMAÑO DE MEMORIA CACHÉ (KB)=64 / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2 / NÚMERO DE PROCESADORES=8

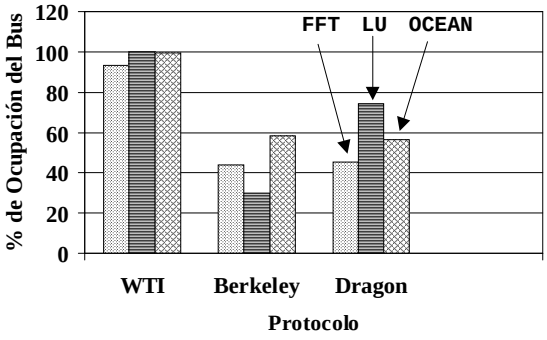
| | FFT | LU | OCEAN |
|---|---|---|---|
| **WTI** | 11536115 | 15553580 | 13294994 |
| **Berk** | 3462973 | 2967719 | 3869975 |
| **Drag** | 3531857 | 4155726 | 3085117 |
| **MP** | 2273356 | 2585198 | 2302317 |




Este ensayo permite arribar a una conclusión bastante predecible: entre los protocolos de aplicación real no hay un "ganador". Comparando las barras correspondientes a Berkeley y Dragon vemos que por ejemplo, LU es el programa de mejor performance con Berkeley y el de peor con Dragon mientras que exactamente lo contrario sucede con OCEAN, en tanto FFT ocupa una posición intermedia en ambos casos. WTI es un protocolo elemental que únicamente implementa la invalidación de las líneas actualizadas pero cada caché se sirve de la memoria principal (y no de otras caché) ante fallos y procede a la escritura directa (*write through*) y no a post escritura (*write back*) cuando es actualizada, por ello exhibe la peor performance para todas las aplicaciones.
Por último y como era de esperar el mejor rendimiento corresponde al protocolo (ideal) de Memoria Perfecta. En él no sólo que todos los accesos a memoria principal se completan en un ciclo sino que además se admiten accesos simultáneos, es decir que todos los accesos programados para un instante dado se completan en un solo ciclo de ejecución.
También es digno de hacerse notar el hecho que la performance de los protocolos de aplicación real está mucho más cerca del caso ideal (Memoria Perfecta) que del elemental (WTI) y esto decididamente, al menos para mí, es un resultado inesperado

**PORCENTAJE DE OCUPACIÓN DEL BUS** -vs- **PROTOCOLO**
SIMULADOR=Limes / TAMAÑO DE MEMORIA CACHÉ (KB)=64 / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2 / NÚMERO DE PROCESADORES=8

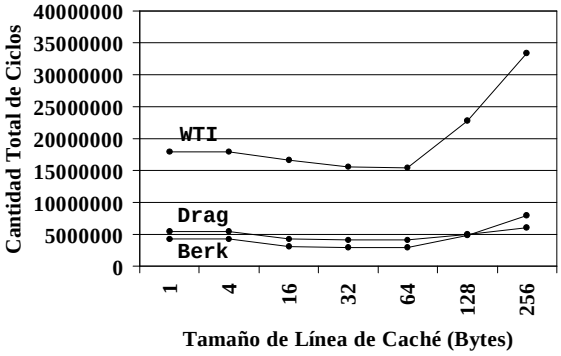
| | FFT | LU | OCEAN |
|---|---|---|---|
| **WTI** | 93,39 | 100,00 | 99,54 |
| **Berkeley** | 43,84 | 29,84 | 58,23 |
| **Dragon** | 45,06 | 74,09 | 56,62 |



Nuevamente una cuestión predecible es que el protocolo de peor performance (WTI) es el que hará mayor uso del Bus, fundamentalmente porque WTI accede a memoria principal en cada fallo. En cuanto a los dos de aplicación real (Berkeley y Dragon) podríamos decir que como los fallos, siempre que sea posible, son asistidos desde otras caché los accesos a memoria principal se ven sensiblemente reducidos y en todo caso las cuestiones comparativas deben contemplar la aplicación ejecutada.

**CANTIDAD TOTAL DE CICLOS** -vs- **TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)**
SIMULADOR=Limes / PROGRAMA=LU / TAMAÑO DE MEMORIA CACHÉ (KB)=64 / GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ=2 / NÚMERO DE PROCESADORES=8

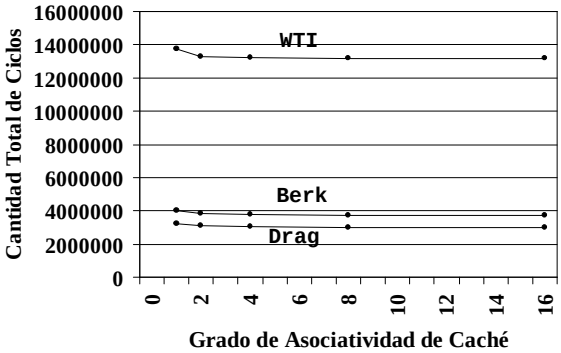
| | WTI | Berkeley | Dragon |
|---|---|---|---|
| **1** | 17944540 | 4258534 | 5472416 |
| **4** | 17944520 | 4258514 | 5472396 |
| **16** | 16585485 | 3102075 | 4337894 |
| **32** | 15553580 | 2967719 | 4155726 |
| **64** | 15479521 | 2926301 | 4077128 |
| **128** | 22745765 | 4920943 | 5043398 |
| **256** | 33434893 | 7982447 | 6060471 |




En este ensayo vemos que para cualquier protocolo los gráficos tienen un aspecto parecido y que todos presentan una zona de mejor rendimiento, empeorando tanto para incrementos como para decrementos del tamaño de línea de caché. ¿Cuál sería la explicación de este fenómeno? Hay dos parámetros que se pueden relacionar con el tamaño de línea: la penalización de fallos y la frecuencia de fallos. La penalización se compone de dos tiempos (sumados) el de acceso a la línea que se va a subir desde memoria principal y el de transferencia de esa línea. El tiempo de acceso es independiente del tamaño de línea y el de transferencia crece linealmente con éste. En cuanto a la frecuencia de fallos podemos decir que será elevada para líneas "demasiado pequeñas" dado el desaprovechamiento del principio de localidad espacial y será elevada también para líneas "demasiado grandes" porque hay un límite de tamaño más allá del cual comenzará a subirse a la caché información suficientemente distante del dato sobre el que se haya producido el fallo que bien podría no ser accedida antes de que la línea sea nuevamente desalojada. Esto hace que el espacio de la memoria caché comience a llenarse de información que no se será accedida en detrimento de información que si lo sería, aumentando consecuentemente la tasa de fallos.
De estos dos parámetros de evaluación de fallos (penalización y tasa) el aumento de tasa predomina para tamaños pequeños de línea aunque no exageradamente dada la compensación que provoca la reducción de penalización, pero para líneas grandes los dos efectos se superponen y la caída en el rendimiento es entonces importante.

**CANTIDAD TOTAL DE CICLOS** -vs- **GRADO DE ASOCIATIVIDAD DE MEMORIA CACHÉ**
SIMULADOR=Limes / PROGRAMA=OCEAN / TAMAÑO DE MEMORIA CACHÉ (KB)=64 / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=32 / NÚMERO DE PROCESADORES=8

| | WTI | Berkeley | Dragon |
|---|---|---|---|
| **1** | 13765884 | 4040166 | 3238415 |
| **2** | 13294944 | 3869975 | 3085117 |
| **4** | 13209821 | 3795744 | 3034482 |
| **8** | 13146890 | 3744566 | 3008672 |
| **16** | 13153194 | 3744506 | 3012929 |



Recordemos el significado de Grado de Asociatividad: toda caché de N líneas está dividida en n conjuntos de k líneas cada uno, de manera que n×k=N. Cuando una línea va a subir desde memoria principal a memoria caché lo primero que se debe definir es en que conjunto se va a colocar. Esa elección se realiza mediante la operación de módulo entre la posición de la línea en memoria principal (P) y la cantidad de conjuntos (P mod n). Luego se designa la posición dentro del conjunto elegido, para lo cual si hay que desalojar una existente, se puede optar en forma aleatoria o bien por la menos recientemente accedida. ¿Cuál sería entonces la mínima cantidad de conjuntos (n) que puede contener una memoria caché? Uno sólo de tamaño N, en este caso, más allá del criterio de reemplazo, las líneas subidas podrán ocupar cualquier posición. ¿Y cuál sería la máxima cantidad de conjuntos que pueda contener una memoria caché? N de una línea cada uno, en ese caso la ubicación de la línea a subir es única y resulta de la operación de módulo entre su ubicación en memoria principal y la cantidad total de líneas de la caché (P mod N). La cantidad de líneas de cada conjunto en los que esté dividida a la memoria caché (k) es lo que se conoce como Grado de Asociatividad y según lo visto podrá variar desde k=1 hasta k=N. El caso k=1 es conocido como caché de correspondencia directa (*direct mapped*), el caso k=N como caché totalmente asociativa y todo caso intermedio como caché asociativa por conjuntos de k vías.
La modalidad de correspondencia directa resulta en un mal aprovechamiento del espacio de la memoria caché aún cuando es la alternativa más simple desde el punto de vista del hardware. El hecho de que cada línea que sube desde memoria principal pueda alojarse en una única posición de la caché constituye de alguna manera una restricción. Para ofrecer a la línea que sube más alternativas (digamos dos, cuatro o más lugares posibles) hace falta dividir a la caché en conjuntos (de dos, cuatro o más vías), ahora bien ¿implica esto un beneficio? La respuesta es ¡sí por un lado y no por otro!, veámoslo de la siguiente manera: toda configuración de caché de correspondencia directa puede alcanzarse también en una asociativa por conjuntos de k vías (para cualquier k) pero no al revés, luego hay más configuraciones posibles o dicho de otro modo hay más conjuntos distintos de N líneas todas distintas formados a partir de la memoria principal (llamémosles $\delta_i$), que se pueden alojar en una caché asociativa por conjuntos que la cantidad que puede alojar la misma caché en correspondencia directa. Un razonamiento análogo conduce a la conclusión que entre dos caché asociativas por conjuntos de $k_1$ y $k_2$ vías respectivamente con $k_1>k_2$, la caché con $k_1$ vías podrá alojar más conjuntos $\delta_i$ que la de $k_2$ vías; y como caso extremo una caché con k=N (totalmente asociativa) permitirá alojar a todos los conjuntos $\delta_i$ que se puedan formar. Parecería entonces que a incrementos del grado de asociatividad deberían corresponderle mejoras de performance, sin embargo hay otro factor que hace que esto no suceda. Siempre la memoria caché es de un orden de tamaño muy inferior que el de la memoria principal, por eso en cada posición podrá alojar un número mayor que uno de líneas de la memoria principal (P/N para el caso de correspondencia directa, P para las totalmente asociativas y toda la gama intermedia para los distintos grados de asociatividad). Cuando la CPU solicita una línea a una caché de correspondencia directa, debe dirigirse a una única posición y chequear si la línea que allí se encuentra es la solicitada. Cuando la solicitud se hace sobre una caché asociativa por conjuntos de k vías, la CPU debe dirigirse al conjunto correspondiente y recorrerlo íntegramente (sus k líneas) en busca de la línea solicitada. Cuando la solicitud se hace sobre una caché totalmente asociativa es necesario barrer todas sus líneas en busca de la solicitada. Luego, cuanto mayor sea k mas tiempo le tomará a la CPU encontrar la línea solicitada por lo que desde este punto de vista la situación es exactamente inversa a la que plantea el análisis a través de los conjuntos $\delta_i$.
Resultará finalmente que hay un valor o una zona de valores de k para los que se da la mejor performance, empeorando tanto para incrementos (recorridos más largos en busca de una línea) como para decrementos (conjuntos $\delta_i$) respecto de esos valores. En nuestro ensayo resultó más evidente el comportamiento señalado para valores reducidos que para valores elevados de k.
Por razones del tipo de las expuestas además de constructivas, rara vez las caché tienen grados de asociatividad distintos de 2 ó 4.

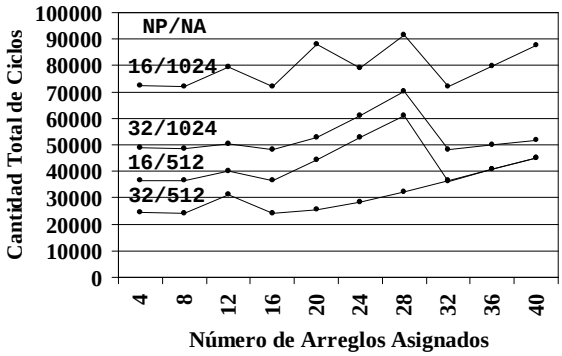
**CANTIDAD TOTAL DE CICLOS** -vs- **NÚMERO DE ARREGLOS ASIGNADOS POR PROCESADOR**

SIMULADOR=MulSim / PROGRAMA=KWQSort / TAMAÑO DE MEMORIA CACHÉ (KB)=∞ / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=1 / LONGITUD DE CADA ARREGLO=16 / PROTOCOLO=Snoopy Invalidate

KWQSort es un programa elemental, muy lejos de la categoría de benchmark, que acompaña el paquete del simulador MulSim pero que no obstante nos va a permitir (al igual que el utilizado en el ensayo siguiente, Prime) sacar algunas conclusiones importantes. Se dispone de N arreglos de dimensión NA para ordenar y de un sistema multiprocesador de NP procesadores. El programa acepta como uno de sus argumentos al parámetro k (Número de Arreglos Asignados) con el siguiente significado: cada vez que un

procesador queda desocupado le son asignados k arreglos para ordenar, tarea que una vez completada lo deja nuevamente desocupado. Fueron ensayadas todas las combinaciones posibles de NP 16 y 32 con NA 512 y 1024.

| | 32/1024 | 16/512 | 32/512 | 16/1024 |
|---|---|---|---|---|
| **4** | 48893 | 36492 | 24643 | 72405 |
| **8** | 48433 | 36636 | 24199 | 72055 |
| **12** | 50203 | 40081 | 31040 | 79580 |
| **16** | 48170 | 36378 | 24105 | 71938 |
| **20** | 52771 | 44383 | 25688 | 87771 |
| **24** | 61081 | 52662 | 28293 | 79100 |
| **28** | 70338 | 60902 | 32249 | 91401 |
| **32** | 48053 | 36237 | 36350 | 72021 |
| **36** | 49914 | 40800 | 40945 | 79918 |
| **40** | 51807 | 44901 | 45022 | 87551 |



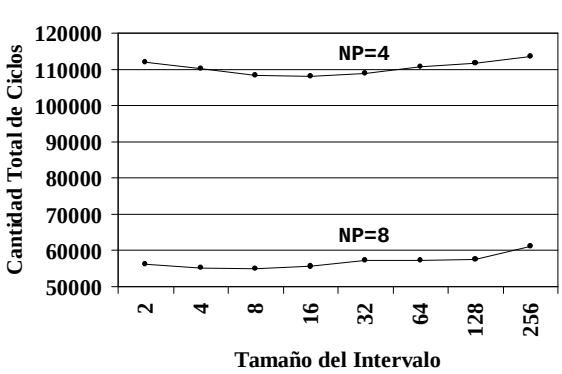

Lo interesante de este ensayo pasa por poner en evidencia dos complicaciones importantes que presenta la programación paralela con relación a la programación en sistemas monoprocesador. La primera de ellas es que la distribución de tareas entre las distintas unidades de procesamiento queda a cargo del programa o digamos mejor ¡del programador!, y esa tarea no es trivial ni siquiera en programas elementales como KWQSort. Un análisis previo podría anticipar que un valor grande de k puede provocar situaciones en las que queden algunos procesadores desocupados mientras los demás están ocupados ordenando y eso indudablemente implica una caída de rendimiento, tal como lo demuestra el ensayo. Lo difícil en este problema es predecir el comportamiento de programa para valores pequeños de k. Por una cuestión de escala no se mostraron los resultados para k=1, 2 y 3 pero en casi todos los casos la Cantidad Total de Ciclos fue mayor que para k=4. La mejor performance parece alcanzarse en k=16.
La otra cuestión a considerar sobre la programación paralela y que en alguna medida se desprende de la anterior es que, a diferencia de la programación en sistemas monoprocesador, el programador debe tener una noción, aunque sea mínima, de la arquitectura del sistema. La idea de distribuir entre los procesadores conjuntos de k arreglos es sólo una de tantas soluciones, pero también serviría por ejemplo llevar la cuenta de los arreglos que a cada procesador le quedan por ordenar e ir asignando una cantidad variable, de modo de "emparejar", sin aguardar que cada uno complete la tarea y manteniendo así al sistema más lejos de la posibilidad de que queden algunos procesadores desocupados y otros trabajando. Corresponde también al programador decidir cosas como por ejemplo,

- si se destina un procesador para distribuir las tareas y llevar el control de todos los demás,
- si al procesador designado a esa tarea no se lo hace ordenar,
- etc.

**CANTIDAD TOTAL DE CICLOS** -vs- **TAMAÑO DE INTERVALO A CHEQUEAR POR PROCESADOR**

SIMULADOR=MulSim / PROGRAMA=Prime / TAMAÑO DE MEMORIA CACHÉ (KB)=∞ / TAMAÑO DE LÍNEA DE MEMORIA CACHÉ (BYTES)=1 / N=4096 / PROTOCOLO=Snoopy Update

| | NP=4 | NP=8 |
|---|---|---|
| **1** | 193895 | 97689 |
| **2** | 112004 | 56344 |
| **4** | 110122 | 55200 |
| **8** | 108341 | 54954 |
| **16** | 108176 | 55758 |
| **32** | 108791 | 57321 |
| **64** | 110694 | 57357 |
| **128** | 111815 | 57438 |
| **256** | 113531 | 61069 |



El programa Prime determina la cantidad de números primos que hay entre 2 y N. Requiere como argumento al parámetro TI (Tamaño del Intervalo) con el siguiente significado: a cada procesador desocupado se le van asignando TI números para chequear de manera que el primero verifica desde 2 hasta 2+TI, el segundo los TI siguientes y así. El problema es bastante similar al anterior, un TI muy elevado asignará muchos números a pocos procesadores (pudiendo en el caso extremo asignarse todos a uno) quedando procesadores desocupados, quizá todo el tiempo. Como contrapartida un TI reducido (pudiendo en el caso extremo ir asignándose hasta un sólo número por procesador) eleva la cantidad de operaciones de carácter global a través de las cuales se produce la comunicación entre procesadores, es decir que cada unidad podría destinar tal vez más tiempo a ir leyendo el o los números asignados y escribiendo los resultados de su cómputo que al proceso de cálculo en sí. El ensayo fue realizado simulando dos sistemas con 4 y 8 procesadores respectivamente. No se muestran en el gráfico los resultados para TI=1 porque ocasionarían una compresión de escala que haría perder de vista la evolución de los restantes. Los mejores rendimientos se producen para TI=16 (NP=4) y TI=8 (NP=8).

# 7. Conclusiones

Las conclusiones a las que permite arribar el presente trabajo merecen clasificarse en dos áreas perfectamente definidas, la utilidad de la simulación como herramienta de análisis de sistemas de cómputo de procesamiento paralelo, por un lado y las diferencias/beneficios de estos sistemas frente a los tradicionales de tipo secuencial, por otro.

Como elemento de apoyo al estudio y observación de estos sistemas, la simulación se impone como una herramienta no sólo inevitable sino casi ideal ya que su contrapartida, es decir el ensayo de sistemas reales, no sólo requiere importantes tareas de laboratorio para cada modificación a ensayar (aún las aparentemente insignificantes) sino también la necesidad de contar con instrumental extremadamente específico y sofisticado para realizar tareas de monitoreo de hardware. Por ejemplo, no reviste mayores inconvenientes construir un prototipo de alguna arquitectura, probarlo y ponerlo en funcionamiento, pero estudiarlo detalladamente requiere "meterse dentro del mismo", medirle no sólo la cantidad de ciclos de reloj que le toma ejecutar un programa, sino también la cantidad de esos ciclos en los que el Bus es utilizado, la cantidad de invalidaciones/actualizaciones que genera tal o cual protocolo, los tiempos o porcentajes del tiempo total de ejecución que cada procesador está bloqueado, etc. Algunas de esas mediciones, u otras equivalentes, son factibles a través del software pero la mayoría de ellas decididamente no, mientras que en un simulador todos los parámetros de interés son materializados por variables y por ende muy fáciles de acceder, medir y analizar. Otra observación, tanto a partir de los dos simuladores utilizados como también de muchos otros que por distintas razones fueron evaluados y descartados es que el espectro de posibilidades que ofrecen suele estar limitado a microarquitecturas y configuraciones alrededor de las cuales los simuladores fueron diseñados, quedando las posibilidades ofrecidas a los usuarios reducidas a la elección de algunos parámetros, ingresados en general bajo la forma de argumentos, ya sea de cada corrida o de cada compilación. Casi siempre se ofrece la posibilidad de modificar los lineamientos alrededor de los cuales los simuladores fueron originalmente pensados, mediante tareas, importantes, de programación (en general en C/C++). Otro punto importante es que estos simuladores no sólo representan una herramienta muy útil para el estudio del hardware de computadores paralelos sino también una opción extremadamente útil, económica y accesible para quienes quieran adentrarse en la programación de los mismos, el ensayo de algoritmos y la solución de problemas mediante lenguajes (o adaptaciones de lenguajes) que de otro modo requerirían no sólo de los compiladores y sistemas operativos correspondientes sino también de los equipos capaces de soportarlos.

En cuanto al procesamiento paralelo en sí podríamos decir que es uno de los caminos a través de los cuales se procura alcanzar una mejora de performance aprovechando no sólo los bajos costos sino también el crecimiento casi ilimitado de posibilidades ofrecidas por el hardware. Además de representar una solución al problema de hacer correr más rápido los programas (paralelizándolos "a la fuerza" llegado el caso, es decir particionándolos con ese sólo objetivo) se presta especialmente para abordar problemas que tienen carácter paralelo en si mismos, en los que se puede permitir el avance del cómputo de una cantidad importante de sub-problemas bajo la condición, obviamente necesaria, de que cada uno pueda prescindir, de a tramos, de los resultados parciales o definitivos de los restantes. Algunos ejemplos de este tipo de problema son: el pronóstico del clima, la estrategia militar, el manejo de grandes bases de datos, el diseño de circuitos de muy alta escala de integración y la inteligencia artificial, por nombrar algunos.

Hay quienes creen no obstante (entre los que humildemente me incluyo) que sin perjuicio de todo lo que quede por hacer alrededor de las arquitecturas conocidas y sus respectivas variantes, podría aparecer en algún laboratorio la persona que Hennessey & Patterson llaman el *"Von Neumann del siglo XXI"* proponiendo un mecanismo revolucionario completamente diferente y alejado de todos los conocidos. A

modo de ilustración de esta idea me planteo lo siguiente: imaginemos que la mente humana fuese capaz de pensar, evaluar y resolver sólo un problema a la vez, abstraída completamente del entorno que pueda representar el resto de los problemas. Imaginemos que surge luego la idea de poner a cooperar muchas mentes a cada una de las cuales se le asigne sólo una parte de un único gran problema, proveyéndoseles a todas la capacidad de comunicarse entre sí. Es muy probable que el problema global pueda resolverse favorablemente. Es muy probable también que para el problema planteado no existiera mente alguna capaz de resolverlo por sí sola. Ahora imaginemos a la mente tal cual es, esquivando la pregunta inevitable ¿cómo es?. Más allá de la repuesta que cada uno quiera o pueda darle a esta pregunta, lo indudable es que tiene la capacidad de resolver problemas globales del tipo de los encargados al equipo de mentes elementales pero mediante mecanismos que seguramente difieren de cosas como resolver todos los sub-problemas en compartimentos estancos para luego recopilar los resultados y posteriormente reprocesarlos, o por ejemplo hacer avanzar la solución de cada problema permitiendo la comunicación de resultados parciales entre sí, o resolverlos a todos simultánea e independientemente hasta donde sea posible y superar el bloqueo mediante el intercambio de resultados parciales, etc. Las mentes elementales del principio serían el computador secuencial de Von Neumann, la integración de mentes elementales comunicadas, los sistemas de procesamiento paralelo y la mente real el paradigma de procesamiento que todavía no se conoce.

Aceptemos entonces la existencia de dos puntos de vista, bastante contrapuestos. Uno de ellos sería que el procesamiento paralelo, aun cuando es una modalidad cuya utilización lleva décadas, plantea hoy por hoy un dilema como el que presentaba la computación en los años ´50 ó ´60 en los que eran más las dudas que las certezas al respecto, ¿serviría la herramienta para resolver cualquier tipo de problema o sólo una gama reducida y muy específica?, ¿justificaría la relación costo/beneficio la inversión en sistemas de cómputo?, ... El tiempo dio una respuesta contundente a todos estos interrogantes pero en aquel momento no muchos hubiesen sido capaces de predecir los sucesos que luego tuvieron lugar. En el otro extremo está la visión menos optimista, la de los que sin dejar de reconocer los méritos de este modelo, no le auguran de aquí en más un crecimiento tan expansivo como el que suponen le cabría a un mecanismo de cómputo completamente nuevo y diferente respecto a todos los conocidos.

# 8. Bibliografía


[1] John L.Hennessy & David A. Patterson: "Arquitectura de Computadores, Un Enfoque Cuantitativo". (ISBN 84-7615-912-9) Mc Graw - Hill, 1993.

[2] Kai Hwang: "Advanced Computer Architecture, Parallelism Scalability Programmability". (ISBN 0-07-031622-8) Mc Graw - Hill, Series in Computer Science, 1993.

[3] Jorge L. Boria: "Construcción de Sistemas Operativos". (ISBN 950-13-9876-5) Editorial Kapelusz S.A., 1988.

[4] Jeff Heid, Denis Reilly: "Extending Snoopy Cache Coherency Protocols". Project 18-742, Carnegie Mellon University, 1997
(http://www.cs.cmu.edu/afs/ece/class/ee742/www/proj_f97/heid/index.html).

[5] Davor Magdic: "Limes: A Multiprocessor Simulation Environment for PC Platforms", IEEE TCCA Newsletter, 1997.

[6] Norman Matloff: "MulSim Multiprocessor Simulator" -User's Guide-. University of California at Davies, 2000 (http://heather.cs.ucdavis.edu/~matloff/MulSim/MulSimDoc.html).

[7] Bob Grim, Rian Zagelow: "Snoopy Cache Coherence Protocols", ECE 576 - Parallel and Distributed Architectures, Oregon State University, 1997 (http://www.ece.orst.edu/~grim/ece576).