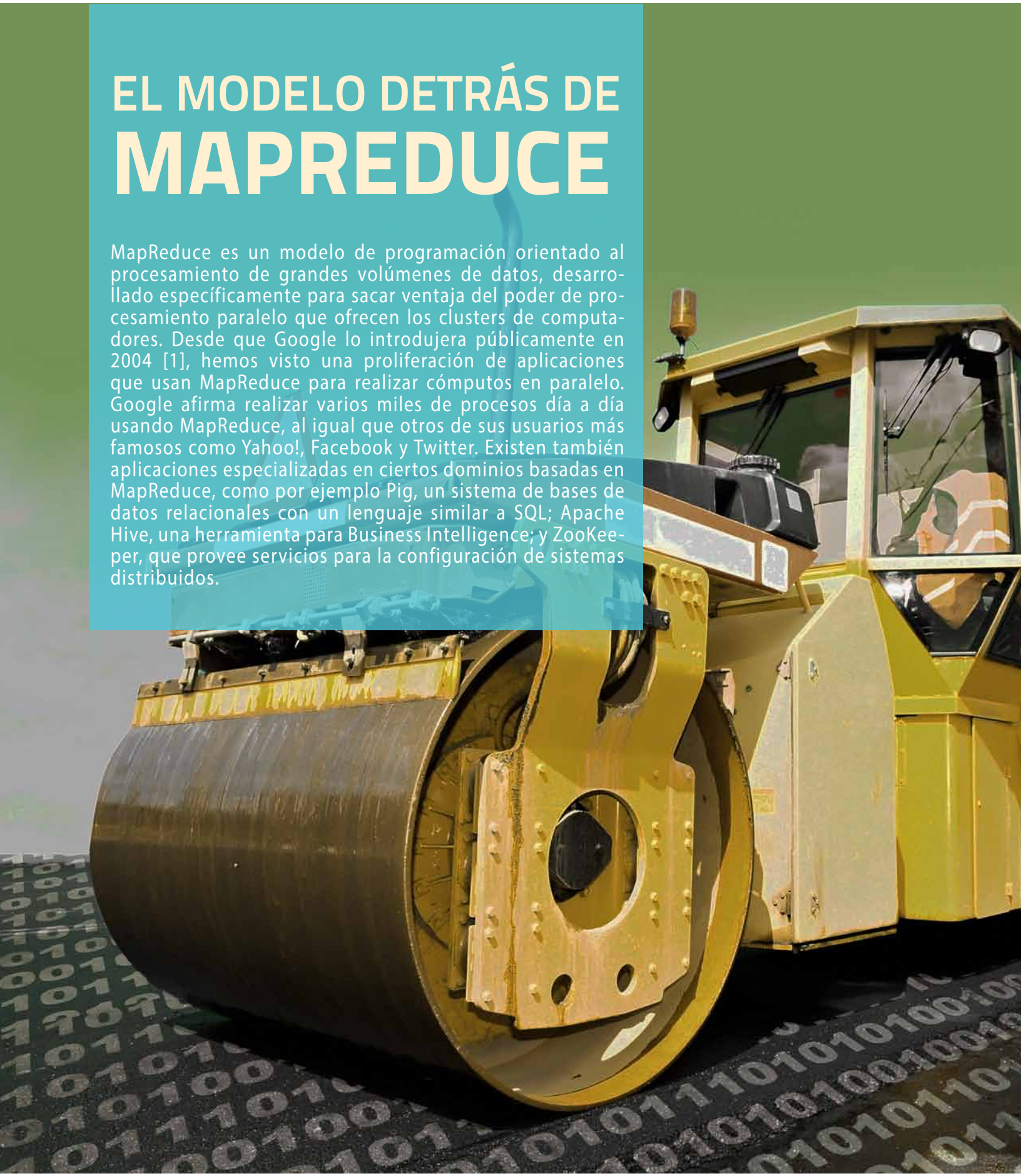
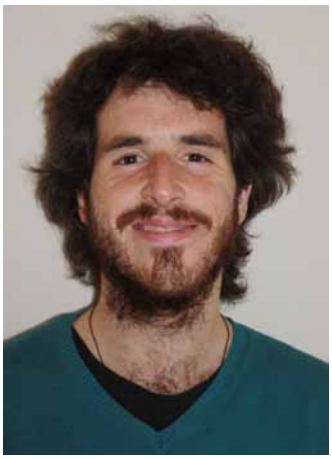
# EL MODELO DETRÁS DE MAPREDUCE

MapReduce es un modelo de programación orientado al procesamiento de grandes volúmenes de datos, desarrollado específicamente para sacar ventaja del poder de procesamiento paralelo que ofrecen los clusters de computadores. Desde que Google lo introdujera públicamente en 2004 [1], hemos visto una proliferación de aplicaciones que usan MapReduce para realizar cómputos en paralelo. Google afirma realizar varios miles de procesos día a día usando MapReduce, al igual que otros de sus usuarios más famosos como Yahoo!, Facebook y Twitter. Existen también aplicaciones especializadas en ciertos dominios basadas en MapReduce, como por ejemplo Pig, un sistema de bases de datos relacionales con un lenguaje similar a SQL; Apache Hive, una herramienta para Business Intelligence; y ZooKeeper, que provee servicios para la configuración de sistemas distribuidos.






## JUAN REUTTER

Profesor Asistente del Departamento de Ciencia de la Computación de la Pontificia Universidad Católica de Chile. Recibió su Doctorado en la Universidad de Edimburgo en mayo de 2013. Sus intereses se enmarcan en el área de Manejo de Datos, incluyendo Sistemas de Bases de Datos, Lenguajes de Consulta, Web Semántica y Lenguajes Formales.

**jreutter@ing.puc.cl**


MapReduce ha sido llamado un "cambio de paradigma" en Computación [2]. Sus adeptos no dudan en asegurar que el uso de MapReduce es capaz de acelerar los cómputos de cualquier aplicación. Algunos incluso han llegado a afirmar que MapReduce va a dejar obsoletos a los sistemas de bases de datos relacionales. Y las cifras parecen avalar este cambio: la firma Cloudera afirma que al año 2013 la mitad de las compañías en el listado Fortune 50 (un listado de las compañías más importantes de los Estados Unidos) usa MapReduce en alguno de sus procesos [3].

A pesar de esta aparente colonización de MapReduce en aplicaciones que lidian con grandes volúmenes de datos, también hay quienes ponen en duda su eficacia. En efecto, los detractores de MapReduce terminan siendo igual de duros que sus adeptos, y han llegado a afirmar que "MapReduce es un gran paso hacia atrás (para la comunidad de manejo de datos)". Es más, estudios de la Universidad de Cornell afirman que la solución ofrecida por MapReduce es generalmente menos eficiente que la obtenida al usar otros modelos más clásicos de computación paralela [4]. Académicos de la Universidad de Brown también han llegado a una conclusión similar: comparando sistemas de bases de datos tradicionales con sistemas basados en MapReduce, concluyeron que los sistemas tradicionales consiguen un mejor rendimiento en casi todas las tareas, con excepción de una familia particular de tareas llamadas Extraer, Transformar y Cargar (ETL por sus siglas en inglés) [5].

Esta controversia se genera porque nadie conoce con certeza en qué casos MapReduce ofrece una aceleración significativa a la hora de realizar cómputos. Por lo mismo, es difícil saber cuánto de la popularidad de MapReduce se debe a las ventajas computacionales de su arquitectura, cuánto se debe a su simpleza y cuánto se debe al bombardeo publicitario que sus desarrolladores continúan llevando a cabo.

En el pasado la Ciencia de la Computación logró resolver problemas similares en otros contextos, mediante el desarrollo de teorías capaces de entender y modelar el comportamiento de distintos paradigmas de computación. De la misma forma, para poder hacer un análisis más preciso de las ventajas y desventajas de esta arquitectura y el tipo de problemas para los que es beneficioso usar MapReduce, necesitamos un modelo formal que sea capaz de predecir el comportamiento de las aplicaciones MapReduce en la práctica. En este artículo discutiremos las características mínimas que debe tener este modelo y hacia dónde tiene que avanzar la comunidad científica para lograrlo.

# EL MODELO DE MAPREDUCE

El modelo de MapReduce es simple. Existen dos funciones, llamadas Map y Reduce, dividiendo la computación en dos etapas: primero se ejecuta un número de llamadas a Map (en forma paralela); tras lo cual los datos entregados por estas funciones son recolectados y ordenados,

→

y luego con estos datos se ejecuta un número de llamadas a Reduce.

Más específicamente, y siguiendo el modelo de Rajaraman y Ullman [6], podemos resumir una ejecución de MapReduce de la siguiente forma:

1. El sistema solicita un número de funciones Map y entrega a cada una de éstas un pedazo de los datos a procesar (que generalmente provienen de un sistema de manejo de datos distribuidos). Cada función Map transforma estos datos en una secuencia de pares (llave,valor).

2. Un controlador maestro recolecta todos los pares (llave,valor) de todas las funciones Map y los ordena de acuerdo a sus llaves. Luego distribuye todas las llaves sobre varias funciones Reduce, de forma que todos los pares (llave,valor) que tienen la misma llave van a parar a una misma copia de Reduce.

3. Cada Reduce toma todos los valores que recibe para una misma llave y realiza algún tipo de cómputo con los valores que recibe.

4. El resultado final de la operación es la combinación de los resultados de cada una de las funciones Reduce.

A modo de ejemplo, imaginemos que tenemos un archivo de texto y queremos contar cuántas veces se repite cada palabra en este documento. La función Map debería recibir un pedazo de este documento, separarlo en palabras distintas, y emitir, para cada ocurrencia de una palabra *w* en el texto, el par (*w*,1), donde *w* es la llave, y el valor corresponde a 1. La función Reduce entonces recibirá una secuencia de pares (llave,valor) idénticos (*w*,1),...,(*w*,1), y debe contar el número de ocurrencias de este par que recibe, lo que corresponde exactamente a la cantidad de veces que aparece la palabra *w* en el documento. Como el controlador llama a una función Reduce por cada palabra distinta en el documento, una vez finalizado el proceso de cómputo cada una de las funciones Reduce nos entregará el número de ocurrencias de una palabra en particular.

## MAPREDUCE PARA TRABAJAR GRANDES VOLÚMENES DE DATOS

Como ya hemos mencionado, nos interesa construir un modelo teórico que sea capaz de clasificar a los problemas de acuerdo a la eficacia con que pueden ser resueltos usando MapReduce, dividiéndolos en dos grupos: aquellos para los que es evidentemente beneficioso usar MapReduce, y aquellos para los que no es así.

Para comenzar es necesario mirar dónde están los cuellos de botella de MapReduce. Volvamos a nuestro problema de contar las palabras del documento. En este caso, cada mapper procesará solo un pequeño pedazo del documento, contará las ocurrencias de las palabras de su pedazo y las enviará a los *reducers* para que agrupen la información. Si asumimos que estamos procesando un texto en lenguaje natural, la comunicación estará bien distribuida, ya que vamos a invocar a una función Reduce distinta por cada una de las palabras del documento.

Otro ejemplo típico de la utilidad de MapReduce es el *join* de dos relaciones en una base de datos relacional. Imaginemos que tenemos una relación *Teléfono*, con atributos *nombreUsuario* y *numTelefono*, que almacena nombres de personas y sus números de teléfono; y otra relación Dirección, con atributos *numTelefono* y *direccionTelefono*, que almacena la dirección asociada a cada número telefónico. Si queremos computar los nombres de las personas junto a sus direcciones, tenemos que hacer un *join* entre las tablas Teléfono y Dirección, uniendo los registros asociados al mismo número telefónico. Podemos resolver este problema fácilmente en MapReduce: la función map identifica como llave a los números telefónicos en las tablas Teléfono y Dirección, de forma que la función Reduce reciba los nombres y las direcciones de todos los registros asociados a un mismo teléfono y compute la respuesta de manera local. Nuevamente, la comunicación enviada por cada *mapper* estará dividida en varios *reducers* (uno por cada teléfono) y los *reducers* recibirán pedazos pequeños de la base de datos.

Los dos problemas que hemos visto tienen un punto en común: en ambos casos el cómputo puede ser efectivamente dividido en múltiples pedazos, y luego fácilmente vuelto a reunir. Pero, ¿qué pasa cuándo no tenemos esa garantía, cuando el problema no se puede dividir?

Consideremos por ejemplo el problema de conectividad de un grafo: se tiene un gran número de nodos, los que están conectados entre sí mediante aristas. El problema de conectividad busca saber si un determinado nodo $n_1$ está conectado a otro nodo $n_2$. Este problema es importante, por ejemplo, en redes sociales: los nodos representan personas, las aristas son las relaciones entre estas personas, e interesa saber si una persona está "conectada" a otra.

¿Cómo podemos usar MapReduce para resolver el problema de conectividad? Este problema ha sido estudiado con bastante entusiasmo y aún no hay una respuesta clara. El consenso es que, en general, utilizar MapReduce para resolver conectividad no es una buena alternativa. De hecho, si nos ponemos en el peor de los casos, puede que el camino entre ambos nodos sea extremadamente largo, y los nodos y relaciones que participan en el camino entre $n_1$ y $n_2$ estén todos divididos en *mappers* distintos. La única solución parece ser juntar todos los pedazos en un mismo *reducer*, pues necesitamos de todo el grafo para resolver el problema. Si nuestro grafo es una gran red social, ¡esto es tremendamente ineficiente!

A través de estos ejemplos hemos observado dos conclusiones clave para nuestro análisis: por un lado, los problemas en que podemos dividir el cómputo en pedazos pequeños y luego unirlos directamente parecen darse bien en MapReduce. Pero además nos interesa restringir la comunicación entre cada *mapper* o cada *reducer*, de forma que todos los datos no vayan a parar a un número muy pequeño de *reducers*.

De esta forma, nuestro modelo teórico debe tomar en cuenta estas conclusiones a la hora de clasificar los problemas. Pero lamentablemente las teorías actuales que intentan discernir cuando un problema es o no paralelizable no son satisfactorias en nuestro contexto. Para empezar, los modelos teóricos actuales de computación paralela se basan casi siempre en una arquitectura en la que todos los nodos tienen acceso gratis al conjunto de los datos (llamadas *share everything*, o compartirlo todo). En cambio MapReduce es una arquitectura en la que los *mappers* no tienen más acceso que a su pedazo de input y los *reducers* tienen cada uno la información asociada a una llave (es prácticamente una arquitectura tipo *shared nothing,* o compartir nada). En complejidad computacional, por otro lado, se asocia la característica de "poder ser computados en paralelo" a una clase muy simple de problemas que excluye a muchísimos ejemplos que usan MapReduce para buenos resultados en la práctica (ver p.ej. [7]).

# UN MODELO TEÓRICO PARA MAPREDUCE

Teóricamente, podemos modelar todo algoritmo de MapReduce con dos funciones, *M* y *R*, por Map y Reduce. Tal como ocurre en la práctica, la función *M* toma como input una secuencia de pares llave-valor y retorna una secuencia de pares llave-valor; y la función *R* toma como input una llave junto a una secuencia de valores y genera nuevamente una secuencia de pares (llave,valor). Dado un problema arbitrario a ser resuelto con MapReduce, asumimos que el input a ese problema está dividido en pedazos $p_1,\ldots, p_n$. La función *M* (*Map*) toma un pedazo de input $p_i$ y lo transforma en una secuencia de pares *(llave,valor)* $M(p_i) = \{(k_1, v_1),\ldots,(k_m, v_m)\}$. Posteriormente se toma la unión de todos los pares (llave,valor) de todas las llamadas a *M*; junta todos los valores asociados a una misma llave, genera una copia de la función *R* por cada llave, y le entrega a *R* esta llave junto con todos sus valores asociados.

De esta forma, para funciones *M* y *R* dadas, el resultado de aplicar MapReduce sobre un input $(1, p_1),\ldots,(n, p_n)$ se define de la siguiente forma. Primero, el resultado de agrupar el output de todas las funciones *M* corresponde a

$$Map((1, p_1),\ldots,(n, p_n)) = \bigcup_{i=1}^{i=n} M(p_i)$$

Este conjunto corresponde a una secuencia de pares (llave,valor). Posteriormente, para cada llave *k* que sea parte de algún par en $Map((1,p_1),\ldots,(n,p_n))$, definimos el conjunto de los pares (llave,valor) asociados a *k* como $pares(k)=\{(k,v) \in Map((1,p_1),\ldots,(n,p_n))\}$. El resultado de MapReduce sobre $(1, p_1),\ldots,(n, p_n)$ se define como

$$MR((1, p_1),\ldots,(n, p_n)) = \bigcup_k R(pares(k)) \quad (1)$$

Es decir, el resultado de MapReduce es el resultado de aplicar *R* sobre cada una de las llaves, junto a sus valores asociados, que pertenecen a la unión de las secuencias llave-valor entregadas por cada llamada a *M* para cada par $(i, p_i)$ del input.

Por ahora no hemos hecho nada más que representar las funciones de MapReduce de forma matemática, pero tenemos que avanzar mucho más si queremos un modelo que cumpla nuestra meta. Para empezar, no hemos especificado ninguna condición sobre la forma en que dividimos nuestro input, por lo que, hasta ahora, podemos simular cualquier algoritmo *A* que computa cierto problema de forma serial (es decir, no en paralelo) de la siguiente forma: para cada input *p* para *A*, generamos una función *M* que reciba *p* y retorne (1, *p*), y definimos a *R* de forma que reciba (1, *p*) y ejecute *A* sobre *p*. En otras palabras, *M* no hace más que generar una llave arbitraria para *p*, y *R* es una copia de *A*. El resultado, obviamente, es el mismo que en el caso serial, aunque no nos estamos aprovechando en absoluto de la arquitectura paralela de MapReduce, porque siempre ejecutaremos una sola función *M* y una sola función *R*. Y no es realista pensar que el input va a venir bien formado en un solo pedazo; por el contrario, como el input es generalmente muy grande, lo más lógico sería pensar que está almacenado en un gran número de pedazos en algún sistema de almacenamiento distribuido de datos.

Para agregar esta restricción a nuestro modelo, fijamos con anterioridad el número de *mappers* (o llamadas a la función *M*) a usar. Para un número *n* arbitrario y una secuencia de *n* pares (llave,valor) $S = \{(1, p_1),\ldots,(n, p_n)\}$, hablaremos de la ecuación (*1*) como el resultado de aplicar MapReduce "usando *n* mappers" sobre la secuencia *S*. Si *n*=1, entonces claramente estamos hablando de un algoritmo que no es paralelo. Pero si asumimos que *n* es relativamente grande, volvemos a nuestra pregunta original, esta vez más refinada: ¿Qué problemas podemos implementar usando MapReduce con *n* mappers?

Con esto podemos modelar el hecho de que nuestro input esta bien distribuido, pero nuevamente podemos simular cualquier algoritmo *A*, incluso si asumimos que *n* es grande. Imaginemos que dividimos arbitrariamente el input *p* de *A* en una secuencia de pares (llave,valor) $S = \{(1, p_1),\ldots,(n, p_n)\}$. Definamos la siguiente función *M*: toma un par $(i, p_i)$ y entrega el par $(1, p_i)$. El resultado de aplicar *n mappers* sobre *S es* $\{(1, p_1),\ldots,(1,p_n)\}$. Ahora podemos definir *R* como la función que une nuevamente todos los pedazos y luego llama a *A*. Si bien esta vez comenzamos con un sistema de datos distribuidos, lo único que hace nuestro algoritmo MapReduce es juntar todos esos pedazos en uno y pasárselo todo a un *reducer*; por lo que nuevamente ejecutamos *A* en un solo nodo de nuestra arquitectura.

→

¿Cómo podemos entonces modelar todas las ejecuciones de MapReduce en las que aprovechamos, en cierta forma, la arquitectura paralela? Una alternativa es limitar la comunicación entre los *mappers* y los *reducers*. Para explicar esto, volvamos a nuestra algoritmo *A* que recibe input *p*, y supongamos que el tamaño de nuestro input es *t* (podemos pensar en *t* como el número de caracteres o de bits que tiene *p*). Si usamos *n mappers*, y distribuimos el input de forma equitativa sobre cada *mapper*, éstos van a recibir un input de tamaño *t/n*. Vamos a obligar a que el output de la función *M*, para cada llave distinta, sea siempre menor que *t/n*. Para ser más formales, digamos que el output de *M*, para cada llave distinta que genere, no puede ser mayor al logaritmo *log(t)* de *t*: sabemos que, para valores grandes de *t*, *t/n* siempre va a ser mayor a *log(t)*.

Hemos llegado a nuestro modelo definitivo. Decimos que un algoritmo *A* puede ser simulado con MapReduce usando *n mappers* si existen funciones *M* y *R*, tal que:

**(c.1)** Para todo input *p* de tamaño *t* y toda división de un input *p* para *A* en una secuencia $S = \{(1, p_1),\ldots,(n, p_n)\}$, el resultado de la ecuación (1) es equivalente al resultado de aplicar *A* a *p*.

**(c.2)** La suma del tamaño de los valores de $M(p_i)$ asociados a cada llave no puede superar *log(t)*.

Finalmente, como nos interesan los casos en los que MapReduce consigue computar más rápidamente el algoritmo *A* que el resultado de ejecutar *A* en forma serial, sin paralelismo, agregaremos una tercera condición:

**(c.3)** El tiempo que demora el mejor algoritmo en serie para computar *A* es siempre mayor o igual al tiempo total que demora computar *A* usando MapReduce, es decir, el máximo entre el tiempo que demora cada *M* sumado al máximo del tiempo que demora cada *R*.

Podemos usar este modelo de la siguiente forma: si podemos simular un problema de acuerdo a estas características, entonces MapReduce es un buen candidato para resolver este problema. Por ejemplo, es evidente que los problemas de contar ocurrencias de palabras y de join podrán ser simulados con nuestro modelo, siempre y cuando el input nos asegure una buena distribución. Es razonable pensar que el problema de conectividad no puede ser resuelto de esta forma, pero, ¿podemos demostrar formalmente este hecho para éste u otro problema? Más importante aún, ¿podemos afirmar que los problemas que no podemos simular con este modelo no se dan bien en MapReduce? La respuesta a estas preguntas tendría una implicancia inmediata en la práctica, al momento de pensar si instalar o no MapReduce para resolver un problema.

Otra dirección importante es la aplicación de rondas sucesivas de MapReduce: hemos señalado que el resultado de MapReduce puede ser la entrada de otro algoritmo de MapReduce distinto. Entonces es natural preguntarnos: ¿existen problemas que no puedan ser simulados usando una ronda de MapReduce, pero que sí lo sean si usamos dos rondas? Y, ¿qué hay de un número mayor de rondas? ¿Es cierto que el problema de conectividad puede ser simulado si usamos un número de rondas igual al tamaño del grafo original, o igual al logaritmo del tamaño del grafo original?

Finalmente, existen otras arquitecturas paralelas bastante simples que se usan hoy en la práctica, siendo quizás los ejemplos más importantes GraphLab o Pregel. Es necesario trabajar en una definición formal similar a lo hecho con MapReduce, para determinar el alcance de éstas otras arquitecturas. Esta tarea quizá podría llevarnos a descubrir una teoría mucho más general de computación paralela. ■

## BIBLIOGRAFÍA

[1] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[2] Patterson, D. A. (2008). The data center is the computer. Communications of the ACM, 51(1), 105-105.

[3]Prn news blog, http://www.prnewswire.com/news-releases/altiors-altrastar---hadoop-storage-accelerator-and-optimizer-now-certified-on-cdh4-clouderas-distribution-including-apache-hadoop-version-4-183906141.html

[4] Stampede Visual Workshop, https://www.cac.cornell.edu/Stampede/default.aspx

[5] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009, June). A comparison of approaches to large-scale data analysis. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (pp. 165-178). ACM.

[6] Rajaraman, A., & Ullman, J. D. (2012). Mining of massive datasets. Cambridge University Press.

[7] Lynch, N. A. (1996). Distributed algorithms. Morgan Kaufmann.