Chris Bielow

Institut für Informatik

Praktikum: Algorithmen und Datenstrukturen in der Bioinformatik
4. Programmieraufgabe

Abgabe Do, 7.01. via GIT um 23:59 Uhr

P1: Wordliste (10 Punkte)

In der Vorlesung wurde/wird der Algorithmus BLAST zum Auffinden lokaler Ähnlichkeiten zwischen einer Sequenz und einer Datenbank besprochen. Implementieren Sie das Generieren der Nachbarschaft eines Wortes der Länge w für alle Wörter/Infixe einer gegebenen Sequenz. Ihr Programm (aufgabe4_main.cpp) bekommt folgende Kommandozeilenargumente mit übergeben.

a) `<'SEQ>'` - Eine Aminosäurensequenz.

b) `<SCORE_MATRIX>` - Eine Datei mit der Scoring-Matrix, z.B. Blosum62. (Hinweis: Funktionen zum Lesen und Arbeiten mit den Matritzen werden gestellt.)

c) `<WORD_SIZE>` - Die Länge der zu generierenden Wörter.

d) `<THRESHOLD>` - Der Grenzwert für welchen ein Nachbarwort mit in die Liste übernommen wird.

e) `<THREADS>` - Die Anzahl der Threads für die parallele Verarbeitung.

Ihr Programm muss also genau 5 Argumente entgegennehmen.

Im ZIP file finden Sie verschiedene Scoring-Matritzen sowie eine Headerdatei "a4_util.h" und die Implementierung "a4_util.cpp". Sie können diese Headerdatei mittels `#include ''a4_util.h''` in Ihrem Programm einbinden. In dieser Headerdatei stehen Funktionen bereit um die Scoring-Matritzen einzulesen und um die Scoring-Matrix zu benutzen. Beim Compilieren müssen sie entsprechend "a4_util.cpp"mit angeben.

Implementieren Sie das Interface, welches in `BLAST_Neighborhood.hpp` gegeben ist, in einer `BLAST_Neighborhood.cpp`. Die Funktion *generateNeighborhood(...)* iteriert über jedes w-Wort der gegebenen Sequenz. Für jedes w-Wort wird die Nachbarschaft gegeben der Scoring-Matrix und des Thresholds berechnet. Achten Sie darauf, dass die zurückgegebenen Wörter der Reihenfolge in der Sequenz entsprechen. Also in der gleichen Reihenfolge wie die sequentielle Abarbeitung ergeben würde. Die Liste mit Nachbarwörtern zu jedem Wort soll alphabetisch sortiert sein.
Achten Sie darauf, dass in `BLAST_Neighborhood.cpp` KEINE Textausgabe auf der Konsole erfolgt!
Wenn Sie die Zusatzaufgabe (siehe unten) nicht implementieren, ist der einzig zulaessige

Wert für Threads in *generateNeighborhood(...)* gleich *1*. Für alle anderen Werte soll von *generateNeighborhood(...)* eine Exception geworfen werden.
Auch Wortgrösse muss immer >=1 sein. Wird etwas anderes an *generateNeighborhood(...)* übergeben, soll auch hier eine Exception geworfen werden. Fangen sie in *main()* keine dieser Exceptions ab (d.h. Exceptions beenden automatisch das Programm).

Geben Sie die Ergebnisse in der Main Methode in aufgabe4_main.cpp aus. Dabei wird für jedes Wort der Sequenz in einer neuen Zeile das Wort selber und dann durch ein Doppelpunkt getrennt die Liste der Nachbarn zu diesem Wort ausgegeben. Die Nachbarn werden als Tupel dargestellt, wobei das erste Element das Nachbarwort ist und das zweite Element der entsprechende Score für diesen Nachbarn.

Beispiel

```
$ ./aufgabe4_main AAHILNMY blosum62 3 14 1
AAH: (AAH, 16)
AHI: (AHI, 16) (AHL, 14) (AHV, 15)
HIL: (HII, 14) (HIL, 16) (HIM, 14) (HLL, 14) (HVL, 15)
ILN: (ILN, 14)
LNM: (LNM, 15)
NMY: (NIY, 14) (NLY, 15) (NMF, 14) (NMY, 18) (NVY, 14)
time: 0.13s
```

Messen Sie die Zeit die Ihr Programm benötigt um die Liste der Nachbarn zu generieren (d.h. nur die Zeit für den Aufruf der Funktion *generateNeighborhood(...)* in *main()*). Nutzen Sie dafür die Funktion *omp_get_wtime()* aus der OpenMP Bibliothek (in *omp.h*; auch wenn sie die Zusatzaufgabe nicht implementieren). Geben Sie die Zeit (Wall-Time) am Ende mit aus (siehe obiges Beispiel!).

Checken Sie `aufgabe4_main.cpp, BLAST_Neighborhood.hpp` und `BLAST_Neighborhood.cpp` im Unterverzeichnis `./aufgabe4/` ins GIT ein.

Praktikumshinweise - Randfälle

- w > |q|, produziert leeren Ausgabevector in *generateNeighborhood(...)*.
- Nicht vorhandene Datei fúr ScoringMatrix produziert Programmabbruch in main()
- bei sehr hohem Score-Threshold könnte die Nachbarschaft leer sein
- Infixe und deren Neighborhood müssen in der richtigen Reihenfolge aus *generate-Neighborhood(...)* zurückgegeben werden (siehe Dokumentation in .hpp).
- Compilieren sie ihr Programm mithilfe des Makefiles auf dem Poolrechner um Compile-Errors (und damit 0 Punkte) auszuschliessen

Zusatzaufgabe: parallelisierte Wordliste (4 Punkte)

Berechnen Sie die Neighborhood mit Hilfe von OpenMP mit mehreren Threads parallel. Achten Sie darauf, dass ihr Programm weiterhin korrekt arbeitet (vergleichen Sie die Ergebnisse für 1 vs. n Threads).
Auf der Eingabe `AAHILNA blosum62 4 20 <Threads>` sollte ihr paralleles Programm mit 2-4 Threads schneller laufen als die serielle Version (1 Thread). Stellen Sie sicher, dass dem so ist.

Praktikumshinweise

- Schreiben Sie das Programm erst seriell. Erst wenn es korrekt funktioniert (ausgiebig testen), fügen Sie die OpenMP Konstrukte hinzu. Dazu ist es jedoch hilfreich wenn Sie beim Design darauf achten, was Sie später parallelisieren wollen.
- Achten Sie darauf dem Compiler die OpenMP flags mitzugeben damit OpenMP auch aktiviert wird. Bei Visual Studio muss OpenMP für Debug und Release Modus getrennt aktiviert werden!
- Zulässige Werte für Threads sind jetzt alle positiven Zahlen (d.h. passen Sie ihre Exception von oben entsprechend an).