# MPI

Innanzitutto, la generazione dei pattern non viene considerata nel calcolo del tempo necessario per l'esecuzione del programma, non è quindi parte del programma da parallelizzare.

Nella nostra implementazione parallela, i rank non si distribuiranno la porzione di sequenza su cui ricercare i pattern, bensì si distribuiranno il numero di pattern da ricercare, a tal proposito:

- In tal modo risulta più semplice la ricerca dei pattern, nel caso in cui due rank si dividessero due differenti porzioni di sequenza, un pattern che si trova a cavallo fra le due porzioni richiederebbe comunicazione fra i due rank.
- Ogni rank in tal modo dovrà salvare la totalità della sequenza, il rank zero si occuperà di generarla, per poi inviarla in broadcast ad ogni altro rank, alternativamente si potrebbe far si che ogni rank generi la sequenza.

```
if(rank==0){
    generate_rng_sequence(&random, prob_G, prob_C, prob_A, sequence, seq_length);
}
// Broadcast per condividere con tutti i rank la sequenza completa
MPI_Bcast(sequence,seq_length,MPI_CHAR,0,MPI_COMM_WORLD);
```

A seguito di ciò ogni rank avrà nella sua memoria l'intera sequenza `sequence`, dovrà iniziare la ricerca dei pattern.

```
int pat_per_thread = ceil((float)pat_number / num_thread);
int pat_start = rank * (pat_per_thread);
int pat_end = pat_start + pat_per_thread + 1 < pat_number ? pat_start + pat_per_thread : pat_number;
```

In tal modo ogni rank calcola:

- il numero di pattern che dovrà cercare all'interno della sequenza
- l'indice del primo pattern che deve cercare
- l'indice dell'ultimo pattern che deve cercare

Chiaramente l'ultimo rank potrebbe dover cercare meno pattern se il numero totale di questi ultimi non è divisibile per il numero di rank.

Appunto : Se i pattern sampled sono tutti prima dei pattern generati casualmente, i processi con numero di rank minore potrebbero impiegare meno tempo rispetto a quelli con numero di rank maggiore ad eseguire la ricerca, non è per ora noto se ciò possa influenzare significativamente la distribuzione del carico di lavoro.

Il ciclo for di ricerca per ogni rank è il seguente

```
for (pat = pat_start; pat < pat_end; pat++)
{
```

Ciò che c'è all'interno del for è rimasto invariato rispetto la versione sequenziale.

Ogni rank avrà la sua copia privata di

- `pat_matches`: la variabile in cui è contenuto il numero di pattern trovati nella sequenza
- `pat_found`: Un array lungo quanto il numero di pattern, in ogni posizione i-esima, è contenuto l'indice in cui la prima occorrenza dell'i-esimo pattern è stato trovato all'interno della sequenza. Se non è stato trovato, ci sarà -1. Chiaramente la parte di questo array interessata al rank riguarda solo i pattern che lui sta ricercando, nelle posizioni dedicate agli altri pattern, ci sarà uno zero, in modo da rendere naturale l'operazione di reduce.
- `seq_matches`: Un array lungo quanto la sequenza, nell'i-esima posizione è presente il numero di pattern che hanno toccate tale posizione, se in quella posizione non è presente alcun pattern, ci sarà 0.

Appunto riguardo `seq_matches`: Nella versione originale, le posizioni in cui non ci sono pattern trovati sono uguali a -1. Nella nostra versione, abbiamo sostituito il -1 con lo 0 per facilitare l'operazione di Reduce, e per correggere un (da noi sospetto) errore logico nel codice.

In seguito al riempimento delle 3 variabili sopra elencate, ogni rank avrà una porzione del risultato finale, sarà sufficiente eseguire una `MPI_Reduce` fra tutti i rank, in modo tale che il rank zero avrà il risultato finale con cui calcolare il checksum.