Optimización de JuliaReach

Bruno Garate
Facultad de Ingeniería UdelaR
Montevideo, Uruguay
bruno.garate@fing.edu.uy

***Resumen*—El garantizar la seguridad de la evolución de sistemas en el mundo real a partir de modelos con miles de variables representa un reto computacional. JuliaReach ofrece una solución competitiva para el estudio de las seguridad de ecuaciones diferenciales ordinarias afines con entradas no deterministas [8]. En este proyecto investigamos la paralelización del algoritmo desarrollado por Forets et al. para mejorar el desempeño.**

***Index Terms*—reachability, performance, julia, dynamic systems, high performance computing, parallel computing**

I. Introducción

Los sistemas dinámicos son sistemas dónde una función describe la evolución del sistema en el tiempo. Las partículas en un flujo de aire, los voltajes en un circuito eléctrico o el sistema mecánico de una lectora de discos pueden ser tratados como sistemas dinámicos.

El estudio de la *reachability* o alcanzabilidad de estos sistemas es la identificación de los estados que se pueden alcanzar partiendo de un número posible de estados iniciales. Esto nos permite definir si un sistema evolucionará fuera de los parámetros seguros hacia regiones de *estados malos*. Estos estados pueden ser niveles de fatiga estructural en el cemento, voltajes elevados para lineas de transmisión o posiciones de trabajo incorrectas para un sistema mecánico.

El problema de la alcanzabilidad es indecidible en el caso general y las soluciones implican muchas veces métodos numéricos para la simulación de su evolución [12]. Para sistemas reales el número de variables involucradas es fácilmente del orden de las decenas de miles por lo que la computación de implica una concesión entre desempeño y precisión.

La solución presentada en [8] se encuentra desarrollada en el lenguaje de programación Julia y representa una mejora en tiempo de ejecución frente a algoritmos existentes. Además, para sistemas descritos por matrices dispersas con ordenes por encima de las decenas de miles, JuliaReach es el único algoritmo que logra presentar una solución.

A continuación se introduce el lenguaje de programación Julia y sus primitivas para el manejo de la computación paralela, se expone brevemente la solución propuesta por los autores de JuliaReach y se presenta el trabajo realizado, los resultados obtenidos y las conclusiones alcanzadas. Se finaliza con una breve sugerencia de trabajo futuro.

II. Julia

Julia es una lenguaje de programación de alto nivel diseñado para computación numérica. Es dinámicamente tipado, soporta *multiple dispatch*, cuenta con *garbage collection* y es multi paradigma: funcional, *multiple dispatch*, procedural, metaprogramación multi nivel. Además, el ecosistema de Julia incorpora una extensa librería de álgebra lineal, procesamiento de señales, manejo de strings, entre otros.

Julia se diseñó para ser altamente performante, comparable con lenguajes estáticamente tipados como C [3].

II-A. Computación paralela en Julia

Julia soporta paralelismo mediante hilos livianos (*threads*) así como mediante procesos. Al momento de la redacción, la librería de *threading* se encuentra en etapa experimental. En este proyecto se empleó la librería de computación paralela distribuida basada en procesos.

La computación paralela en Julia se construye sobre dos primitivas: referencias remotas (*remote references*) y llamadas remotas (*remote calls*). Las referencias remotas son objetos que permiten, como proxies, referirse a objetos en otros o el mismo proceso. Las llamadas remotas, de forma similar, permiten invocar métodos en otros u el mismo proceso.

A su vez, las referencias remotas se encuentran en forma de futuros (*futures*) o canales remotos (*remote channels*). El primero representa un valor futuro como resultado de una operación asíncrona mientras que los canales remotos operan como un buffer para la comunicación entre procesos donde se colocan y retiran datos.

Julia maneja el concepto de procesos trabajadores (*workers*). Además del proceso interactivo que se inicia por defecto, es posible ejecutar otros procesos trabajadores. Cuando se solicita la ejecución de una tarea en Julia, con el método *remotecall* o la macro *@spawn* la tarea es asignada a uno de los procesos trabajadores y se retorna un futuro representando el resultado futuro de la tarea. Para obtener el valor de este futuro se puede utilizar el método *fetch* que bloquea hasta que el resultado se encuentre disponible o lo devuelve inmediatamente en caso de ya haber sido generado.

Sobre estas primitivas se construyen el restos de los elementos de computación paralela de Julia. Por ejemplo, la macro *@parallel* prefijando un bloque *for* crea una clausura sobre el bloque y distribuye las iteraciones entre los trabajadores.

```
numerosAlAzar =
        @parallel (vcat) for i = 1:10000
            rand(1:6)
        end
```

Listing 1: Ejemplo de la macro @parallel

En el ejemplo 1, se distribuye la tarea de simular diez mil tiradas al azar de de dados y luego concatenarlos en un arreglo utilizando la función *vcat*.

Es importante notar que *@parallel* cambia totalmente la semántica del *for*. En ese sentido, se parece al modelo Map-Reduce, donde el for genera n tareas y la función reductora, en el ejemplo *vcat*, reduce los resultados cuando se encuentran disponibles. Además, al generarse los procesos debe considerarse que se realiza una clausura y una copia de las variables en la memoria de cada uno de los procesos que ejecutan la tarea. Esto es importante por dos motivos: debe cuidarse de minimizar la cantidad de memoria a transferirse y todas las modificaciones locales de las variables en un proceso no impactarán el estado de las mismas variables en otros procesos.

```
tiradasDeDados = zeros(100000)
@parallel for i = 1:100000
    tiradasDeDados[i] = rand(1:6)
end
```

Listing 2: Posible mal uso de @parallel

En el ejemplo 2 cada proceso trabajará sobre una copia del arreglo *tiradasDeDados* por lo que el resultado probablemente no sea el deseado. En cambio, se puede utilizar *SharedArray, SharedVector o SharedMatrix*, como se ve en el ejemplo 3 para utilizar arreglos que operan como un arreglo compartido entre los procesos.

```
tiradasDeDados =
        SharedArray{Float64}(100000)
@parallel for i = 1:100000
    tiradasDeDados[i] = rand(1:6)
end
```

Listing 3: Uso de SharedArray

Respecto a la copia de memoria entre procesos, debe considerarse definir variables locales dentro de la clausura frente fuera de la clausura para evitar que se copiadas entre procesos al ser capturadas. Ambos enfoques se pueden ver en los ejemplos 4 y 5.

```
# Copia a a todos los procesos
a = zeros(100000)
@parallel for i = 1:100000
    for j = j:100000
        a[j] = j*i;
    end
end
```

Listing 4: Ejemplo de variable capturada por la clausura con copia de arreglo entre procesos

III. JuliaReach

El código fuente de JuliaReach se encuentra disponible en [4] en forma una librería desarrollada en Julia, acompañado por varias librerías auxiliares como son LazySets y SX. JuliaReach se enfoca en la aproximación de los estados alcanzables

```
@parallel for i = 1:100000
    # a se inicializa en cada proceso
    a = zeros(100000)
    for j = j:100000
        a[j] = j*i;
    end
end
```

Listing 5: Ejemplo de variable no capturada por la clausura sin copia de arreglo entre procesos

comprometiendo cierta precisión por performance, alcanzando el estudio de casos densos con más de 10 000 variables, dos órdenes de magnitud por encima de los enfoques actuales [7], [10].

III-A. Definición del sistema

A continuación se comentan la solución propuesta por los autores de JuliaReach. La notación es la misma que la encontrada en su articulo pero la exposición se da de forma mucho menos rigurosa a fin de dar un vistazo al algoritmo propuesto para poder entender los puntos paralelizables del algoritmo original.

En este trabajo, el problema de la alcanzabilidad viene dado por:

$$\mathcal{X}(k+1) = \Phi\mathcal{X}(k) \oplus \mathcal{V}(k) \quad k = 0, 1, \ldots, N$$

Donde $\mathcal{V}(k)$ representa la contribución de entradas no deterministas o ruido, $\uplus$ denota la suma de Minkowski [11], $\mathcal{X}(0)$ representa un conjunto de estados iniciales y Φ es la matriz $n \times n$ que especifica el funcionamiento del sistema dinámico estudiado.

Los estados sucesivos $\mathcal{X}(k) \quad k = 1, 2, \ldots, N$ son los conjuntos de posibles estados que puede alcanzar el sistema dado el conjuntos de estados iniciales, las posibles entradas $\mathcal{V}(k)$ y la matriz Φ que gobierna el sistema.

El problema de escalabilidad de otros métodos propuestos dependen de la dimensión n de la matriz Φ. En cambio JuliaReach hace uso de la descomposición en menores dimensiones de la matriz para hacerse parcialmente independiente de n. Además, se aprovecha la dispersión de las matrices y sus potencias.

Supongamos un sistema dinámico gobernado por la ecuación diferencial:

$$x' = Ax(t) + Bu(t)$$

Con $u(t) \in \mathcal{U}(t)$ siendo un número de entradas no deterministas.

Se define *trayectoria* de la ecuación diferencial como la solución única del sistema $x_{x_0,u(t)}(t) : [0, T] \to \mathbb{R}$ dados un estado inicial x_0 y la entrada $u(t)$. T es el horizonte temporal analizado, que se asume finito.

A la union de trayectorias de las trayectorias desde todos los estados iniciales para la señal de entrada $u(t)$ la denominamos *conjunto de alcanzabilidad* (*reach set*). Dada la union de los

conjuntos de alcanzabilidad para todas las entradas obtenemos el *tubo de alcanzabilidad* (*reach tube*).

A continuación exponemos las etapas en las que se descomponen la solución propuesta por JuliaReach.

III-B. Discretización temporal

El primer paso es la discretización del dominio temporal en los casos donde este es continuo. Para esto, los autores del trabajo ofrecen los mismos mecanismos que los utilizados en SpaceEx, dos métodos de interpolación temporal hacia adelante y hace atrás conocidos como Le Guernic - Girard (LGG).

III-C. Descomposición de $\mathcal{X}(0)$

Consideremos que el estado $\mathcal{X}$ es aproximado por un poliedro y a continuación proyectado en conjuntos $\hat{\mathcal{X}}_1, \hat{\mathcal{X}}_2, \ldots, \hat{\mathcal{X}}_b$, su producto cartesiano es a su vez una sobre aproximación de $\mathcal{X}$.

$$\mathcal{X} \subseteq \hat{\mathcal{X}}_1 \times \hat{\mathcal{X}}_2 \times \ldots \times \hat{\mathcal{X}}_b$$

En particular, el cálculo de la aproximación por poliedros y su sucesiva proyección es hecha para el estado inicial $\mathcal{X}(0)$. Después, los elementos quedan en una representación mucho más económica desde el punto de vista computacional.

III-D. Cálculo de sucesores

La matriz Φ de $n \times n$ es descompuesta en b bloques de 2×2, de forma que la ecuación de evolución del sistema queda dada como:

$$\mathcal{X}(k+1) = \Phi\mathcal{X}(k) \oplus \mathcal{V}(k) =$$
$$\begin{pmatrix} \Phi_{11} & \cdots & \Phi_{1b} \\ \vdots & \ddots & \vdots \\ \Phi_{b1} & \cdots & \Phi_{Bb} \end{pmatrix} \mathcal{X}(k) \oplus \mathcal{V}(k)$$

Entonces, podemos obtener las b proyecciones de $\hat{\mathcal{X}}(k+1)$ como la sobreaproximación:

$$\hat{\mathcal{X}}_i(k+1) = \bigoplus_{j=1}^{b} \Phi_{ij}\hat{\mathcal{X}}_j \oplus \hat{\mathcal{V}}_i \quad \forall i = 1, \ldots, b$$

III-E. Proyección

Finalmente, se proyectan las variables de salida que se desean observar.

III-F. LazySets

Las solución Reachability del proyecto JuliaReach viene acomañado de un conjunto de liberías. En particular, LazySets, una librería para el cálculo en conjunto convexos que emplea técnicas de evaluación perezosa (*lazy evaluation*) para retrasar el cálculo de los resultados hasta que son requeridos. Esta librería, junto con Reachability se encuentran en el material modificado para el proyecto.

IV. METODOLOGÍA

Aunque la propuesta de JuliaReach es algorítmicamente más eficiente que las soluciones existentes como SpaceEx y Hylaa mediante la reducción de la dimensionalidad del problema y el aprovechamiento de la *sparcity* de las matrices involucradas, la implementación existente no hace aprovechamiento de las capacidades de computación paralela existentes en Julia.

JuliaReach implementa algoritmos para un gran número de paramétricas: comprobación de alcanzabilidad(*reach*) y verificación de seguridad(*check*), dominios temporales discretos y continuos, matrices dispersas y densos, con sobreaproximación por un número de direcciones, evaluación de perezosa de potencias de matrices, etc. En este proyecto se paralelizaron dos variaciones de dos parámetros distintos como prueba de concepto:

- **Evaluación perezosa de potencias de matrices**. Para matrices muy grandes y muy dispersas se puede utilizar la clase *SparseMatrixExp*. Es representan de forma simbólica el cálculo de sus potencias. Esta utiliza el método implementado en [1] basado en [14] aplicado de forma perezosa.
- **Aproximación por cajas y por direcciones octagonales.** En JuliaReach es posible determinar si la aproximación por poliedros es realizada por cajas (hiperrectángulos) o por direcciones octagonales. Estos últimos son vectores que son cero en todos salvo dos dimensiones con valores ± 1.

Luego de estudiado el trabajo original presentado por los desarrolladores de JuliaReach se procedió a realizar la optimización en un procedimiento de dos pasos: análisis de puntos calientes o *hotspots* y paralelización de la implementación.

IV-A. Análisis de puntos calientes

En esta etapa se ejecutó en las diferentes parámetros predeterminados el algoritmo y se observó los puntos de mayor tiempo de computación. Para esto se hizo uso extensivo de la macro *@time* que permite obtener la duración de la ejecución de una porción del código, la cantidad de alocaciones de memoria realizadas, la cantidad de bytes alocados y el porcentaje de tiempo empleado ejecutando el *garbage collector*. Un ejemplo la salida de una sentencia anotada con la macro *@time* se puede ver en el listado 6.

```
1630.946267 seconds (2.82 G allocations:
 ↪  95.367 GiB, 1.39% gc time)
```

Listing 6: Salida de la macro @time

Una característica del algoritmo de JuliaReach es la presencia de etapas claramente definidas, representadas en 4, con un relación de dependencia de los datos: para que cualquier etapa ejecute, se requiere que la etapa anterior se encuentre finalizada. Estas etapas son las presentadas en la sección anterior.

Esto permite identificar objetivos de optimización con fronteras bien definidas.

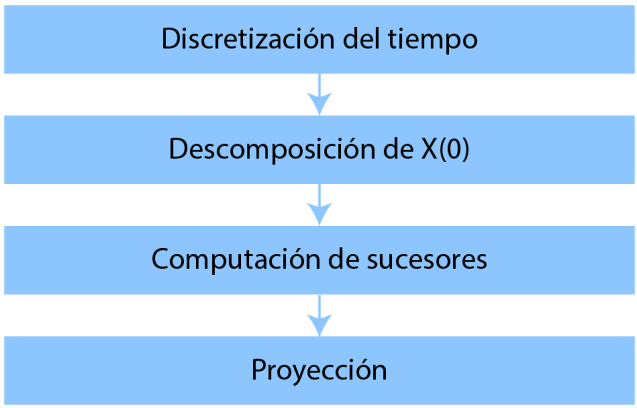

Figura 1. Etapas del algoritmo

IV-B. Paralelización de la implementación

Para realizar la paralelización se debió identificar la dependencia de datos y separar el algoritmo en pasos con capacidad de concurrencia. Los principales candidatos fueron aquellos que iteran sobre las dimensiones del problema de forma independiente. En este caso la paralelización consta de tareas idénticas sobre diferentes datos. Otros candidatos constituyeron tareas diferentes que no presentasen dependencia de datos entre ellos.

Se construyó un patrón general para la distribución de las tareas entre los procesos. Dada una tarea T a iterarse sobre un número N de entradas distribuidas en un arreglo con M trabajadores disponibles, al primer trabajador se le asignan las tareas sobre las primeras M/N entradas, al segundo las siguientes M/N y así sucesivamente.

```
entradas = Vector{T}(tamanoEntradas)
salidas = Vector{T}(tamanoEntradas)

for i in 1:tamanoEntradas
    salidas[i] = tarea(entrada[i])
end
```

Listing 7: Tarea sin paralelizmo

Suponiendo que *tarea* carece de efectos secundarios, el ejemplo 7 presenta un caso simplificado de un proceso paralelizable con una tarea a ejecutarse sobre un número de entradas para obtener un número igual de salidas. El patrón diseñado recibe las entradas sobre las que tiene que trabajar, como se ve en el ejemplo 8.

Sabiendo que Julia numera los procesos $1, 2, \ldots, N$, sin espacios, en el ejemplo 9 se ve una función que emplea el número de proceso para definir el rango asignado al proceso actual. Esto se hace calculando los cortes de forma de distribuir de forma lo más uniformemente posible las tareas entre todos los procesos.

Con un par de funciones adicionales presentadas en el ejemplo 10 podemos distribuir la tarea entre todos los procesos

```
function procesar_pedazo(
    entradas::Vector{T},
    rango::UnitRange{Int64})

    salidas = Vector{T}(length(rango))
        cuenta = 1
    for i in rango
        salidas[cuenta] = tarea(entrada[i])
        cuenta += 1
    end

    return salidas
end
```

Listing 8: Tarea a ejecutarse sobre un rango de entradas

```
function mi_rango(largo)
    # procesos numerados 1, 2, 3...
    indice = myid()
    # cantidad de procesos
    pedazos = length(procs())
    # devuelve los cortes de las particiones
    cortes = [round(Int, s)
        for s in linspace(0,largo,pedazos+1)]
    # porcion correspondiente al proceso actual
    return cortes[indice]+1:cortes[indice+1]
end
```

Listing 9: Asignación del rango de entradas a según el número de proceso

disponibles. Para cada proceso, realizamos una invocación remota con *remotecall* indicando el proceso p a invocar el método y los parámetros a recibir. Para cada tarea recibimos un futuro que representa el resultado de la ejecución del método. Con la macro *sync* esperamos a que las tareas enviadas a los procesos dentro del *scope* del bloque anotado sean finalizadas. Finalmente devolvemos un arreglo con los resultados de las tareas con *fetch*.

Finalmente basta llamar a *distribuir_tarea!*, como se ve en el ejemplo 11, para que se distribuya el procesamiento de la tarea entre todos los procesos.

Cabe mencionar, que el mecanismo descrito distribuye la tarea entre todos los procesos incluyendo el proceso 1 , que denota al proceso interactivo donde ejecuta REPL. Esto difiere con la macro @parallel que distribuye las tareas entre los procesos trabajadores, dejando disponible al primer proceso. Además, el patrón propuesto permite mayor flexibilidad a la hora de distribuir las tareas incluyendo el manejo de la memoria copiada y el valor retornado por las invocaciones remotas.

V. Implementación

A continuación se describen los cambios implementados a partir de los puntos calientes identificados.

```
function asignar_pedazo!(
    entradas::Vector{T}
    ) where {T} =
    procesar_pedazo!(entradas,
     ↪ mi_rango(length(entradas)))

function distribuir_tarea!(
        entradas::Vector{T}
    ) where {T}

    tareas =
     ↪ Vector{Future}(length(procs()))

    @sync begin
        for (i, p) in enumerate(procs())
            tareas[i] =
             ↪ remotecall(asignar_pedazo!,
             ↪ p, entradas)
        end
    end

    return [fetch(t) for t in tareas]
end
```

Listing 10: Distribución de las tareas entre los procesos

```
salidas = distribuir_tarea!(entradas)
```

Listing 11: Invocación del mecanismo de distribución de tareas

V-A. *Nuevo algoritmo*

JuliaReach emplea por defecto la opción *"explicit"* como valor del parámetro *:algorithm* para indicar el algoritmo a ejecutar para realizar el análisis de *reachability*. Para indicar que se emplearían la versión paralela del algoritmo se incorporó la opción *"explicit_parallel"*.

V-B. *Discretización del tiempo*

Esta etapa aplica un modelo de aproximación para transformar de un sistema afín de tiempo continuo en uno de tiempo discreto como es explicado en III-B. Fue paralelizado el método *discr_bloat_interpolation* que implementa la interpolación hacia adelante y hacia atrás (ver [10, Lemma 3]).

En esta etapa se ajustaron dos métodos: *symmetric_interval_hull* que aproxima un conjunto convexo por un hiperrectángulo centrado en el origen y *get_columns* de las matrices potencia perezosas implementadas en la clase *SparseMatrixExp*, ambos de la librería de LazySets.

Como se ve en el listado 12, el algoritmo itera las dimensiones de S y devuelve una tupla de dos arreglos. El método es sencillamente paralelizable distribuyendo el cálculo para cada dimensión entre los procesos ya que el este es independiente en cada una de las dimensiones. Además, se emplearon dos arreglos unidimensionales (*SharedVector*) para almacenar los resultados en vez de devolver tuplas.

Lo que ocurre con el método presentado en el listing 13 es similar. Este método es empleado para devolver las columnas

```
function
 ↪ symmetric_interval_hull(S::LazySet{N})
 ↪ where {N<:Real}
    zero_N = zero(N)
    one_N = one(N)
    n = dim(S)
    c = Vector{N}(n)
    r = Vector{N}(n)
    d = zeros(N, n)
    @inbounds for i in 1:n
        d[i] = one_N
        htop = rho(d, S)
        d[i] = -one_N
        hbottom = -rho(d, S)
        d[i] = zero_N
        c[i] = (htop + hbottom) / 2
        r[i] = (htop - hbottom) / 2
    end
    return c, r
end
```

Listing 12: Algoritmo para la aproximación por cajas

indicadas en el arreglo J de la matriz potencia Φ^k. En este caso se utilizó el patrón presentado en IV-B para resolver de forma paralela el valor de cada columna y luego concatenar el resultado antes de devolverlo.

```
function
 ↪ get_columns(spmexp::SparseMatrixExp{N},
 ↪ J::AbstractArray)::Matrix{N} where {N}
    n = size(spmexp, 1)
    aux = zeros(N, n)
    ans = zeros(N, n, length(J))
    count = 1
    one_N = one(N)
    zero_N = zero(N)
    @inbounds for j in J
        aux[j] = one_N
        ans[:, count] = expmv(one_N,
         ↪ spmexp.M, aux)
        aux[j] = zero_N
        count += 1
    end
    return ans
end
```

Listing 13: Algoritmo para la obtención de columnas de una matriz exponencial

V-C. *Descomposición del estado* $\mathcal{X}(0)$

La descomposición del estado se encuentra parametrizada y permite definir aspectos como la dimensiones de la descomposición (cajas, diagonales de cajas, direcciones octagonales) o la precisión requerida de la descomposición (con el parámetro ϵ) cuando se utiliza refinamiento iterativo [2].

Cada bloque de interés de la matriz, definidos por la variables a analizar, es iterado y proyectado como se muestra en III-C. Basta con utilizar la macro *parallel* con *vcat* como función reductora para obtener el resultado de la proyección en cada dirección de la descomposición.

V-D. Computación de sucesores

A partir del estado inicial descompuesto se calculan de forma iterativa las aproximaciones sucesivas del sistema sujeto a las entradas. En este caso se paralelizaron las versiones dispersas con y sin potencias perezosas de matrices.

Ambas variantes del algoritmo son similares con pequeñas variaciones en las representaciones y clases empleadas (*SparseMatrixCSC* vs *SparseMatrixExp*). El algoritmo itera para cada paso de tiempo de $t, t+\delta$. Los cálculos de los bloques se realizan linea a linea avanzando de forma horizontal. El resultado por cada linea de bloques es independiente de las otras por lo que el algoritmo es paralelizable en este punto.

Como se ve en 14, en este caso se empleó el patrón propuesto en la sección IV-B tomando en cuenta que existían dos que se podían realizar en la misma iteración, el cálculo de $\hat{\mathcal{X}}(k)$ y de $\hat{\mathcal{V}}(k)$.

```
@inbounds while true
  Xhatk, Whatk = distrubte_chunks!( ... )

  if k == N
      break
  end

  phipowerk = phipowerk * phi
  k += 1
end
```

Listing 14: Algoritmo simplificado del cálculo de sucesores paralelizado

V-E. Otras consideraciones

Además de las cuestiones propias de la paralelización, se tomaron en cuenta otros aspectos asociados a la performance en Julia a partir de las propuestas de [5].

- **Evitar contenedores de tipos abstractos** Para código altamente performante, se debe evitar el uso de contenedores de tipos abstractos como *Vector{Real}* ya que deben alojar memoria para almacenar cualquier tipo que herede de *Real* como *AbstractFloat*, *Integer*, *Rational* o *Irrational*.
- **Indicar el tipo de fuentes de datos sin tipo** Para permitir ciertas optimizaciones por parte del compilador, las variables deben estar tipadas. Si un método devuelve una variable no tipada, es aconsejable indicar el tipo al asignarla, como se muestra en el ejemplo 15
- **Estabilidad de tipos** Es importante que las variables mantengan su tipo a lo largo su *scope* así como las funciones devuelvan siempre un mismo tipo. Esto permite al compilador realizar optimizaciones así como evita múltiples realocaciones de memoria.
- **Dividir funciones** El compilador de Julia realiza la especialización de código en la frontera de las funciones. Dividir funciones permite al compilador especializarlas e inlinearlas, permitiendo código mucho más performante.

```
# Si foo devuelve Any, x es de tipo Any
x = foo(a)
# y es de tipo Int. Si foo devuelve otra
 ↪ cosa, se lanza una excepción
y = foo(b)::Int
```

Listing 15: Forzar el tipado

VI. Análisis experimental

Se tomaron dos benchmarks para la comparación de los resultado. En primer lugar, un caso muy grande con 10913 variables, uno de los casos densos más grandes ensayados en la literatura, a fin de determinar la posibles ventajas de la paralelización del algoritmo. Por el otro lado, un caso mucho menor con 270 variables, para estudiar el posible overhead del uso de computación paralela para problemas muy pequeños. Estos ejemplos fueron empleados por los desarrolladores de JuliaReach y obtenidos de [9].

Para la evaluación cuantitativa se consideraron las duraciones de las cuatro etapas de algoritmo. Aunque Julia consta con herramientas minimales para el perfilado de memoria, se decidió enfocarse en la optimización en el dominio temporal sin asumir restricciones en el espacio. Las pruebas fueron realizadas sobre un estación de trabajo con procesador i7 7700HQ de 4 núcleos físicos y 2 hilos cada uno y 16 Gb de RAM. La pruebas se ejecutaron desde un notebook de Jupyther corriendo al versión 0.6.2.2 de Julia.

VI-A. MNA 5

El primer modelo empleado utiliza matrices dispersas, con 10913 variables, 9 entradas y 9 salidas. Este sistema describe el estado de un circuito eléctrico con resistencias, inductores y capacitores utilizando *Modified Nodal Analysis* [13].

Se habilitó la utilización de matrices dispersas y la potencia perezosa (*lazy exponential*) de estas matrices utilizando el método implementado en la librería [1]. Se tomaron $N = 3000$ pasos de tiempo con un $\delta = 1 \times 10^{-3} s$ lo que totalizan $T = 3s$.

Cuadro I
Resultados de MNA5 para descomposición por cajas

	Paralelo	**Serial**	**Speed up**
Discretización	230	443	1,93
Descomposición	133	924	6,95
Sucesores	244	127	0,52
Proyección	0,29	0,02	0,08
Total	607,29	1494,02	2,46
Total de partes fijas	363,29	1367,02	3,76

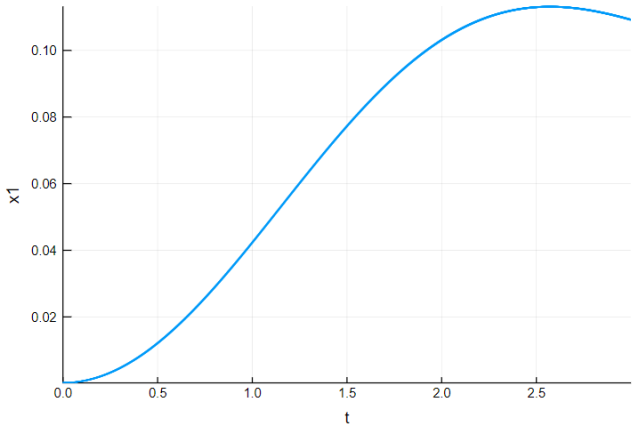

Figura 2. Resultado del modelo ISS con $N = 3000$ y $\delta = 1 \times 10^{-3}$

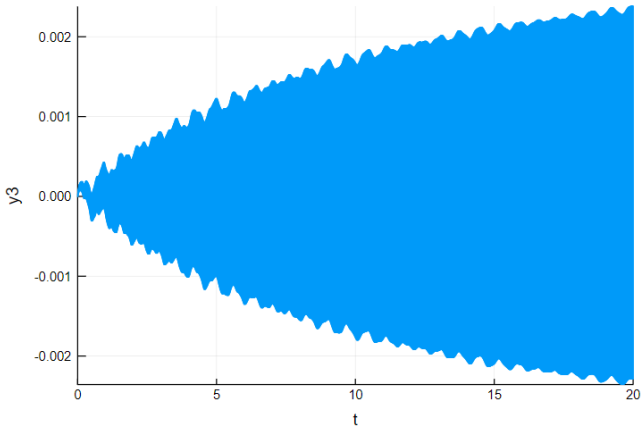

Figura 3. Resultado del modelo ISS con $T = 20$

Cuadro II
RESULTADOS DE MNA5 PARA DESCOMPOSICIÓN POR DIRECCIONES OCTAGONALES

	Paralelo	**Serial**	**Speed up**
Discretización	227,45	437,19	1,92
Descomposición	247,26	1839	7,44
Sucesores	1155	196,33	0,17
Proyección	1,10	1,01	0,92
Total	1630,81	2473,97	1,52
Total de partes fijas	475,81	2278	4,79

VI-B. Estación espacial

El segundo modelo empleado es el del componente 1r (módulo de servicio ruso), empleado en la estación espacial internacional. Consta de 270 variables, tres entradas y tres salidas [6]. Se realizaron los cálculos para $T = 20s$ y $\delta = 5 \times 10^{-3}$.

Cuadro III
RESULTADOS DE ISS PARA DESCOMPOSICIÓN POR CAJAS

	Paralelo	**Serial**	**Speed up**
Discretización	0,62	0,73	1,19
Descomposición	0,02	0,07	3,20
Sucesores	126,31	30,05	0,24
Proyección	0,01	0,02	1,29
Total	126,96	30,87	0,24
Total de partes fijas	0,65	0,82	1,25

Cuadro IV
RESULTADOS DE ISS PARA DESCOMPOSICIÓN POR DIRECCIONES OCTOGONALES

	Paralelo	**Serial**	**Speed up**
Discretización	0,58	0,58	0,99
Descomposición	0,03	0,07	2,78
Sucesores	447,58	315,19	0,70
Proyección	0,06	0,06	1,09
Total	448,24	315,9	0,70
Total de partes fijas	0,66	0,71	1,07

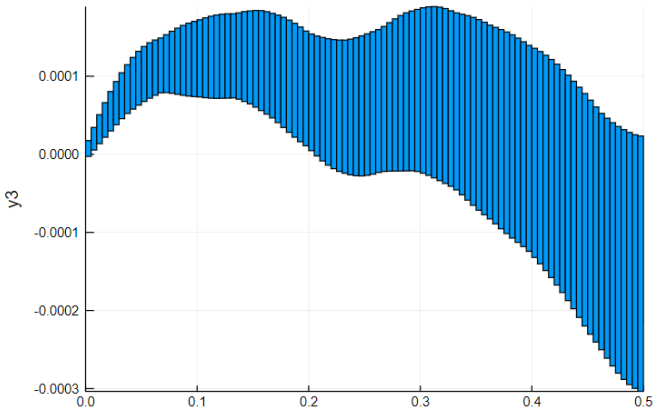

Figura 4. Resultado del modelo con $T = 0{,}5$

VII. ANÁLISIS DE LOS RESULTADOS

La diferencia es notoria en el caso de MNA5 y el de ISS. Para el primero, el *speedup* es significativo para todos los casos salvo el cálculo de sucesores mientras que en el caso de ISS el speedup es mucho menos significativo.

El caso de ISS es justificable dada la sencillez del modelo. Resulta muy costoso el uso de procesos, mecanismos de comunicación, serialización y deserialización de entidades para le ejecución de tareas de muy corta duración. El *overhead* esconde los beneficios de la paralelización.

En el caso del cálculo de sucesores el problema es más acusado. Cada iteración es muy corta y requiere la copia de una gran cantidad de información, resultando así sumamente costoso el uso de estos mecanismo por lo que representa en un speedup menor a uno.

Cabe notar que se manejan dos tipos de tiempos: fijos y variables. Por un lado, la discretización del dominio temporal, la descomposición de $\mathcal{X}(0)$ y la proyección son tiempos constantes que dependen del problema y no de la cantidad de pasos a ejecutar. Por el otro lado, el cálculo de sucesores es variable y depende particularmente de cuanto se quiera extender la simulación en el eje temporal.

En una simulación muy corta, probablemente las etapas de tiempo fijo dominen a las etapas de tiempo variable, mientras que en una simulación con un horizonte muy grande, el cálculo de sucesores tome gran parte del tiempo de procesamiento. Por eso a partir de ahora consideraremos las partes del procesamiento fijas. Como es sencillo deshabilitar el paralelismo para el calculo de sucesores donde no resulta beneficioso, asumiremos el mejor resultado en este caso, el del caso serial, y continuaremos el análisis considerando únicamente las etapas de discretización temporal y descomposición del estado inicial $\mathcal{X}(0)$. La etapa de proyección queda fuera de discusión dado que representa un porcentaje despreciable del tiempo total.

La porción de tiempo que toman estos procesos de las etapas de duración fija se ven en las tablas VI y V

Cuadro V
TIEMPO PORCENTUAL DE MNA5 DE LAS ETAPAS FIJAS

	Cajas		**Octagonales**	
	Paralelo	**Serial**	**Paralelo**	**Serial**
Discretización	63 %	32 %	48 %	19 %
Descomposición	37 %	68 %	52 %	81 %
Proyección	0 %	0 %	0 %	0 %

Cuadro VI
TIEMPO PORCENTUAL DE ISS DE LAS ETAPAS FIJAS

	Cajas		**Octagonales**	
	Paralelo	**Serial**	**Paralelo**	**Serial**
Discretización	95 %	90 %	88 %	81 %
Descomposición	3 %	8 %	4 %	10 %
Proyección	2 %	2 %	8 %	9 %

Como se puede observar, el tiempo que la descomposición frente a la duración total incrementa al utilizar un número mayor de direcciones para la aproximación por poliedros del estado inicial. Esto es esperable considerando que esta etapa hace especial uso de los métodos *project* y *overapproximate*.

Es notoria la mejora de alrededor del $600\,\%$ en la descomposición de $\mathcal{X}(0)$, por encima del número de procesos corriendo en paralelo. En estos casos correspondería analizar más a fondo otros factores como coherencia del caché, *false sharing* y otras particularidades de los sistemas multiprocesador.

Por eso es importante tomar en cuenta que el incremento de performance se obtuvo sobre las etapas de tiempo fijo y no la de tiempo variable y aunque en el ejemplo de MNA5 la paralelización representó un descenso considerable del tiempo de procesamiento , en simulaciones muy extendidas es posible que su contribución disminuya significativamente.

VIII. CONCLUSIONES

La paralelización de la implementación de JuliaReach demostró ser de mayor utilidad en los casos donde JuliaReach ya se destaca frente al estado del arte: casos de gran dimensionalidad con matrices dispersas. De hecho, en las pruebas realizadas por Forets et al. en el articulo original, *SpaceEx* no logró finalizar la ejecución de las pruebas de MNA5.

Por lo tanto, para simulaciones dentro de un tiempo definido, la mejora inducida por el paralelismo es considerable. pero, en cambio, por encima de este horizonte, la parte del variable del tiempo de procesamiento domina a la fija y el impacto es mucho menor.

La etapa de duración variable de cálculo de sucesores requeriría de un acondicionamiento y calibración fina que balancee apropiadamente el *overhead* de la inicialización y mantenimiento de procesos paralelos para obtener una mejora en el tiempo de ejecución frente a caso serial.

De todas formas, la pararelización del algoritmo se considera satisfactoria a efectos de una prueba de concepto. El algoritmo resultó sencillamente paralelizable en muchos aspectos claves gracias a la independencia estructural de los componentes que lo conforman: la independencia de las dimensiones de la proyección de los conjuntos así la independencia de los bloques la matriz Φ que define al sistema. Estos elementos hacen del algoritmo un objetivo natural de la computación de alta performance.

El haber alcanzado una optimización no despreciable empleando una estación de trabajo de propósito general abre el camino al uso de la librería en ambientes académicos y profesionales así como a futuras optimizaciones en entornos especializados que hagan un mejor uso de la predisposición natural del algoritmo a la paralelización como son las tarjetas gráficas que soportan computación de propósito general.

IX. FUTURO

Este trabajo representó una prueba de concepto para la implementación de las herramientas de computación paralela provistas por Julia en el marco de JuliaReach. El alcance no se acercó a una implementación completa sobre las librerías Reachability y LazySets. El desarrollo de estas características requeriría consideraciones arquitectónicas y de diseño fuera del alcance de este trabajo.

Además, se sugiere la utilización de las librerías de *threading* de Julia para la reducir el *overhead* asociado a las herramientas multiproceso.

Por otro lado, existen otros enfoques no considerados como son el uso de cómputo de unidades gráficas para el procesamiento de propósito general.

X. CÓDIGO FUENTE

El código fuente de las librerías modificas se encuentra en los repositorios:

- **https://github.com/bgarate/Reachability.jl** *branch: paralelizacion*
- **https://github.com/bgarate/LazySets.jl** *branch: paralelizacion*

Además, se incorpora el repositorio que incluye el notebook Jupyter de los benchmarks realizados:

- **https://github.com/bgarate/reachability-benchmark** *branch: master*

REFERENCIAS

[1] Expokit.jl.

[2] Iterative Refinement · LazySets.jl.

[3] Julia Micro-Benchmarks.

[4] JuliaReach.

[5] Performance Tips · The Julia Language.

[6] Athanasios C Antoulas, Danny C Sorensen, and Serkan Gugercin. A survey of model reduction methods for large-scale systems. Technical report, 2000.

[7] Stanley Bak and Parasara Sridhar Duggirala. Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, HSCC '17, pages 173–178, New York, NY, USA, 2017. ACM.

[8] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Andreas Podelski, Christian Schilling, and Frédéric Viry. Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices. *CoRR*, abs/1801.09526, 2018.

[9] Younes Chahlaoui and Paul Van Dooren. Benchmark examples for model reduction of linear time-invariant dynamical systems. In Peter Benner, Danny C. Sorensen, and Volker Mehrmann, editors, *Dimension Reduction of Large-Scale Systems*, pages 379–392, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer, 2011.

[11] H. Hadwiger. Minkowskische addition und subtraktion beliebiger punktmengen und die theoreme von erhard schmidt. *Mathematische Zeitschrift*, 53(3):210–218, Jun 1950.

[12] Emmanuel Hainry. Reachability in linear dynamical systems. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 241–250, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Altan Odabasioglu, Mustafa Celik, and Lawrence T Pileggi. Prima: Passive reduced-order interconnect macromodeling algorithm. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 58–65. IEEE Computer Society, 1997.

[14] Roger B. Sidje. Expokit: A software package for computing matrix exponentials. *ACM Trans. Math. Softw.*, 24(1):130–156, March 1998.