Relazione LAR TGW 3D - Gruppo 8b

Luca Maria Lauricella Valerio Marini

June 20, 2022

Progetto relativo al Corso di Calcolo Parallelo e Distribuito del Prof. Paoluzzi presso l'Università Roma Tre.

Repository del progetto: https://github.com/lauriluca99/TGW-3D.jl

Documentazione del progetto: https://lauriluca99.github.io/TGW-3D.jl

Contents

Studio esecutivo

Nello studio esecutivo abbiamo analizzato il codice nei notebooks cercando delle possibili ottimizzazioni. Non è stato possibile ottimizzare tutte le funzioni, infatti le principali modifiche sono state effettuate nelle funzioni: **frag_face** e **merge_vertices**.

Per migliorare il codice, sono stati presi in considerazione i libri: *Julia High Performance* e *Hands-On Julia Programming*, nei quali vengono menzionate le seguenti macro per migliorare le performance e la stabilità del codice:

- `@async`: racchiude l'espressione in un Task ed inizierà con l'esecuzione di questa attività procedendo con qualsiasi altra cosa venga dopo nello script, senza aspettare che il Task termini.
- `@sync`: contrariamente al precedente, questa macro aspetta che tutti i Task creati dalla parallelizzazione siano completati prima di proseguire.
- `Thread.@spawn`: Crea un Task e schedula l'esecuzione su un qualsiasi thread disponibile. Il Task viene assegnato ad un Thread quando diventa disponibile.

- `@simd`: si utilizza solo nei for per permettere al compilatore di avere più libertà nella gestione del ciclo consentendo di riordinarlo.
- `@inbounds`: elimina il controllo dei limiti degli array all'interno dell'espressione
- `@views`: converte le operazioni di taglio sull'array in una data espressione per ritornare una variabile di tipo View.
- `@code_warntype`: viene utilizzato per individuare i problemi causati dai tipi delle variabili, operando conseguentemente con un'assegnazione specifica che riduce la complessità del codice.
- `@benchmark`: questa macro può essere usata solo davanti alle chiamate di funzione. Valuta i parametri della funzione separatamente e chiama la funzione più volte per costruire un campione di tempi di esecuzione.
- `@btime`: simile a `@benchmark` ma restituisce meno informazioni, quali il tempo minimo ed il numero di allocazioni.
- `@profile`: questa macro esegue l'espressione collezionando dei campionamenti periodici. Nei campioni si può vedere la gerarchia delle funzioni ed il tempo di esecuzione di ogni riga.

frag_face

Utilizzando la macro `@code_warntype` si individuano molte variabili assegnate al tipo `Any`. Questo significa essenzialmente che ci sarà un'allocazione per la posizione della memoria e l'indirezione al valore effettivo durante l'esecuzione della funzione.

```
@benchmark frag_face(Lar.Points(V),EV,FE,[2,3,4,5],2)
```

```
BenchmarkTools.Trial: 8171 samples with 1 evaluation.
 Range (min ... max):  376.300  s ... 59.363 ms   GC (min ... max):  0.00% ... 97.57%
 Time  (median):     436.200  s              GC (median):     0.00%
 Time  (mean ± ):   607.541  s ±  1.482 ms   GC (mean ± ):  16.21% ±  7.21%
```

```
  376  s          Histogram: frequency by time         1.17 ms <

 Memory estimate: 265.71 KiB, allocs estimate: 4874.
```

Tramite ProfileView otteniamo un grafico in cui si ottiene la misurazione temporale di ogni singola riga di codice. La larghezza delle barre mostra il tempo trascorso in ogni locazione di chiamata, mentre la gerarchia di chiamata è rappresentata dalle varie altezze del grafico.

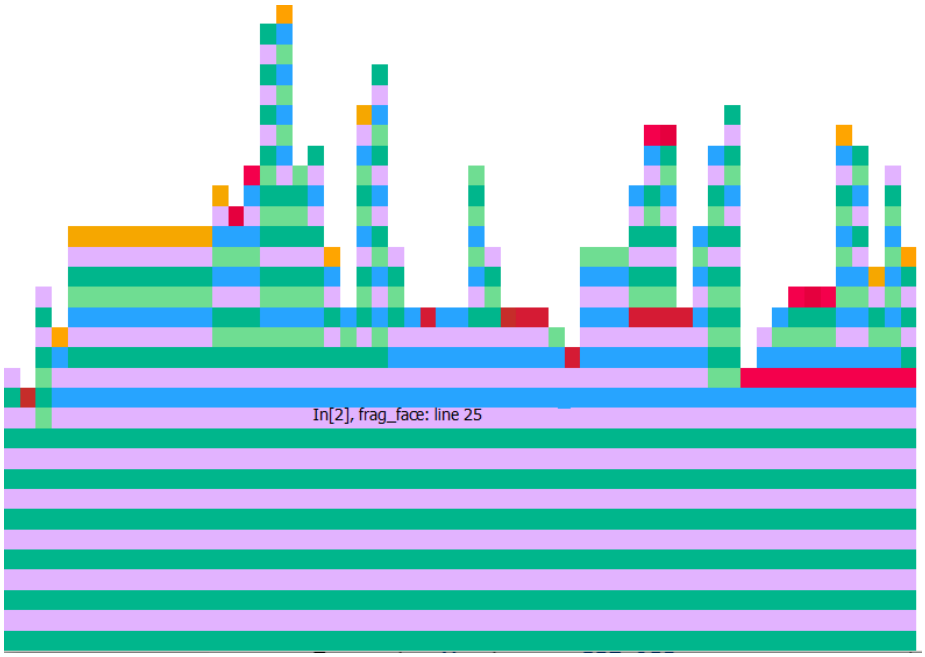

Figure 1: Grafico di ProfileView della funzione originale

Per ottimizzare la funzione abbiamo assegnato alle variabili locali un tipo deterministico per rimuovere il tipo `Any` ed avere la funzione *type-stable*. Inoltre si possono creare delle viste degli array quando c'è un'operazione di slicing, con la macro `@views`, le quali permettono di accedere ai valori dell'array senza dover effettuare una copia.

Dopo aver eseguito vari test, si è optato per utilizzare la macro `@async` per parallelizzare il ciclo *for* che calcola l'intersezione della faccia sigma con le facce in *sp_idx[sigma]*.

Quindi, applicando le suddette modifiche, si è raggiunto un tempo minimo di esecuzione inferiore di circa 20% dalla versione originale.

```
@benchmark frag_face2(Lar.Points(V),EV,FE,[2,3,4,5],2)
```

```
BenchmarkTools.Trial: 9179 samples with 1 evaluation.
 Range (min ... max):  297.300  s ... 35.314 ms   GC (min ... max):  0.00% ... 97.50%
 Time  (median):     376.200  s               GC (median):     0.00%
 Time  (mean ± ):   540.377  s ±  1.575 ms   GC (mean ± ):  19.48% ±  6.66%


  297  s          Histogram: frequency by time          924  s <
```

```
Memory estimate: 218.65 KiB, allocs estimate: 3873.
```

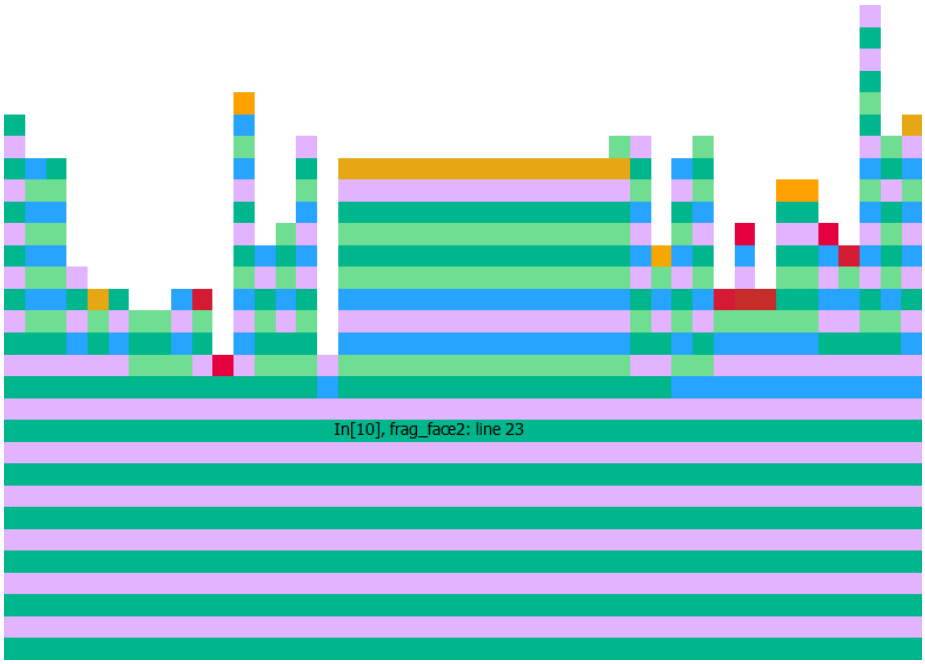

Figure 2: Grafico di ProfileView della funzione modificata

merge_vertices

Anche per quanto riguarda questa funzione abbiamo effettuato un controllo su i vari tipi di variabili assegnati utilizzando la macro `@code_warntype`, questa volta però non erano presenti variabili di tipo `Any` che ci avrebbero dunque destabilizzato i tipi della funzione. Per ottimizzare quest' ultima abbiamo, un'altra volta, introdotto opportunamente davanti ai cicli *for* la macro `@async` che, parallelizzando le operazioni di calcolo, ci ha permesso di avere un'ottimizzazione del 30% sul tempo di esecuzione.

```
@benchmark merge_vertices(Lar.Points(V),Lar.ChainOp(EV),Lar.ChainOp(FE),1e-4)

BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min ... max):  39.200  s ...  13.339 ms   GC (min ... max):  0.00% ... 99.16%
 Time  (median):     49.400  s               GC (median):     0.00%
 Time  (mean ± ):   74.714  s ± 368.686  s   GC (mean ± ):  18.01% ±  3.68%
```

```
  39.2  s       Histogram: log(frequency) by time       141  s <

 Memory estimate: 64.00 KiB, allocs estimate: 920.
```

Notiamo in seguito come vengono modificati i tempi dopo l'ottimizzazione.

```
@benchmark merge_vertices2(Lar.Points(V),Lar.ChainOp(EV),Lar.ChainOp(FE),1e-4)

BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min ... max):  34.700  s ... 94.229 ms   GC (min ... max):  0.00% ... 99.75%
 Time  (median):     54.600  s               GC (median):     0.00%
 Time  (mean ± ):   99.073  s ±  1.576 ms   GC (mean ± ):  27.47% ±  1.73%
```

```
  34.7  s         Histogram: frequency by time         218  s <

 Memory estimate: 56.56 KiB, allocs estimate: 779.
```

spatial_arrangement

Di nuovo, anche per questa funzione è stato effettuato un controllo su i vari tipi di variabili utilizzando la macro `@code_warntype`, quindi abbiamo stabilizzato i tipi delle variabili e parallelizzando le funzioni merge_vertices e frag_face, che vengono richiamate all'interno della funzione corrente, abbiamo ottenuto un codice che si comporta come una versione più veloce del codice precedente risparmiando circa un 40% del tempo di esecuzione.

```
@benchmark Lar.Arrangement.spatial_arrangement(Points(V),ChainOp(EV),ChainOp(FE),false)

BenchmarkTools.Trial: 1551 samples with 1 evaluation.
 Range (min ... max):  2.869 ms ... 7.011 ms   GC (min ... max):  0.00% ... 52.79%
 Time  (median):     2.966 ms               GC (median):     0.00%
 Time  (mean ± ):   3.222 ms ±  846.503  s   GC (mean ± ):  6.52% ± 12.05%
```

```
  2.87 ms      Histogram: log(frequency) by time     6.47 ms <

 Memory estimate: 3.02 MiB, allocs estimate: 58808.
```

Notiamo in seguito come vengono modificati i tempi dopo l'ottimizzazione.

```
@benchmark spatial_arrangement(Points(V), ChainOp(EV), ChainOp(FE), false)

BenchmarkTools.Trial: 2612 samples with 1 evaluation.
 Range (min ... max):  1.661 ms ... 6.831 ms   GC (min ... max):  0.00% ... 60.79%
 Time  (median):     1.732 ms               GC (median):     0.00%
```

```
Time  (mean ± ):   1.911 ms ±  668.375  s   GC (mean ± ):  6.28% ± 11.05%

  1.66 ms      Histogram: log(frequency) by time      5.26 ms <

 Memory estimate: 1.59 MiB, allocs estimate: 28956.
```

Azioni Github

Grazie all'utilizzo del libro *Hands-On Julia Programming*, in particolare il capitolo 13, si sono costruite diverse `Actions` di Github, le quali eseguono delle istruzioni specifiche quando Github rileva gli eventi di attivazione corrispondenti. Ogni volta che c'è un nuovo `push` sul branch `master`, viene effettuata una simulazione per verificare che il modulo `TGW3D.jl` venga correttamente aggiunto sui sistemi operativi Ubuntu (x86 e x64), Windows (x86 e x64) e macOS (x64). Inoltre, tramite la libreria `Documenter.jl`, viene creata la documentazione del nostro progetto, tramite i `docstrings` presenti nel modulo TGW3D, ed è inserita sulla pagina di Github corrispondente.