# Análisis Sintáctico Estocástico y Paralelismo*

**Fco. Mario Barcala Rodríguez, Oscar Sacristán Agulló, Jorge Graña Gil**

Departamento de Computación
Universidad de A Coruña
{barcala, agullo, grana}@dc.fi.udc.es

**Resumen** Los algoritmos de análisis sintáctico tipo CYK presentan una naturaleza intrínsecamente paralela: existen muchas celdas de la tabla de análisis que pueden ser calculadas simultáneamente. En este trabajo se realiza un estudio sobre cuál debe ser la técnica de paralelismo adecuada para obtener un rendimiento óptimo del algoritmo CYK extendido, un algoritmo de análisis sintáctico estocástico que mantiene el mismo nivel de expresividad de la gramática original y hace más viables las posteriores tareas de *parsing* robusto. Se consideran dos métodos de paralelización: memoria distribuida y memoria compartida. Los excelentes resultados obtenidos con el segundo de ellos convierten a este algoritmo en una alternativa que podría competir con otras técnicas de *parsing* mucho más eficientes a priori.

## *1 Introducción*

El algoritmo CYK [Kasami 1965, Younger 1967] puede extenderse para trabajar con cualquier tipo de gramática independiente del contexto, no restringiéndose a la representación en forma normal de Chomsky. Una alternativa para conseguir esto es a través de la manipulación de la gramática original, convirtiéndola a forma normal de Chomsky, para lo cual existen diversos algoritmos. Pero el problema fundamental de esta alternativa radica en que es necesaria una reestructuración de la gramática, que en la mayoría de los casos conlleva la aparición de nuevos símbolos no terminales auxiliares y una pérdida importante de comprensión de la misma. Esto provoca que la interpretación de la tabla de análisis sea mucho más compleja, y que los árboles de análisis a los que pueda dar lugar no se correspondan con los árboles lógicos o intuitivos que refleja la gramática original.

Siguiendo los trabajos de [Erbach 1994] y de [Chappelier y Rajman 1998], surge otra alternativa mucho más interesante: readaptar el algoritmo para que pueda trabajar con cualquier gramática independiente del contexto de manera directa.

El nuevo algoritmo, denominado *algoritmo CYK extendido*, maneja tanto gramáticas de contexto libre estocásticas como no estocásticas. Para una frase de longitud $n$ palabras, es posible, por tanto, saber si pertenece o no al lenguaje generado por la gramática, calcular su probabilidad total (no sólo la de sus análisis individuales), obtener los $N$ análisis más probables de dicha frase o de cualquier subsecuencia de la misma en tiempo proporcional a $\mathcal{O}(n^3)$, y todo ello manteniendo la estructura de la gramática original. También es posible la extracción explícita de los árboles de análisis asociados a la frase de entrada o a cualquier subfrase de ella. El coste de este paso depende de la complejidad de la gramática, pero incluso cuando el número de árboles asociados a la frase de entrada crece exponencialmente respecto a $n$, la complejidad de espacio del *chart* (tabla o esquema de análisis) utilizado por el algoritmo para su representación es proporcional a $\mathcal{O}(n^2)$.

Sin embargo, una complejidad temporal proporcional a $\mathcal{O}(n^3)$ podría ser demasiado elevada para determinadas aplicaciones. Esto es lo que motiva una reflexión sobre la construcción de una versión paralela del algoritmo CYK extendido. La sección 2 detalla los pasos estándar de dicho algoritmo, la sección 3 discute las diferentes alternativas de paralelización, la sección 4 muestra los resultados experimentales y en la sección 5 se plantean las conclusiones alcanzadas después del análisis

* Este trabajo ha sido parcialmente financiado por el proyecto 1FD97-0047-C04-02 del FEDER y por el proyecto PGIDT99XI10502B de la *Xunta de Galicia*.

de los datos obtenidos.

## 2 *Algoritmo CYK extendido*

En esta sección, comenzaremos presentando el algoritmo CYK extendido básico, posponiendo momentáneamente las consideraciones estocásticas. Además, a pesar de que el algoritmo puede trabajar con gramáticas independientes del contexto arbitrarias, nos ocuparemos sólo de aquéllas que estén formadas por reglas no parcialmente lexicalizadas. En este tipo de gramáticas, los símbolos terminales aparecen sólo en reglas de la forma $A \rightarrow w_1\, w_2\, \ldots\, w_k$, donde $A$ es un símbolo no terminal y los $w_i$ son símbolos terminales, es decir, palabras. La mayoría de las veces $k$ será igual a 1. Los casos donde $k > 1$ corresponden a las palabras compuestas, expresiones hechas o locuciones. Esta restricción no es crítica para el algoritmo, y se introduce sólo para aislar el procesamiento de las reglas léxicas en el paso de inicialización, que en la práctica se realiza mediante el acceso a un diccionario.

### 2.1 Aproximación no estocástica

La estructura de procesamiento del algoritmo es la tabla de análisis del algoritmo CYK básico, con la excepción de que las celdas contendrán ahora ítems similares a los que se utilizan en el algoritmo de Earley, en lugar de los símbolos no terminales utilizados por el algoritmo original. La tabla de análisis o *chart* es una matriz triangular inferior con $\frac{n(n+1)}{2}$ celdas, donde $n$ es el número de palabras de la frase que se desea analizar. Cada celda $N_{ij}$ de la tabla contiene dos listas de ítems:

- Los ítems de la primera lista de una celda $N_{ij}$, denominada lista tipo 1, representan los símbolos no terminales que derivan la subsecuencia $w_i, w_{i+1}, \ldots, w_{i+j-1}$, es decir, si $A$ es uno de esos símbolos, entonces $A \stackrel{*}{\Rightarrow} w_i w_{i+1} \ldots w_{i+j-1}$, y el ítem correspondiente se denota como $[A; i, j]$.
- Los ítems de la segunda lista de una celda $N_{ij}$, denominada lista tipo 2, representan análisis parciales $\alpha$ de la subsecuencia $w_i, w_{i+1}, \ldots, w_{i+j-1}$, es decir, secuencias $\alpha$ de símbolos no terminales tales que $\alpha \stackrel{*}{\Rightarrow} w_i w_{i+1} \ldots w_{i+j-1}$ para los cuales existe al menos una regla en la gramática de la forma $A \rightarrow \alpha\beta$, donde $\beta$ es una secuencia no vacía de símbolos no terminales ($\beta = \lambda$ es precisamente el caso que consideran los ítems de la lista tipo 1). Estos ítems se denotan como $[\alpha \bullet \ldots; i, j]$.

  Como se puede comprobar, los ítems de la lista tipo 2 representan una generalización de las reglas punteadas utilizadas en el algoritmo de Earley, donde sólo se representa la parte inicial de la regla (es decir, lo que se ha analizado hasta ese momento), y se obvia la parte izquierda (que de hecho aún no ha sido reescrita) y la parte final (que aún no ha sido analizada). Esto proporciona una representación mucho más compacta de las reglas punteadas, que se puede aplicar debido a la naturaleza ascendente del análisis.

Además, a cada ítem de cualquiera de las dos listas se le asocia también la lista de todas sus posibles producciones. Esto permite una factorización que evita la repetición del ítem para cada una de esas producciones, obtiene los ítems una sola vez, aún cuando puedan ser producidos varias veces en la misma celda, y permite una mayor velocidad en el proceso de *parsing*. Para propósitos de extracción de los árboles de análisis, cada producción contiene una referencia explícita a los ítems de las celdas correspondientes que han sido utilizados para crearlos.

El algoritmo de análisis sintáctico CYK extendido, para determinar si una frase $s = w_1, w_2, \ldots, w_n$ pertenece al lenguaje generado por una gramática independiente del contexto, con reglas no parcialmente lexicalizadas, $G = (N, T, P, S)$, consta de los siguientes pasos:

1. **Paso de inicialización**. Este paso consiste en rellenar todas las celdas de la tabla de análisis para las cuales existe una regla léxica asociada con las palabras o secuencias de palabras de la frase de entrada. Es decir, se rellenan las listas tipo 1 de las celdas de las primeras filas de la tabla. Más concretamente, si una regla $A \rightarrow w_i \ldots w_{i+j-1} \in P$, el ítem $[A; i, j]$ se añade a la lista tipo 1 de la celda $N_{ij}$.

   Para completar el paso de inicialización, se necesita una fase de autorrellenado, que actualice las listas tipo 2 de estas celdas. Esta fase también se utiliza en el paso de análisis que explicamos a continuación.

2. **Paso de análisis**. Este paso consiste en aplicar dos fases, la fase de rellenado estándar, que afecta a todas las celdas de la tabla, excepto a las afectadas por el paso de inicialización, y la fase de autorrellenado, que afecta a todas las celdas $N_{ij}$ de la tabla. Dichas fases se aplican fila a fila, de forma ascendente.

    - **Fase de rellenado estándar**. Se combina un ítem $[\alpha \bullet \ldots; i, k]$ de la lista 2 de una celda, con un ítem $[B; i+k, j-k]$ de la lista tipo 1 de otra celda ($1 \leq k < j$), si y sólo si existe una regla de la forma $A \to \alpha B \beta$ en la gramática. Si $\beta = \lambda$, el ítem $[A; i, j]$ se añade a la lista tipo 1 de la celda $N_{ij}$. En caso contrario, se añade el ítem $[\alpha B \bullet \ldots; i, j]$ a la lista tipo 2 de la celda $N_{ij}$.

      Hay que tener en cuenta que se añade el ítem completo siempre y cuando éste no exista ya. En el caso de que el ítem generado ya exista, sólo se añade la nueva producción a la sublista de producciones del mismo.

    - **Fase de autorrellenado**. Esta fase es un procedimiento mediante el cual, para cada ítem $[B; i, j]$ de la lista tipo 1 de cada celda, y para cada regla de la forma $A \to B\beta$, se hace lo siguiente:

        - Si $\beta = \lambda$, se añade el ítem $[A; i, j]$ a la lista tipo 1 de la celda. Nótese que sobre este nuevo ítem generado se debe aplicar nuevamente la fase de autorellenado.
        - Si $\beta \neq \lambda$, se añade el ítem $[B \bullet \ldots; i, j]$ a la lista tipo 2 de la celda.

      Esta segunda fase es necesaria para considerar el encadenamiento de reglas unitarias de la forma $A \to B$, y mantener así las listas de tipo 2 actualizadas después de que la fase de rellenado estándar haya sido realizada.

Por lo tanto, $s \in L(G)$ si el ítem $[S; 1, n]$ está en la celda $N_{1n}$.

## 2.2 Transiciones $\lambda$

Si la gramática contiene reglas $\lambda$, es decir, reglas de la forma $A \to \lambda$, el algoritmo debe completarse con las dos siguientes consideraciones:

- Cuando se produce un ítem $[\alpha \bullet \ldots; i, j]$ en la lista tipo 2 de una celda, para cada no terminal $B$ que produzca $\lambda$ ($B \to \lambda \in P$), y que pueda completar ese ítem ($A \to \alpha B \beta \in P$), se hace lo siguiente:

    - Si $\beta = \lambda$, se añade el ítem $[A; i, j]$ a la lista tipo 1 de la misma celda.
    - Si $\beta \neq \lambda$, se añade el ítem $[\alpha B \bullet \ldots; i, j]$ a la lista tipo 2 de la misma celda.

  Este proceso debe ser realizado recursivamente sobre los ítems nuevos que se van generando, hasta que no se puedan añadir más.

- La fase de autorrellenado debe ser modificada de manera que, para cada ítem $[B; i, j]$ de la lista tipo 1 de una celda, si existe una regla de la forma $A \to CB\beta$, donde $C \to \lambda$, entonces:

    - Si $\beta = \lambda$, se añade el ítem $[A; i, j]$ a la lista tipo 1 de la misma celda.
    - Si $\beta \neq \lambda$, se añade el ítem $[B \bullet \ldots; i, j]$ a la lista tipo 2 de la misma celda.

  Es decir, se procede como si realmente la regla $A \to B\beta$ estuviera también en la gramática.

## 2.3 Extracción de los árboles de análisis

La extracción de los árboles de análisis a partir de la tabla se realiza simplemente recorriendo las producciones de los ítems de tipo 1 de la celda superior que contengan el símbolo inicial de la gramática, si es que existe alguno. En todo caso, este procedimiento es también aplicable a cualquier otra celda para la extracción de subárboles.

Cada producción de estos ítems se considera como un árbol de análisis, y para cada uno de estos árboles o producciones del ítem elegido como raíz, se realiza un recorrido recursivo de manera que: (1) Se crea el nodo raíz, etiquetado con el símbolo del ítem elegido. (2) Por cada ítem al que se llega en el recorrido se genera un nodo, hijo del nodo generado por el ítem origen. Si el ítem es de tipo 1, se etiqueta el nodo con el símbolo incluido en dicho ítem. Por el contrario, si el ítem es de tipo 2, este nodo no se etiqueta. (3) Los nodos no etiquetados se eliminan, haciendo que sus hijos etiquetados pasen a ser hijos de su primer ancestro etiquetado. (4) Se añaden las palabras de la frase.

### 2.4 Consideraciones estocásticas

Para las gramáticas estocásticas, la probabilidad de un árbol de análisis es el producto de la probabilidad de la regla utilizada para generar los subárboles del nodo raíz, y de las probabilidades de cada uno de esos subárboles[1].

En el caso del algoritmo CYK extendido, se puede definir la probabilidad de una producción $p$ de un ítem tipo 1 $[A; i, j]$, como la probabilidad del subárbol de análisis correspondiente a la interpretación $A$ de la secuencia $w_i, \ldots, w_{i+j-1}$ para esa producción $p$, o lo que es lo mismo, como el producto de las probabilidades de los ítems a partir de los cuales se genera, y la probabilidad de la regla involucrada en dicha generación.

De la misma forma, para los ítems de tipo 2 se puede definir la probabilidad de una producción $p$ de un ítem $[\alpha \bullet \ldots; i, j]$ como el producto de las probabilidades de los ítems que la generan. En este caso no se multiplica por la probabilidad de ninguna regla, ya que los ítems de tipo 2 consideran interpretaciones parciales.

A partir de esto, es posible definir la probabilidad máxima para un ítem como el máximo de las probabilidades de cada una de las producciones, evitando así mantener las probabilidades de todas, y teniendo en cuenta que una producción de un ítem se obtiene combinando solamente dos ítems ya existentes (uno de tipo 2 con otro de tipo 1). Esto plantea la ventaja de que se pueden mantener los $N$ análisis más probables durante la construcción de la tabla, ya que los $N$ valores más grandes para un ítem están incluidos entre los $N^2$ productos de los $N$ análisis más probables de cada uno de los dos ítems que intervienen en su generación. Si a posteriori estos $N$ análisis más probables no son suficientes, se pueden extraer recursivamente todos los análisis o subanálisis de una frase de entrada, calculando sus probabilidades durante el proceso de extracción a partir de las probabilidades de sus constituyentes.

Además, la probabilidad de cualquier frase o de cualquier subsecuencia de la misma, definida como la suma de las probabilidades de todos sus análisis, se calcula también durante el proceso de construcción del *chart*. Esto se realiza simplemente sumando siempre las probabilidades de cada nueva producción de un ítem, incluso aunque no vaya a ser mantenida entre las $N$ mejores.

## *3 Consideraciones de paralelismo*

La complejidad temporal del algoritmo CYK extendido, medida en términos de $n$, la longitud de la frase de entrada, es claramente proporcional a $\mathcal{O}(n^3)$. Sin embargo, la constante multiplicativa asociada a $n^3$, en este caso, juega un importante papel. Se podría calcular una complejidad más detallada para el peor caso de análisis, en función del cardinal de los conjuntos de no terminales y de reglas, y veríamos que es menor que la de otros algoritmos de *parsing* similares. No vamos a desarrollar este cálculo aquí (puede verse en detalle en [Chappelier y Rajman 1998]), pero la explicación intuitiva es que la ganancia que se produce, respecto a otros algoritmos del tipo Earley o del tipo CYK, es debida al hecho de que el algoritmo CYK extendido realiza una mejor factorización de las reglas punteadas (en los ítems de las listas de tipo 2), y procesa menos elementos (los ítems aparecen sólo una vez y no se hacen predicciones).

Aún con todo esto, dicha complejidad se puede rebajar, debido a la naturaleza intrínsecamente paralela de los algoritmos tipo CYK: existen muchas celdas de la tabla de análisis que pueden ser calculadas simultáneamente. Para la paralelización del algoritmo CYK extendido [Barcala y Graña 1999], se han planteado dos alternativas diferentes, barajando las dos tendencias principales que existen actualmente en el procesamiento paralelo: (1) memoria distribuida, en este caso, mediante paso de mensajes entre distintos computadores, y (2) memoria compartida, es decir, ejecución

[1] Mediante el uso de probabilidades logarítmicas, se podrían cambiar los productos por sumas, lo cual aceleraría los cálculos y evitaría los problemas de pérdida de precisión resultantes de multiplicar números menores que 1.

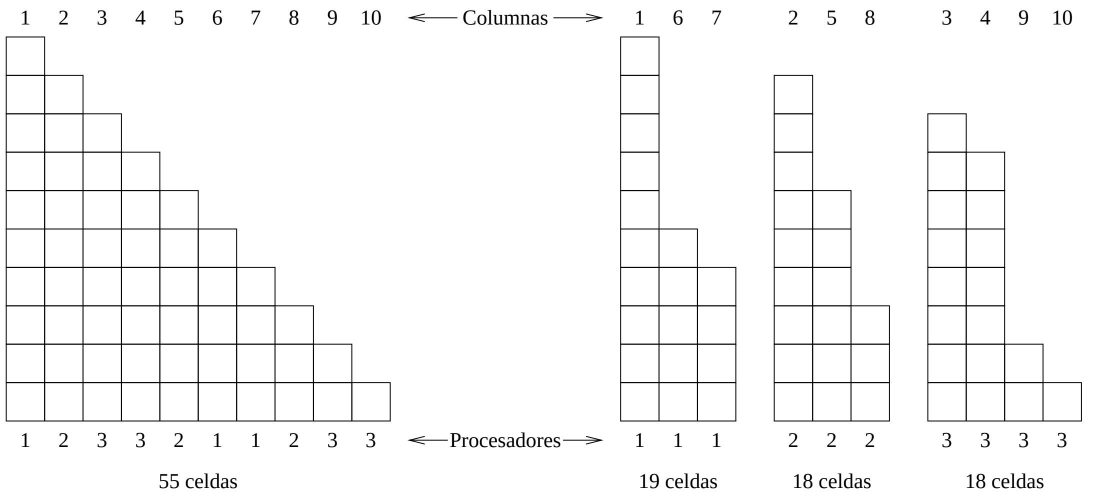


Figura 1: Reparto de columnas de la tabla de análisis en una paralelización del algoritmo CYK extendido mediante memoria distribuida

de varios procesos que acceden concurrentemente a la misma memoria física. A continuación se exponen ambas aproximaciones.

## 3.1 Memoria distribuida

En esta alternativa se plantea la posibilidad de que cada máquina calcule un fragmento de la tabla de análisis, de manera que ésta quede repartida por la memoria de todos los computadores utilizados. Una posibilidad es que cada máquina calcule una o varias columnas de la tabla, de manera que el paralelismo se afronta a nivel de columna. Este reparto presenta la ventaja de que cada procesador siempre tiene una celda de las que le hacen falta para calcular la siguiente. Un inconveniente consiste en que una vez que un procesador termina con sus columnas, no ayuda a realizar el trabajo de los demás. Por otra parte, si existen más procesadores que palabras en la frase, los procesadores sobrantes no realizan ningún trabajo.

Bajo esta perspectiva, sólo queda por determinar cómo se hace el reparto de las columnas entre las máquinas y cómo es el intercambio de mensajes entre los procesadores. Con respecto al reparto, lo más razonable es efectuar una distribución cíclica invertida, de tal manera que si se tienen, por ejemplo, 10 columnas y 3 procesadores, se asignan las columnas 1, 6, y 7 al procesador 1, las columnas 2, 5, y 8 al procesador 2, y las columnas 3, 4, 9 y 10 al procesador 3, tal y como se muestra en la parte izquierda de la figura 1. De esta forma, se obtiene una distribución balanceada de la carga de procesamiento entre las máquinas. La parte derecha de la figura muestra las subtablas que mantendría en memoria cada procesador.

En cuanto a las comunicaciones, si los procesadores envían cada celda, una vez calculada, a los demás a los que les hará falta para calcular celdas posteriores, se elimina la necesidad de establecer una comunicación bidireccional de petición y envío de celdas entre los procesadores. No habrá peticiones, sino sólo envíos. Además, dichos envíos se pueden realizar una sola vez a cada procesador, de manera que el destino, cuando recibe una celda, debe determinar si le hará falta otra vez para calcular otra celda de otra columna, en cuyo caso debe envíarse a sí mismo la celda nuevamente.

Sin embargo, esta alternativa daría más prioridad a la rapidez de cálculo que a la fiabilidad. Efectivamente, con la supresión de las comunicaciones bidireccionales se produce un aumento considerable de la velocidad de procesamiento. Cada procesador busca las celdas que necesita en su cola de entrada. Si no encuentra una determinada celda, será porque el procesador responsable de calcularla aún no ha terminado su tarea y aún no se la ha enviado, con lo cual se producirá una espera momentánea hasta que dicho cálculo termine. Pero existe el problema de que un procesador puede agotar su capacidad de almacenamiento de celdas. Dicho problema se acentúa cuando se utilizan pocos procesadores, pudiendo producirse un bloqueo general del algoritmo.

Incluso con esta optimización del intercambio de mensajes y aunque no se produzcan bloqueos, el gran problema de la aproximación con memoria distribuida es la *granularidad* del proceso que se quiere paralelizar. En nuestro caso, si la cantidad de información que reside en las celdas es muy grande, incluso aunque no existan demasiadas esperas, es muy probable que los procesadores dediquen más tiempo al envío y recepción de los datos, que al cálculo de los mismos. Puede ocurrir que la ganancia producida por la paralelización no sea suficientemente significativa con respecto a los recursos computacionales utilizados, o incluso que el mismo proceso se ejecute más rápido utilizando una sola máquina secuencial. Todas estas reflexiones nos llevan al estudio de la alternativa con memoria compartida.

## 3.2 Memoria compartida

En una paralelización mediante memoria compartida, todos los procesadores comparten la misma tabla de análisis, y deben ir rellenando sus celdas simultáneamente. Al contrario que en el caso anterior, las celdas se van calculando de izquierda a derecha y de abajo a arriba, independientemente de los procesadores y a medida que éstos van quedando libres, hasta rellenar toda la tabla. De esta manera no queda ningún procesador libre antes de tiempo y hay un mayor aprovechamiento de la capacidad de procesamiento de cada uno de ellos. Esta alternativa no resulta viable con memoria distribuida, ya que la sincronización de todos los procesadores supone un coste de comunicación muy elevado.

Una característica importante inherente al paradigma de paralelismo con memoria compartida es que no hay transferencia de datos entre los procesadores. Esto supone una gran ventaja en casos como el que nos ocupa, donde la cantidad de información que se desea compartir es elevada. Sin embargo, hay que tener en cuenta que los procesadores deben acceder concurrentemente a las mismas zonas de memoria, por lo que hay que determinar las secciones críticas y realizar una protección adecuada sobre ellas. En nuestro caso, es suficiente con definir una variable asociada a cada celda de la tabla, que indique si está ya calculada o no, y un semáforo asociado a dicha variable.

Hay que determinar que el orden de los lazos que determinan qué celda se procesa es importante para el adecuado comportamiento del analizador, y debe ajustarse perfectamente a los requerimientos de esta alternativa. Así, el lazo externo debe ser el de las filas, y el interno el de las columnas. De esta manera, no se producirán situaciones de interbloqueo, es decir, situaciones donde un proceso necesite una casilla que está calculando otro, y viceversa, que el otro necesite la casilla que está calculando el primero. Cuando un procesador comienza a calcular una celda, las celdas que va a necesitar, o ya están calculadas, o se están calculando, y ningún procesador que las esté calculando va a necesitar la celda del primero, ya que ésta está por encima o como mucho al mismo nivel. Por tanto, está garantizado que esta alternativa de paralelización no producirá abrazos mortales.

De todo esto se puede deducir que el algoritmo de análisis sintáctico CYK extendido presenta un esquema de paralelización natural. Aunque como veremos mas adelante, no es apropiado implementarlo sobre un multicomputador de memoria distribuida, ya que la penalización en las comunicaciones afecta mucho a su eficiencia, la implementación en un multiprocesador considerando varios *threads* (hilos de ejecución) aportará unos resultados sorprendentes.

# *4 Análisis de resultados*

Los experimentos de paralelismo se han realizado sobre un computador con sistema operativo Linux, formado por los siguientes componentes:

- 1 nodo principal con 2 procesadores Pentium II a 350 MHz., y con 384 MB de memoria RAM (multiprocesador de memoria compartida).
- 23 nodos adicionales AMD K6 a 300 MHz., con 96 MB de memoria RAM, formando un *cluster* a través de un *switch fast ethernet* (multicomputador de memoria distribuida).

Las versiones secuencial y *multithread* se ejecutan en el nodo central. La versión de memoria distribuida se ejecuta repartidamente por todos los nodos, incluido el nodo principal, e implementa la comunicación entre los procesadores mediante la librería estándar MPI (*Message Passing Interface*).

Los datos sobre los que se ha evaluado el algoritmo han sido tomados del *treebank* SUSANNE [Sampson 1994a, Sampson 1994b], el cual ha sido dividido en dos partes: la prime-

| Análisis de 2.188 frases | | | | | | | |
|---|---|---|---|---|---|---|---|
| Tiempos | Secuencial | 1 *thread* | 2 *thread* | 2 MPI | 4 MPI | 8 MPI | 16 MPI |
| Real | 03:45.259 | 03:52.648 | 02:24.311 | 10:47.108 | 09:28.020 | 07:57.424 | 07:40.391 |
| Usuario | 03:44.810 | 03:50.580 | 03:53.040 | 05:08.900 | 03:55.730 | 02:36.370 | 02:07.190 |
| Sistema | 00:00.036 | 00:00.530 | 00:01.370 | 05:16.200 | 05:26.370 | 05:17.350 | 05:28.990 |

Tabla 1: Tiempos de ejecución del algoritmo CYK extendido sobre el *treebank* SUSANNE

ra formada por las frases sin trazas (4.292 frases, 77.275 palabras), y la segunda formada por las frases con trazas (2.188 frases, 60.759 palabras). La primera parte ha sido utilizada para extraer una gramática (compuesta por 17.669 reglas y 1.525 símbolos no terminales) [Graña *et al.* 1999], y la segunda para medir los tiempos de análisis de las diferentes versiones del algoritmo.

Los resultados de la tabla 1 presentan tiempos relacionados con: secuencial, 1 *thread*, 2 *threads*, 2 MPI, 4 MPI, etc. Cada uno de estos descriptores se corresponde, respectivamente, con la versión secuencial, la versión *multithread* con un solo *thread* (de esta forma se deduce el coste de inicialización y manejo de los semáforos), la versión *multithread* con 2 *threads*, y las versiones de memoria distribuida con 2 procesadores, 4 procesadores, etc. Los tiempos se muestran en formato `mm:ss.mmm` (minutos, segundos y milésimas de segundo). Los tiempos de usuario para 2 *threads* representan la suma de los tiempos de ejecución de los dos procesadores. Por tanto, en este caso, aunque sea menos fiable, es más representativo el tiempo real.

A través de estos datos se observa que existe un coste adicional por el manejo de semáforos, pero que aún así la versión *multithread* con 2 *threads* es considerablemente más rápida que la versión secuencial. Además, la implementación evaluada considera el paralelismo a nivel de frase, por lo que los *threads* se crean y destruyen para cada una de las frases (con el coste que ello supone), pudiendo optimizarse aún más si se mantienen los *threads* entre frase y frase.

Por otra parte, la versión con memoria distribuida parece la peor en cualquier caso, y esto se debe a tres razones principales, alguna de las cuales ya ha sido comentada anteriormente. En primer lugar, para frases pequeñas, el coste de comunicación de los datos entre los diferentes procesadores es elevado. En segundo lugar, cuando la frase es grande, debería existir una repartición equilibrada de datos entre las diferentes columnas, quedando equilibrado el cálculo entre los procesadores, pero esto no es lo usual debido a la asimetría de las tablas de análisis, que típicamente contienen gran cantidad de celdas vacías. Y por último, cuando la frase es compleja, las celdas suelen contener demasiada información de manera que resulta muy cara la comunicación entre los diferentes procesadores. Puede observarse que al aumentar el número de procesadores, disminuye el tiempo de ejecución, pero aumenta el tiempo del sistema, que en este caso incluye, fundamentalmente, el tiempo de comunicación entre los diferentes procesadores.

Existe una ganancia obvia de la implementación con 2 *threads*: un tiempo de análisis de 2:24.311 sobre 2.188 frases implica una media aproximada de 0,065 segundos por frase. Pero este dato tampoco es representativo en absoluto. Para demostrarlo, basta echar un vistazo a la tabla 2, en la cual se estudian los tiempos de ejecución de 5 frases de manera independiente. Hay frases que se analizan en tiempos muy superiores a la media, que oscilan entre 318 milésimas de segundo y casi dos minutos. Algo a destacar también es el comportamiento sobre la frase 5, donde la aproximación mediante *threads* no produce una mejora significativa del rendimiento. Esto se debe a que muy probablemente en la tabla de análisis de esta frase haya muchas celdas vacías, y de hecho es así, ya que el tiempo de análisis es realmente bajo para tratarse de una frase de 214 palabras (23.005 celdas). Pero la existencia de pocas celdas con ítems hace que las esperas entre los procesadores sean muy frecuentes. Es decir, el paralelismo se explota mucho mejor cuanto más compleja sea la frase a analizar.

Lo que sí resulta también evidente es que la complejidad de una frase no es proporcional al número de palabras de la misma, lo que queda claro comparando los tiempos de análisis de las frases 3 y 4, y que, en todo caso, la mejora de la versión de 2 *threads* res-

| | Frase 1 | Frase 2 | Frase 3 | Frase 4 | Frase 5 |
|---|---|---|---|---|---|
| Palabras | 30 | 35 | 55 | 150 | 214 |
| Tiempos en Secuencial | | | | | |
| Real | 00:00.351 | 00:04.646 | 02:40.821 | 00:00.750 | 00:22.268 |
| Usuario | 00:00.340 | 00:04.630 | 02:40.600 | 00:00.740 | 00:22.160 |
| Sistema | 00:00.010 | 00:00.020 | 00:00.220 | 00:00.010 | 00:00.100 |
| Tiempos en 2 *threads* | | | | | |
| Real | 00:00.318 | 00:02.735 | 01:57.759 | 00:00.623 | 00:21.004 |
| Usuario | 00:00.460 | 00:04.810 | 02:39.800 | 00:00.950 | 00:23.480 |
| Sistema | 00:00.001 | 00:00.020 | 00:00.140 | 00:00.030 | 00:00.200 |

Tabla 2: Tiempos de ejecución independientes del algoritmo CYK extendido sobre 5 frases

pecto a la secuencial sigue presente.

## 5 Conclusiones

En resumen, la paralelización del algoritmo CYK extendido mediante memoria compartida lo convierte en una alternativa que podría competir con otras técnicas de *parsing* mucho más eficientes. Si bien esas otras técnicas también podrían ser paralelizables, esta paralelización, en caso de que sea posible, no será en absoluto inherente al propio algoritmo, y dará lugar a una complejidad de diseño e implementación mucho mayor.

Por otra parte, no hemos resaltado suficientemente las ventajas de la combinación del paradigma estocástico con las técnicas de *parsing* robusto. Efectivamente, se trata de dos características que aparecen también de manera natural en el algoritmo CYK extendido, y que actualmente se requieren en muchas aplicaciones de procesamiento de lenguaje natural. La disponibilidad de herramientas eficientes de análisis léxico y sintáctico capaces de enfrentarse a diccionarios y gramáticas incompletos con la ayuda del marco estadístico, abre perspectivas de aplicación inmediata en sistemas de tratamiento de información a alto nivel (recuperación, extracción, traducción, ...).

## Referencias


[Barcala y Graña 1999] Barcala Rodríguez, M. (1999). Diseño e implementación de una herramienta de análisis sintáctico estocástico para el procesamiento de textos en lenguaje natural. *Proyecto Fin de Carrera en Ingeniería Informática*, dirigido por J. Graña Gil en el Departamento de Computación de la Universidad de A Coruña.

[Chappelier y Rajman 1998] Chappelier, J.-C.; Rajman, M. (1998). A practical bottom-up algorithm for on-line parsing with stochastic context-free grammars. *Rapport Technique 98/284*, Departement d'Informatique, Ecole Polytechnique Fédérale de Lausanne (Suisse).

[Earley 1970] Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, vol. 13(2), pp. 94-102.

[Erbach 1994] Erbach, G. (1994). Bottom-up Earley deduction. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-94)*, Kyoto (Japan).

[Graña *et al.* 1999] Graña Gil, J.; Chappelier, J.-C.; Rajman, M. (1999). Using syntactic constraints in natural language disambiguation. *Rapport Technique 99/315*, Departement d'Informatique, Ecole Polytechnique Fédérale de Lausanne (Suisse).

[Kasami 1965] Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. *Technical Report*, AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

[Sampson 1994a] Sampson, G. (1994a). The SUSANNE corpus, release 3, 04/04/1994. School of Cognitive & Computing Sciences, University of Sussex, Falmer, Brighton, England.

[Sampson 1994b] Sampson, G. (1994b). English for the computer. *Oxford University Press.*

[Younger 1967] Younger, D.H. (1967). Recognition of context-free languages in time $n^3$. *Information and Control*, vol. 10(2), pp. 189-208.