Cada vez más rápido

DOI: 10.29236/sistemas.n161a7

Acelerando algoritmos con técnicas de paralelismo.

Resumen

El diseño de algoritmos es fundamental para la construcción de artefactos de software que son correctos y eficientes. La importancia de los algoritmos se ha visto resaltada en los últimos años, donde debido a los requerimientos de procesamiento de altos volúmenes de datos en dominios como la inteligencia artificial o el procesamiento en la nube con arquitecturas de BigData. Ya no es suficiente diseñar algoritmos con tiempos de complejidad teórica óptimos. Además, estos algoritmos deben optimizar al máximo el tiempo de ejecución de los procesos. Para responder a esta necesidad, la programación paralela se ha enfocado en el desarrollo de algoritmos que pueden ejecutar múltiples tareas de forma simultánea, cada una en una unidad de procesamiento distinta. En este trabajo, presentamos la forma de mejorar el tiempo de ejecución de algoritmos óptimos aplicando técnicas de paralelismo a los algoritmos para aprovechar al máximo las capacidades de multi-procesamiento de las máquinas modernas. Para hacer concreto este proceso, utilizamos el método de descomposición de raíz cuadrada sobre grandes volúmenes de datos. Los resultados obtenidos de nuestro experimento muestran una mejora de hasta 1.9x sobre la versión secuencial del algoritmo, confirmando los resultados teóricos del diseño del algoritmo.

Palabras claves

Algoritmos, Paralelismo, Consultas en Rango, Descomposición de raíz cuadrada

1. Introducción

El diseño de algoritmos es fundamental para la construcción de artefactos de software que son correctos y eficientes. Los sistemas de software modernos (*e.g.,* sistemas de BigData o inteligencia artificial) requieren una alta capacidad de procesamiento de datos y operaciones complejas. Para poder construir estos sistemas es de suma importancia que los algoritmos utilizados sean diseñados de forma óptima (*i.e.,* en complejidad de tiempo y espacio), para si poder satisfacer las necesidades del sistema. La ley de Moore ya no es cierta. La capacidad de procesamiento de los computadores ya no se duplica cada dos años (Patterson & Hennessy, 2009). Por lo tanto, no podemos depender de las capacidades del software para lograr un mayor rendimiento de los sistemas en tiempo y espacio. En consecuencia, el uso de algoritmos óptimos ya no es suficiente para satisfacer los requerimientos de procesamiento de altos volúmenes de datos. Para alcanzar la eficiencia deseada, los desarrolladores de software deben diseñar nuevos algoritmos que puedan sobrepasar la barrera de memoria y energía existente actualmente (Wulf & McKee, 1995) (Knapp & H., 2005).

Una de las técnicas de diseño y aceleración de algoritmos que ha tomado mucha fuerza en los últimos 20 años está basada en los modelos de programación en paralelo (Raube & Rünger, 2013). La programación paralela es una técnica que se enfoca en diseñar algoritmos que permiten la ejecución de diferentes tareas simultáneamente. De esta manera, es posible acelerar la ejecución de algoritmos, o aumentar el procesamiento de datos (dentro del mismo tiempo de ejecución) sin requerir mayor capacidad de procesamiento, pero utilizando más unidades de procesamiento.

En este artículo exploramos la paralelización de algoritmos óptimos, ejemplificada por medio de un ejemplo concreto, la descomposición de raíz cuadrada para resolver problemas de consultas en rango. Nuestro diseño de algoritmos paralelos demuestra un beneficio de aceleración de hasta 1.97x para nuestro ejemplo concreto. Sin embargo, enfatizamos que estas técnicas son aplicables a diferentes algoritmos y los beneficios serán proporcionales dentro de otros dominios de aplicación mejorando así el tiempo de ejecución de los sistemas de software.

2. Descomposición de raíz cuadrada

El método de descomposición de raíz cuadrada (DRC), es un algoritmo que busca optimizar la complejidad temporal de las consultas y actualizaciones dentro de un conjunto de datos. Este tipo de proble-

mas se conocen como los problemas de consultas en rango (Yao, 1982). La idea detrás de este método es organizar los datos dentro de un arreglo, y descomponer el arreglo en bloques $\sqrt{n}$ de tamaño $\sqrt{n}$ donde n corresponde al tamaño total del conjunto de datos.

Como un ejemplo de DRC, consideremos un arreglo arr=[1,2,3,4,5, 6,7] de nueve elementos. Vamos a ejecutar una serie de consultas y actualizaciones sobre arr basados en una operación de agregación para los elementos en un rango $[l,r]$, donde $0 \leq l,r < 9$. Para nuestro ejemplo, utilizaremos la suma (+) como función de agregación, pero otras operaciones podrían ser utilizadas.

DRC divide el arreglo arr en 3 secciones de tamaño 3 cada una, teniendo en cuenta que el tamaño n del arreglo es un cuadrado perfecto. Esta división se ve en la Figura 1. Como se muestra en la figura, el primer paso del algoritmo es utilizar la función de agregación entre todos los elementos de cada sección. Por ejemplo, la agregación del segundo bloque suma 11.

Para responder a las consultas en un rango $l \leq r$, existen dos casos dependiendo de los valores que tomen l y r. En el primer caso, los límites l y r corresponden exactamente a los límites de alguno de los bloques del arreglo. En ese caso, se puede resolver la consulta ejecutando la agregación para cada uno de los bloques en el rango. Por ejemplo, para resolver la consulta query(3,8) solo es necesario sumar los últimos dos bloques de arr de tal forma que query(3,8) = 11 + 15 = 26. En el segundo caso, si los valores de l y r no corresponden con los límites de algún bloque, por ejemplo en la consulta query(1,7), debemos agregar los valores de los bloques totalmente contenidos dentro del rango, y luego debemos agregar los elementos que se encuentran fuera de esos bloques y en el rango. En nuestro ejemplo, esta consulta requiere agregar el valor del segundo bloque, 11, y operar sobre los elementos restantes de los otros bloques, el resultado se muestra en la siguiente Ecuación.

$$\text{arr}[1] + \text{arr}[2] + 11 + \text{arr}[6] + \text{arr}[7] = 26$$

Figura 1. Arreglo de datos a procesar por medio de consultas y actualizaciones

El Fragmento 1 de código muestra el segundo caso de las consultas para el problema de consultas en rango. En las Líneas 10-14 podemos ver el cálculo de la agregación (marcado como la función fun dentro del código) de los elementos de los bloques, y la agregación de los elementos fuera de estos bloques en el arreglo original.

Utilizando el método de DRC es posible mejorar el algoritmo para solucionar el problema de consultas en rango. La complejidad original del algoritmo es $0(n)$, utilizando DRC, y teniendo en cuenta los dos posibles casos de consultas, es posible notar que el máximo número de operaciones a ejecutar por consulta es $\sqrt{n}$ por lo tanto obteniendo una complejidad total para el algoritmo de $o(\sqrt{n})$.

3. Algoritmo paralelo para el método de raíz cuadrada

Tomando como base el método de DRC para los algoritmos de consultas de rango, podemos observar que la división del conjunto de da-

```
1  def query(array, chunks, l, r, fun, root, acc) do
2    tasks=[]
3    prev_root=root-1
4    chunk1=trunc(l/root)
5    chunk2= trunc(r/root)
6    case {rem(l, root), rem(r, root)} do
7      #...
8      {_,_} -> if chunk2-chunk1>1 do
9        last = root*chunk1+root-1
10       start2 = root*chunk2
11       tasks= tasks ++[operate(array, l, last, acc, fun)]
12                   ++[operate(chunks, chunk1+1, chunk2-1, acc, fun)]
13                   ++[operate(array, start2, r, acc, fun)]
14       IO.puts Enum.reduce(tasks, acc, fn x, acc-> fun.(acc, x) end)
15       else
16       if (chunk2-chunk1)==0 do
17         IO.puts operate(array, l, r, acc, fun)
18       else
19         last = root*chunk1+root-1
20         start2 = root*chunk2
21         tasks= tasks ++[operate(array, l, last, acc, fun)]
22                      ++[operation(array, start2, r, acc, fun)]
23         IO.puts Enum.reduce(tasks, acc, fn x, acc-> fun.(acc, x) end)
24       end
25     end
26   end
27 end

29 defp operate(array, l, r, acc, fun) do
30   Enum.slice(array, l..r)|> Enum.reduce(acc, fun)
31 end
```

Fragmento 1.Consulta secuencial para DRC en Elixir

tos en bloques de tamaño $\sqrt{n}$ expone una propiedad clave para la ejecución de algoritmos en paralelo, el paralelismo de tareas (Subhlok, Stichnoth, O'hallaron, & Gross, 1993). Es decir, al dividir el conjunto de datos en $\sqrt{n}$ bloques independientes, tenemos la posibilidad de procesar cada uno de los bloques en paralelo. En nuestra implementación paralela de DRC, utilizamos esta propiedad en dos momentos distintos, que permiten acelerar la ejecución de las consultas.

Primero, la construcción inicial del arreglo, requiere dividirlo en $\sqrt{n}$ bloques para luego aplicar la función de agregación para cada uno de los bloques. En nuestra implementación, estas operaciones se ejecutan en procesos (*i.e.,* hilos de ejecución o threads) independientes de forma simultanea, como se muestra en la Figura 2. para el arreglo de ejemplo presentado en la Sección 2. Paralelizando el pre procesamiento inicial del arreglo, es posible disminuir la complejidad de las operaciones ejecutadas secuencialmente (con complejidad $0(n)$, a el procesamiento de un solo bloque, $0(\sqrt{n})$.

El segundo momento en el que utilizamos el paralelismo de tareas es en el cálculo de las consultas. Como presentamos en la Sección 2, y como se muestra en el Fragmento 2 de código, el procesamiento de las consultas requiere, en el peor de los casos, la agregación de los bloques totalmente contenidos entre el rango $[l,r]$, la agregación de los elementos entre el límite izquierdo de la consulta y el primer bloque totalmente contenido en el rango y la agregación de los elementos entre el último bloque totalmente contenido en el rango y el límite derecho de la consulta. Este ejemplo se muestra entre las Líneas 10 y 12, donde los nuevos procesos para hacer el cálculo se lanzan por medio del llamado a Task.async() y el método operate ejecuta la agregación sobre los elementos del arreglo. Cada uno de estos cálculos requiere un máximo de $\sqrt{n}$ operaciones y se ejecutan de forma simultánea dentro de cada un de los procesos independien-

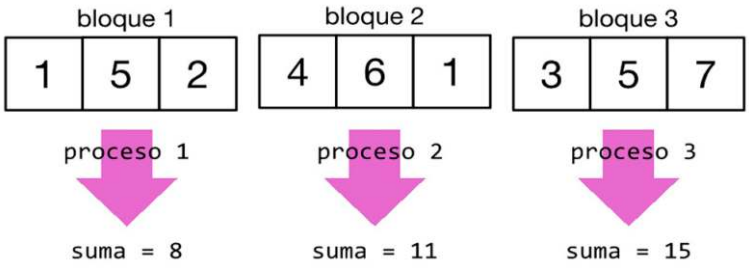

Figura 2. Paralelismo en las tareas. Tres procesos ejecutando la función de agregación simultáneamente

```
def query(array, chunks, l, r, fun, src, root, acc) do
  tasks=[]
  chunk1=trunc(l/root)
  chunk2= trunc(r/root)
  case {rem(l, root), rem(r, root)} do
    #...
    {_,_} -> if chunk2-chunk1>1 do
      last = root*chunk1+root-1
      start2 = root*chunk2
      tasks=tasks++[Task.async(operate(array, l, last, acc, fun)]
          ++[Task.async(operate(chunks, chunk1+1, chunk2-1, acc, fun)]
          ++[Task.async(operate(array, start2, r, acc, fun)]
       spawn(src, tasks, acc, fun)
       {chunks, array}
     else
       if (chunk2-chunk1)==0 do
         spawn_task(__MODULE__, :send, src, [{:answ, operate(array,
l..r, acc, fun)}])
         {chunks, array}
       else
         last = root*chunk1+root-1
         start2 = root*chunk2
         tasks=tasks++[Task.async(operate(array, 0, last, acc, fun)]
                 ++[Task.async(operate(array, start2, r, acc, fun)]
         spawn(src, tasks, acc, fun)
         {chunks, array}
       end
     end
   end
end

defp spawn(src, array, acc, fun) do
  spawn_task(__MODULE__, :send, src, [{:answ, Enum.reduce(array, acc,
fn x, acc-> fun.(acc, Task.await(x)) end)}]
end
```

Fragmento 2. Código paralelo para hacer consultas

temente. De esta forma, en teoría, podemos reducir el tiempo de ejecución hasta tres veces dado que se ejecutan las tres agregaciones en un mismo momento de operación.

Cabe notar que una de las condiciones para poder realizar la paralelización de esta forma, es que la función de agregación sea asociativa. De esta forma es posible operar cualquier de los resultados pre calculados, sin importar el orden en el que sus procesos respectivos finalicen la ejecución.

4. Resultados

Para mostrar el beneficio del uso de la implementación paralela del algoritmo de DRC, diseñamos un experimento que consiste en realizar consultas sobre arreglos de distintos tamaños, demostrando la escalabilidad en tiempo de ejecución de nuestra implementación. Para el experimento exploramos arreglos de enteros de tamaños entre [1000,

1000000] elementos. Para cada uno de los arreglos realizamos 10 consultas del segundo tipo, donde los límites de las consultas no coinciden con los límites de los bloques.

Más aún, en nuestro experimento nos enfocamos en el peor caso de ejecución, en el cual los límites de la consulta hacen parte del primer y último bloques. Finalmente, nuestro experimento utiliza una función polinómica como función de agregación (fun = $5x + 2y - 1$), sin embargo, cualquier otra función asociativa podría ser utilizada (*e.g.*, min, max, +, *).

El experimento se ejecutó en una máquina con un procesador Intel Core i& de 2.8GHz con cuatro núcleos y 16GB de RAM, corriendo la versión 10.14.6 de macOS. El ambiente de ejecución esta compuesto por dos instancias de la máquina virtual de BEAM con la versión de Erlang/OTP 24 y la versión 1.12.1 de Elixir.

Haciendo un análisis de la ejecución del experimento, vemos que las consultas requieren la mayor cantidad de cálculos a realizar. Primero es necesario aplicar la función de agregación sobre los valores pre calculados entre los bloques 2 y $\sqrt{n} - 1$, para un total de $\sqrt{n} - 2$ operaciones. Luego es necesario calcular los valores para los dos bloques restantes, requiriendo $\sqrt{n} - 1$, operaciones en cada caso (si ambos bloques son de tamaño máximo $\sqrt{n}$). Luego el total de operaciones requeridas en el caso secuencial es $3\sqrt{n}$ (*i.e.* complejidad $0(\sqrt{n})$)como se discutió en la Sección 2.

En el caso de la ejecución paralela, cada una de las tres operaciones discutidas anteriormente se puede ejecutar dentro de su propio proceso. Por lo tanto el total de operaciones observable correspondería a $\sqrt{n}$ (con una complejidad del algoritmo de $0(\sqrt{n})$); y el tiempo total correspondería a la operación más lenta ejecutando en cada uno de los procesos.

La Figura 3. muestra el resultado promedio de la ejecución de las 10 consultas (eliminando los valores atípicos causados por el warmup del sistema) de nuestro experimento. Los resultados obtenidos confirman los resultados analíticos descritos anteriormente. Como es posible ver de la figura, la ejecución en paralelo presenta un mejor rendimiento (en tiempo de ejecución) que el caso en que la ejecución es secuencial. La aceleración promedio en la ejecución de cada con-sulta, siguiendo la ley de Amdhal (Rodgers, 1985) $\overline{Speedup(n)} = \frac{T(1)}{T(n)}, n = 4,$ es de $1.09x$, con una aceleración máxima de $1.97x$.

Como podemos ver, la aceleración obtenida experimentalmente es menor que la aceleración teórica. La razón del comportamiento subóptimo de los algoritmos paralelos, es el costo de generar y sincronizar los procesos.

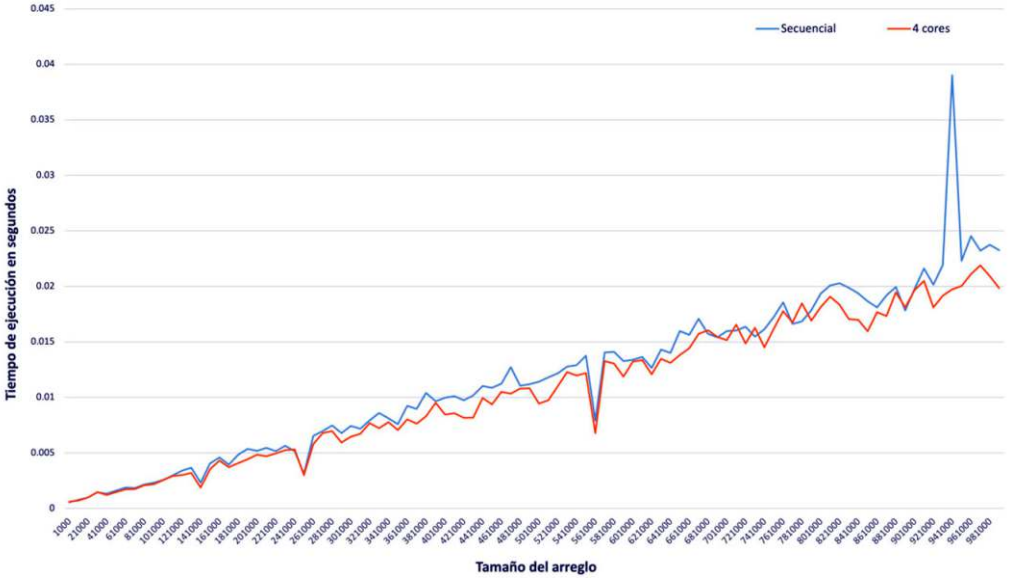

Figura 3. Comparación de tiempo de ejecución entre la versión secuencial y paralela del algoritmo de DRC

5. Conclusión

Actualmente ya no es posible depender del hardware para escalar las operaciones ejecutadas por un sistema de software. Es imperativo desarrollar las técnicas que nos permitan mejorar el rendimiento de los sistemas de software modernos. Este trabajo muestra técnicas de computación en paralelo para mejorar la complejidad espacial y temporal de algoritmos óptimos. El objetivo de la aplicación y el diseño de nuevos algoritmos paralelos es incrementar el rendimiento, en tiempo de ejecución de los sistemas de software, para así poder satisfacer los requerimientos de procesamiento (volumen de datos a procesar y restricciones temporales) que imponen los sistemas de software moderno. La necesidad de aplicar estas técnicas para diseñar nuevos algoritmos responde lograr sobrepasar los límites de memoria y energía que presentan los procesadores actuales.

Los resultados de nuestro trabajo muestran inclusive que, partiendo de algoritmos óptimos y utilizando técnicas de programación en paralelo es posible acelerar el tiempo de ejecución de dichos algoritmos, en especial cuando se trata del procesamiento de grandes volúmenes de datos, como por ejemplo en el caso de aplicaciones de BigData o inteligencia artificial. Como conclusión de nuestro trabajo resaltamos el importante rol que juegan los algoritmos para satisfacer los requerimientos de los sistemas de software.

Referencias

Raube, T., & Rünger, G. (2013). Parallel Programming. New York: Springer Heildelberg.

Yao, A. C. (1982). Space-time tradeoff for answering range queries (Extended Abstract). Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (págs. 128--136). San Francisco, California, USA: ACM.

Patterson, D. A., & Hennessy, J. L. (2009). Computer Organization and Design. Morgan Kaufmann.

Knapp, S. N., & H., J. W. (2005). SE2 when processors hit the power wall (or "when the CPU hits the fan"). IEEE International Digest of Technical Papers. Solid-State Circuits Conference (págs. 16-17). San Francisco: IEEE.

Wulf, W. A., & McKee, S. A. (1995). Hitting the Memory Wall: Implications of the Obvious. SIGARCH Computing Architecture News, 20-24.

Subhlok, J., Stichnoth, J. M., O'hallaron, D. R., & Gross, T. (1993). Exploiting task and data parallelism on a multicomputer. ACM SIGPLAN symposium on Principles and practice of parallel programming (págs. 13-22). ACM.

Rodgers, D. P. (1985). Improvements in multiprocessor system design. ACM SIGARCH Computer Architecture News, 225-231.

Juan Felipe Ramos. *Juan Felipe Ramos es Ingeniero de Sistemas y Computación de la Universidad de los Andes. En la actualidad trabaja para el start-up Preki.*

Juan Sebastián Gómez. *Juan Sebastián Gómez es Ingeniero de Sistemas y Computación de la Universidad de los Andes. En la actualidad trabaja para el start-up Cleverlynk.*

Nicolás Cardozo. *Nicolás Cardozo es Profesor Asistente de la Universidad de los Andes desde el 2016. Especializado en el desarrollo de lenguajes de programación. Doctor en ingeniería de la Université Catholique de Louvain, Bélgica 2013 y doctor en ciencias de la Vrije Universiteit Brussel, Bélgica 2013; Maestro en ciencias de la Vrije Universiteit Brussel, Bélgica 2009; Ingeniero de sistemas y computación y Matemático de la Universidad de los Andes, 2008.*