Praktikum Algorithm Enginnering:
Multicore-Programmierung mit der (MC)STL

Peter Sanders, Johannes Singler SS 2007

http://algo2.iti.uni-karlsruhe.de/mcstl-praktikum.php

Arbeitsblatt 2

Abgabe: 21. Mai 2007

Aufgabe 1: Integration von `set_union` in die MCSTL (Abgabe)

Integrieren Sie Ihre Implementation von `set_union` vom letzten Arbeitsblatt in die aktuelle MCSTL, Version 0.7.3-beta, von der Website.

- Ergänzen Sie passende Zeilen in `mcstl_define.h` und `mcstl_undefine.h`, um die Original-Implementierung zu verstecken, aber doch greifbar zu halten.
- Fügen Sie den (High-Level-) Code von `set_union` in den Header `algorithm` ein. Erzeugen Sie mittels Template-Spezialisierung [1] eine Entscheidung zur Compilezeit, die nur Ihren neuen Code aufruft, wenn alle übergebenen Iteratoren wahlfreien Zugriff erlauben. Es ergeben sich acht Möglichkeiten. Vermeiden Sie eine weitere Verdopplung der Anzahl Varianten, indem Sie die Signatur ohne explizitem Komparator möglichst direkt auf die Signatur mit dem expliziten Komparator `std::less<T>()` abbilden. Ein bereits vorhandenes Beispiel ist `mismatch`.

  Freiwillige Zusatz-Aufgabe: Wie lässt diese Art Fallunterscheidung für viele Funktionen zusammenfassen?
- Fügen Sie eine analoge Fallunterscheidung für Ihren `unique_copy`-Code in den Header `algorithm` ein. Der parallel Code wird dann aufgerufen, wenn alle Bedingungen dazu erfüllt sind, wiederum analog zu `mismatch`. Führen Sie zu diesem Zwecke eine neue Tuning-Variable `unique_copy_minimal_n` ein.
- Der eigentliche Code für `unique_copy` kommt in eine neue Datei `mcstl_unique.h` im Unterordner `bits`. Fügen Sie bitte den passenden Lizenz-Header ein, und dokumentieren Sie ihr Copyright.
- Prüfen Sie die Verbindung aller dieser Komponenten und schreiben Sie ein Programm, dass `set_union` mit verschiedensten Eingaben instanziiert. Überprüfen Sie mittels eines Debuggers oder einer Debug-Ausgabe in `set_union`, ob jeweils die gewünschte Varianten aufgerufen wird oder nicht.

Aufgabe 2: Weitere Mengen-Operationen (Abgabe)

Die STL enthält vier Algorithmen für Mengen-Operationen, nämlich `set_union` (Vereinigung), `set_intersection` (Schnitt), `set_difference` (Differenz) und `set_symmetric_difference` (symmetrische Differenz, $(A \cup B) \setminus (A \cap B)$). Gemäß der Spezifikation der Mengen-Algorithmen liegen die Eingabe-Mengen als sortierte Sequenzen vor. Jede Sequenz repräsentiert eine echte Menge, enthält also keine gleichen Elemente. Zusätzlich können Sie annehmen, dass die Iteratoren wahlfreien Zugriff erlauben (Random Access).

- Beweisen Sie für für jedes $\circ$ aus den vier Operationen:

$$A \circ B = ((A \cap U_1) \circ (B \cap U_1)) \dot\cup \ldots \dot\cup ((A \cap U_n) \circ (B \cap U_n)) \tag{1}$$

  wobei $U = U_1 \dot\cup \ldots \dot\cup U_n$ das Universum der Elemente ist, aus denen die Menge gebildet werden. Wie lässt sich daraus eine mögliche Parallelisierung ableiten?

- Implementieren Sie parallele Varianten aller vier Algorithmen, basierend auf einem gemeinsamen Grundalgorithmus, einer Verallgemeinerung von `merge`:

  1. Teile jede Sequenz in p Stücke, sodass die Gesamtlängen der *Scheiben* (alle i-ten Stücke zusammengenommen) gleich sind (plus minus 1). Dazu kann die Funktion `multiseq_partition` verwendet werden, die bereits in der MCSTL vorhanden ist. (Nützliche Eigenschaft: Diese nimmt bevorzugt Elemente aus Sequenzen mit kleinerer Nummer am Ende hinzu, wenn es viele gleiche über einer Schnittstelle hinweg gibt.) Sie soll p-fach parallel ausgeführt werden, um p Stücke zu erzeugen.
  2. Führe die gewünschte Mengenoperation parallel auf den Scheiben durch, schreibe jedoch kein Ergebnis, sondern zähle nur die Anzahl Elemente im Ergebnis.
  3. Bestimme die Positionen, an denen die Ausgabe jedes Threads stehen muss, z. B. mittels `partial_sum`.
  4. Führe die gewünschte Mengenoperation nochmals parallel aus, schreibe die Ergebnisse direkt an die passenden Stellen.

- Schreiben Sie ein Programm, das die Funktion für beliebige Sequenzen und Anzahlen von Threads testet und dabei die Laufzeit misst. Verwenden Sie dieses Mal eine Klasse für komplexe Zahlen, mit den Komponenten a und b als `double`. Die Eingabedaten sollen zufällig, gleichverteilt im achsenparallelen Quadrat von $[-1, -1]$ und $[1, 1]$ erzeugt werden, und vorsortiert werden. Einmal wird per Funktor nach Winkel verglichen (`arctan2(b, a)`), standardmäßig der Betrag (`sqrt(a*a + b*b)`). Die Korrektheit des Ergebnisses muss automatisch überprüft werden, d. h. mit dem Ergebnis des sequentiellen Algorithmus verglichen werden. Dokumentieren Sie die Speedups für bis zu 8 Threads auf einem entsprechenden Rechner, für Eingabegrößen von 1 bis 100.000.000, in Schritten von $\sqrt{10}$. Experimentieren Sie auch mit dem Fall, dass beide Eingabemengen nicht gleich groß sind. Fassen Sie die Ergebnisse sinnvoll zusammen; erstellen Sie ein Schaubild für jede der Operationen, das den Speedup über der Eingabegröße zeigt.

Hinweise:

- Es gelten weiterhin die Hinweise von Arbeitsblatt 1. Binden Sie jedoch die Funktionen gleich in die MCSTL ein, ebenfalls mit der statischen Entscheidung über die Iteratoren.
- Vermeiden Sie Code-Duplikation, die den Code schwer wartbar machen würden. Der Grundalgorithmus soll nur einmal geschrieben und für die vier Fälle über Templates parametrisiert werden.
- Testen Sie auch Randfälle und Spezialfälle ab, z. B. nur gleiche Elemente, nur gleiche Elemente pro Sequenz, (teilweise) leere Eingabe. Begründen Sie bei jedem dieser Fälle, wieso das Programm (trotzdem) korrekt arbeitet.
- Die SGI-STL sichert bei diesen Operationen zusätzlich Stabilität zu, d. h. von zwei gleichen Elemente wird gegebenenfalls das aus der ersten Sequenz ins Ergebnis übernommen.

Dokumentieren Sie ihre Lösung sowie die Messergebnisse schriftlich. Geben Sie dieses Dokument ausgedruckt ab, und schicken Sie dem Betreuer dieses als Datei sowie die Quelltexte zur zweiten Aufgabe, als Teil der erweiterten MCSTL-Distribution.

Literatur

[1] Wikipedia-Artikel Template-Spezialisierung. `http://de.wikipedia.org/wiki/Template_(Programmierung)#Spezialisierung_bei_Klassentemplates`.