Trabajo Tema 2:

Julia a Language programing

Jose Luis Martínez Ortiz

25 de abril de 2017

Resumen

La programación científica sobretodo necesita un gran rendimiento y para ello la vía actual es la programación distribuida. Esta faceta del rendimiento y la paralelización y programación distribuida se ha potenciado enormemente en este lenguaje de programación que vamos a analiza y este es el motivo por el que he elegido hacer el trabajo sobre Julia. Nos centraremos sobretodo en explicar los mecanismos que nos ofrece y cómo gestiona la programación paralela de forma interna y transparente al programador. Ofreciendo las ventajas de un lenguaje diseñado y desarrollado en la actualidad y pensando en los equipos reales que contamos hoy en día para el computo.

1. Introducción

Julia es un lenguaje de programación liberado en 2012 por Jeff Bezanson, Stefan Karpinski, Viral Shah y Alan Edelman. Su creación fue motivada porque el equipo de trabajo utilizaba distintos lenguajes de programación científica o de propósito general como pyhton o mathlab. Esto se debía a que cada lenguaje está orientado a un ámbito: computación científica, aprendizaje automático, minería de datos, álgebra lineal, etc, y ofrece unas ventajas especificas en cada campo. La solución para poder aprovechar todas las ventajas de cada lenguaje fue crear un lenguaje que esté a más nivel de estos y que permita integrar otros lenguajes y librerías. De esta forma Julia se nutre del conocimiento y trabajo desarrollado hasta el momento en otros lenguajes. Julia tiene la gran ventaja que soporta de forma nativa código en C,R, Ruby, mathlab, python, etc. Otro punto a favor de Julia es que se diseño para que soportara también la programación distribuida de forma nativa y auto-gestionada.

Sus principales propiedades, tal como expresan los propios desarrolladores, son: que es un lenguaje de alto nivel dinámico y cuenta con un compilador "sofisticado" y muy potente. Que permite la compilación en tiempo de ejecución, por lo que podemos definir que es un lenguaje interpretado. Además cuenta con un dispacher múltiple que permite y fomenta la programación paralela. Permite la ejecución de hebras y co-rutinas al estilo java[2]. La declaración de tipos es de forma anónima como en Ruby, solo se tienen en cuenta a la hora de ejecutar, y permite la declaración de tipos por parte de los usuarios ,sus creadores comentan que son tan rápidos como los nativos del lenguaje.

2. Julia y la Programación Distribuida

Julia incorpora varios mecanismos para la programación paralela y distribuida de una forma cómoda y fácil. Para ello aprovecha que prácticamente todos los equipos actuales cuentan con multiprocesador ya sea en la misma máquina o juntando varias máquinas físicas formando clusters de procesamiento. La forma de tratar el paralelismo que tiene Julia es de forma "unilateral" entre procesos, es decir, en una comunicación entre dos procesos el programador solo tiene que tener en cuenta uno de ellos, el que realiza la llamada. De esta forma se alivia la tarea entre la comunicación y la programación. Se asemeja mucho a las llamadas entre métodos.

Este lenguaje de programación se fundamenta en dos primitivas para la computación paralela: referencias remotas y llamadas remotas. Las referencias remotas son objetos que se pueden referenciar desde un proceso distinto al que fue instanciado dicho objeto. Esté mecanismo, soportado de forma nativa, permite una gran potencia de programación paralela y distribuida ya que un proceso en una máquina puede instanciar y utilizar objetos declarados y definidos en otra máquina que puede estar en cualquier parte del mundo. La referencias remotas tiene dos versiones, la "future" y la "RemoteChannel". La versión "future" permite obtener el resultado en un momento concreto, es decir, el objeto solicitante decide cuando recoger el objeto solicitado mediante llamadas especiales de Julia. Un ejemplo que viene en su documentación y es muy explicativo es el siguiente:

```
> ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2,1,3,Nullable{Any}())

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2,1,6,Nullable{Any}())

julia> fetch(s)
2x2 Array{Float64,2}:
 1.60401  1.50111
 1.17457  1.15741
```

En el ejemplo vemos como se ejecuta una llamada remota al método “rand” y los demás elementos son los parámetros con los que se les llama. Para Julia no hace falta indicarle nada pero internamente ha parelilizado la ejecución del método “rand” en distintas unidades de procesamiento disponibles. Y cuando deseemos utilizar el resultado de la operación basta con llamar al método “fetch” para indicar que estamos listos para recoger el resultado. Gracias a este mecanismo nos queda un código más elegante y fácil de aprender.

Para las llamadas remotas o remotecall vemos otro ejemplo, procedente de la documentación oficial.

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.10824216411304866
```

En este caso se ha realizado una llamada remota al método getindex y hemos obtenido el resultado. Al igual que pasa con los objetos remotos la ejecución de este método ha sido paralela y ejecutada en cualquiera de las máquinas del sistema.

Podemos decir a muy groso modo que Julia tiene un sistema de llamadas sincronas y asincronas en sus formas de comunicación remota. Ofreciendo una gran variedad de alternativas y posibilidades para cada trabajo específico. De esta forma logramos el objetivo de que si por ejemplo necesitamos un sistema software distribuido, podemos tener con Julia programado todo el sistema. Los clusters de procesamiento al estar en Julia ofrecen un rendimiento y paralelización a un alto nivel y abstrayendo al programador de prácticamente toda la tarea de configurar la paralelización del software. En los software de control y middleware podemos utilizar servicios, librerías, software anterior o de otro proyecto e incluso drivers que a existan y añadirlos al nuevo sistema software sin necesidad de utilizar librerías externas ni un proceso de adaptacióndel código puesto que Julia permite la ejecución nativa de otros lenguajes como ya se comento anteriormente. Consiguiendo así una integración rápida y eficiente de sistemas antiguos construidos en varios lenguajes.

Otra ventaja que posee es la dualidad para mandar datos a los distintos equipos del sistema, ya que no solo permite el envío de datos resultado, si no operaciones explicitas. Para ilustrar estas dos formas utilizaré un ejemplo de la documentación que es muy sencillo y aclarativo.

```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
```

En el primer método se calcula una matriz cuadrada de números aleatorios y luego con la macro “@spawn” se manda la matriz cuadrada y la operación asociada a esta, para que se ejecute en el objeto a donde ha sido enviada. En cambio en la segunda forma de hacer el envío de datos se manda la instrucción de construir la matriz de números aleatorios y luego hacerle el cuadrado a dicha matriz. De esta forma tan sencilla nos provee de una herramienta muy potente con la que por ejemplo podemos tener en un lugar el equipo de desarrollo y científico indicando que en el custers de servidores de computación se ejecuten operaciones que están escribiendo en el momento en el entorno de trabajo de Julia, como se puede hacer con Python en un Notebook. Y en otro lugar el equipo especialista en los cluster encargandose de que todo esté correctamente. Para este ejemplo no sería necesario la utilización de conexiones remotas como ssh o cualquier software para trabajar de forma remota, si no que es el propio lenguaje de programación el que te abstrae de esta tarea.

3. Conclusiones

Julia es un lenguaje que sorprende por su gran versatilidad y los resultados que ofrece para un lenguaje de tan alto nivel. En los resultados de rendimiento en el cálculo numérico, realizado en su web comparándolo con otros lenguajes similares o de uso extendido, obtiene valores cercanos a los conseguidos con C. Se nota que es fruto de un equipo de científicos e ingenieros informáticos multidisciplinar donde han aunado esfuerzos para conseguir un lenguaje común con todas, o casi todas, las ventajas de los lenguajes en los que se han basado. Teniendo un acierto que a mi parecer ha sido clave, la retrocompatibilidad con los principales lenguajes utilizados elimina eliminando la barrera de tener que empezar de cero si quieres utilizar Julia. De esta forma puedes migrar tu sistema software poco a poco sin que deje de funcionar y totalmente transparente para los usuarios finales, ya que con su gestor de paquetes moderno permite la carga de estos sin que el usuario tenga que instalarlos o descargarlos y situarlos en un lugar visible para el software.

Ahora lo que falta por ver es si la comunidad está dispuesta a darle una oportunidad o seguirá utilizando python o Matlab por ejemplo. Es un lenguaje relativamente joven, muy joven con apenas 5 años de vida y que su equipo de desarrollo está constantemente actualizandolo y sacando nuevas versiones más estables y con más características. Las opiniones en la redes especializadas del sector matemático e informático son escépticas por el momento pero con un sabor general de curiosidad e intriga para ver si de verdad es capaz de cumplir lo prometido. En los próximos años veremos si Julia se hace con un hueco en el sector de los lenguajes.

4. Bibliografía

[1] *Julia documentación online* - https://docs.julialang.org/en/stable/manual/introduction/

[2] *Massachusetts Institute of Technology* - http://web.mit.edu/1.124/LectureNotes/Multithreading.htm