

## Studio esecutivo: LAR cuboids & simplicial

Nel seguente documento verrà illustrato lo studio esecutivo per il progetto numero 1 - **LAR cuboids & simplicial**, nel quale saranno descritte le ottimizzazioni fatte per i due file sorgenti *largrid.jl* e *simplexn.jl* presenti all'interno della libreria **Lar** (Linear Algebraic Representation).

### Autori:

- Alessandro Dell'Oste: 502589
- Maurizio Brini: 505195
- Manuel Granchelli: 512406

### Repo GitHub:

<https://github.com/adelloste/LinearAlgebraicRepresentation.jl>

### Indice

- Obiettivo
- Analisi
- Test

#### Obiettivo

L'obiettivo preliminare del seguente studio esecutivo è stato quello di analizzare i file sorgente e individuare le eventuali ottimizzazioni da poter effettuare all'interno del codice, facendo riferimento a quanto discusso in aula durante le lezioni e ai capitoli del libro consigliato **Julia High Performance**. In seguito, sono state testate alcune tecniche di ottimizzazione sui file sorgenti *simplexn.jl* e *largrid.jl*.

#### Analisi

L'analisi è partita dal file sorgente **largrid.jl** nel quale sono state analizzate inizialmente le funzioni più compatte, come ad esempio le funzioni: **qn**, **grid\_0**, **grid\_1**, nelle quali sono stati cambiati i tipi delle variabili, come per esempio, è stato cambiato il tipo *Array{T,1}* con il tipo *Vector{T}*. Il cambio di tipo alle variabili, anche se non di molto, ha migliorato le prestazioni delle funzioni. Un'altra ottimizzazione fatta nella prima parte è stata quella di utilizzare la funzione *reduce* insieme alla funzione *vcat* nelle parti di codice dove era presente sola la funzione *vcat*. Anche in questo caso c'è stato un piccolo miglioramento delle prestazioni, in quanto, utilizzando le due funzioni insieme c'è una minore allocazione di memoria.

Successivamente, sono state considerate le altre funzioni, nelle quali inizialmente sono stati aggiornati, anche in questo caso, i tipi delle variabili ed in seguito, in alcune funzioni, sono state aggiunte le macro **@inline**, **@inbound** e **@simd**.

Per quanto riguarda la macro `@inline` è utile ed è stata utilizzata a livello di funzione. In Julia il compilatore ha determinate euristiche con le quali esegue l'inline automaticamente e in genere tale euristica è basata sulla dimensione del corpo della funzione. Infatti, per le funzioni piccole la macro `@inline` viene eseguita automaticamente e nel seguente caso la macro è stata aggiunta alle funzioni che vengono richiamate all'interno di altre funzioni, come ad esempio la funzione `larGridSkeleton` che viene richiamata all'interno di `cuboidGrid`. `@inline` è stata aggiunta prima della definizione della funzione sulla stessa riga, come nel seguente esempio:

```
@inline function f_inline()
    ...
end
```

La macro `@inbounds`, invece, come viene definita sul libro, serve per eliminare il costo del controllo dei limiti. Ci sono operazioni extra per le letture e le scritture in un array che hanno un costo molto basso e di solito sono un buon compromesso per la sicurezza. Quando, in alcune situazioni, si è sicuri che i limiti dell'array non vengono mai superati è possibile rimuovere tali controlli con la macro in questione. Nel seguente caso la macro è stata utilizzata all'interno delle funzioni quando sono presenti dei cicli `for` che accedono ad array. `@inbounds` è stata aggiunta prima di un ciclo `for` sulla stessa riga, come nel seguente esempio:

```
function f_inbounds(a)
    @inbounds for i in 2:size(a, 1)
        ...
    end
end
```

L'ultima macro utilizzata nel seguente studio esecutivo è la macro `@simd`. Single Instruction, Multiple Data (SIMD) è un metodo per parallelizzare il calcolo all'interno della CPU, in base al quale una singola operazione viene eseguita su più elementi di dati contemporaneamente. In Julia l'inserimento in un ciclo `for` di questa macro (`@simd`) offre al compilatore la libertà di utilizzare le istruzioni SIMD per le operazioni all'interno del ciclo `for` in questione. Tale macro non rende ogni ciclo più veloce, infatti, per poter essere utilizzata è necessario che il ciclo abbia alcune proprietà fondamentali. `@simd` è stata aggiunta prima di un ciclo `for` sulla stessa riga, come nel seguente esempio:

```
function f_simd(x)
    @inbounds @simd for i = 1:length(x)
        ...
    end
end
```

Inoltre, nel seguente studio esecutivo è stata testata anche la macro `@threads`. Tale macro può eseguire le iterazioni di un ciclo `for` su un thread diverso in base al numero di threads disponibili sulla nostra macchina. Nei test effettuati la

macro ha portato a peggioramenti dei tempi di esecuzione perché, come riporta il libro consigliato **Julia High Performace**, se all'interno del `for` sono presenti funzioni che modificano la stessa posizione in memoria, la macro peggiora le prestazioni. Per poter utilizzare `@threads` occorre importare il modulo `Threads` tramite il comando `using Base.Threads`. `@threads` è stata testata nel seguente modo, aggiungendola prima di un ciclo `for` sulla stessa riga:

```
function f_threads(x)
    @threads for i in 1:nthreads()
        a[threadid()] = threadid()
    end
end
```

Infine, oltre all'aggiunta di macro e al cambio del tipo delle variabili, alcune funzioni sono state suddivise e sono state create nuove funzioni che vengono chiamate all'interno di funzioni esistenti. Un esempio è la funzione **larModelProduct** all'interno del file **largrid.jl**, nella quale una parte della funzione si occupa di creare i vertici e un'altra parte della creazione delle celle. Per entrambe le due parti di codice sono state create due funzioni (`createVertices()` e `createCells()`), che vengono richiamate all'interno della nuova funzione **larModelProduct**.

Le macro e le varie ottimizzazioni successivamente sono state eseguite anche all'interno del file **simplexn.jl**.

Le modifiche al codice sono state eseguite inizialmente sui notebooks *largrid.ipynb* e *simplexn.ipynb* e successivamente sono stati modificati i due file sorgenti **largrid.jl** e **simplexn.jl**. Una volta eseguite le modifiche sono stati lanciati i test presenti all'interno del **Repo GitHub** nella cartella **test** per verificare il corretto funzionamento del codice. Nel successivo studio si cercherà di migliorare e ottimizzare ulteriormente il codice.

## Test

Nella seguente sezione sono riportati alcuni dei test effettuati su alcune funzioni ottimizzate dei due file sorgenti. I test sono stati effettuati all'interno del notebook utilizzando la macro `@btime`. Come si può osservare dall'output delle funzioni ottimizzate, in alcuni casi, le funzioni sono migliorate di molto (esempio: funzione **grid** - (Figura (2)))

## grid

Output funzione:

```
@btime grid(1,-1,1,-1,1,-1,1,-1,-1)
✓ 3.6s

2.384 μs (51 allocations: 3.78 KiB)

([0.0 1.0 ... 9.0 10.0], [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
```

Figure 1: grid

Output funzione ottimizzata:

```
@btime grid_opt(1,-1,1,-1,1,-1,1,-1,-1)
✓ 2.2s

476.072 ns (11 allocations: 1.12 KiB)

([0.0 1.0 ... 9.0 10.0], Any{[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]]})
```

Figure 2: grid\_opt

## grid\_0

Output funzione:

```
@btime grid_0(5)
✓ 2.4s

753.302 ns (12 allocations: 864 bytes)

1×6 Matrix{Int64}:
 0  1  2  3  4  5
```

Figure 3: grid\_0

Output funzione ottimizzata:

```
@btime grid_0_opt(5)
✓ 1.9s

164.369 ns (2 allocations: 224 bytes)

1×6 Matrix{Int64}:
 0  1  2  3  4  5
```

Figure 4: grid\_0\_opt

**qn**

Output funzione:

```
@btime qn(3)([1.5,-2,0.5])
✓ 1.9s

2.245 μs (33 allocations: 1.95 KiB)

([0.0 1.5 ... 11.5 12.0], [[1, 2], [3, 4], [4, 5], [6, 7], [7, 8], [9, 10]])
```

Figure 5: qn

Output funzione ottimizzata:

```
@btime qn_opt(3)([1.5,-2,0.5])
✓ 1.4s

1.546 μs (29 allocations: 1.78 KiB)

([0.0 1.5 ... 11.5 12.0], [[1, 2], [3, 4], [4, 5], [6, 7], [7, 8], [9, 10]])
```

Figure 6: qn\_opt

## larVertProd

Output funzione:

```
@btime larVertProd([larGrid(2)(0), larGrid(2)(0)])  
✓ 4.3s  
  
4.810 μs (76 allocations: 5.28 KiB)  
  
2×9 Matrix{Int64}:  
 0  0  0  1  1  1  2  2  2  
 0  1  2  0  1  2  0  1  2
```

Figure 7: larVertProd

Output funzione ottimizzata:

```
@btime larVertProd_opt([larGrid_opt(2)(0), larGrid_opt(2)(0)])  
✓ 2.8s  
  
3.417 μs (58 allocations: 4.25 KiB)  
  
2×9 Matrix{Int64}:  
 0  0  0  1  1  1  2  2  2  
 0  1  2  0  1  2  0  1  2
```

Figure 8: larVertProd\_opt

## simpleGrid

Output funzione:

```
@btime simplexGrid([1,1,1])  
✓ 5.3s  
  
7.052 μs (138 allocations: 9.47 KiB)  
  
([0.0 1.0 ... 0.0 1.0; 0.0 0.0 ... 1.0 1.0; 0.0 0.0 ... 1.0 1.0], [[1, 2, 3, 5], [2, 3, 5, 6], [3, 5, 6, 7], [2, 3, 4, 6], [3, 4, 6, 7], [4, 6, 7, 8]])
```

Figure 9: simpleGrid

Output funzione ottimizzata:

```
@btime simplexGrid_opt([1,1,1])  
✓ 4.7s  
6.989 μs (138 allocations: 9.47 KiB)  
([0.0 1.0 - 0.0 1.0; 0.0 0.0 - 1.0 1.0; 0.0 0.0 - 1.0 1.0], [[1, 2, 3, 5], [2, 3, 5, 6], [3, 5, 6, 7], [2, 3, 4, 6], [3, 4, 6, 7], [4, 6, 7, 8]])
```

Figure 10: simplexGrid\_opt