Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Daniel Lorenz, lorenz@cs.tu-darmstadt.de

Systemnahe und parallele Programmierung (WS 16/17)

Praktikum: C und OpenMP

Die Lösungen müssen bis zum 13. Dezember 2016 in Moodle submittiert werden. Anschließend müssen sie ihre Lösung einem Tutor vorführen. Das Praktikum wird benotet. Im Folgenden einige allgemeine Bemerkungen, die für alle Aufgaben auf diesem Blatt gelten:

- Für die zu implementierenden Module werden Header-Dateien bereitgestellt. Diese Header-Dateien dürfen nicht modifiziert werden. Die dort definierten Funktionen und Strukturen sollen in einer Quelldatei mit dem selben Basis-Namen implementiert werden. Also z.B. soll der Inhalt von `list.h` in einer Datei namens `list.c` implementiert werden.
- Wenn eine Teilaufgabe das Erstellen eines Programms erfordert, muss auch immer ein `Makefile` erstellt werden, dass dieses Programm baut. Ob je ein `Makefile` pro Teilaufgabe oder nur Eines mit mehreren Regeln für jede Teilaufgabe erstellt werden, bleibt Ihnen berlassen.
- Oft bauen folgende Aufgaben auf vorherigen Aufgaben auf und erfordern nur eine Modifikation eines früheren Programms. In diesen Fällen soll auch die frühere Version des Programm von der vorherigen Aufgabe erhalten werden.
- Dynamisch allokierter Speicher muss frei gegeben werden bevor das Programm endet.
- Es folgt eine Liste von Funktionen der C Standardbibliothek, die bei der Lösung hilfreich sein könnten. Bitte informieren Sie sich im Internet über die genaue Funktion und Verwendung dieser Funktionen.
  - `isalpha()`
  - `strdup()`
  - `strncpy()`
  - `strrchr()`
  - `toupper()`
- Zum Testen der Lösung, insbesondere für die Zeitmessungen, benötigen sie große Textdateien. Diese können z.B. vom Gutenberg-Projekt (`www.gutenberg.net`) kostenlos heruntergeladen werden. Achten sie darauf, die `.txt` Version der Texte herunterzuladen. Um nennenswerte Laufzeiten messen zu können, sollten sie Texte mit mindestens 1 MB Größe verwenden. z.B.
  - `http://www.gutenberg.org/files/82/82-0.txt`
  - `http://www.gutenberg.org/cache/epub/30/pg30.txt`
  - `http://www.gutenberg.org/cache/epub/2145/pg2145.txt`

Aufgabe 1

Im ersten Aufgabenblock soll eine Baum-basiertes Wörterbuch implementiert werden. Das Wörterbuch soll durch das Parsen eines Textes gefüllt werden. Anschließend soll eine zweite Textdatei eingelesen werden und alle Wörter gezählt werden, die nicht im Wörterbuch enthalten sind. In den folgenden Aufgaben werden dann Teile des Programms parallelisiert.

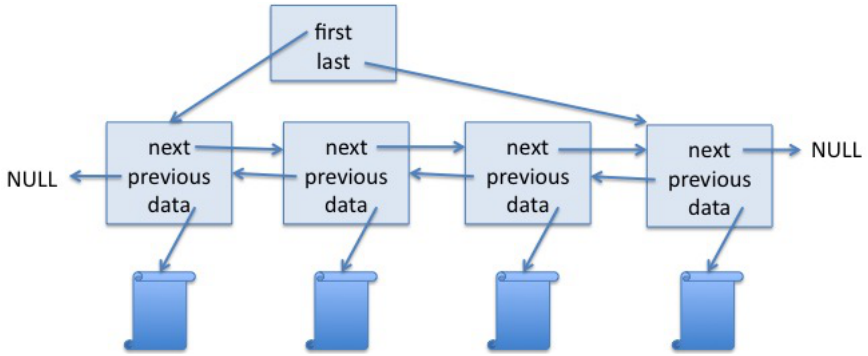

Figure 1: Eine doppelt verlinkte Liste.

a) (12 Punkte) Als ersten Schritt soll eine doppelt verlinkte Liste implementiert werden, die einen `char*` als Datenelement speichert. Eine doppelt verlinkte Liste ist eine Datenstruktur, bei der jedes Element einen Zeiger auf das folgende und das vorhergehende Element speichert. Der Zeiger zum nächsten Element des letzten Listenelements und der Zeiger zum vorhergehenden Element der ersten Elements sind `NULL`. Eine Beispiel ist in Abbildung 1 zu sehen.

Für die Implementierung der doppelt verlinkten Liste sollen zwei Datenstrukturen definiert werden:

- `LinkedListNode`
  Diese Struktur repräsentiert ein Listenelement. Es enthält einen Zeiger auf das vorhergehende und das nächste Listenelement und einen `char*` Zeiger auf die Daten.
- `LinkedList`
  Diese Struktur repräsentiert die gesamte Liste und enthält je einen Zeiger auf der erste und auf das letzte Element der Liste. Ist die Liste leer, sind beide Zeiger `NULL`.

Des Weiteren sollen Funktionen zum Zugriff und Bearbeiten der Liste erstellt werden. Die entsprechenden Funktionen sind in der Datei `list.h` deklariert.

- `LinkedList* LinkedList_create()`
  Diese Funktion gibt eine neue doppelt verlinkte Liste zurück. Sie allokiert den nötigen Speicher und initialisiert alle Felder mit `NULL`.
- `void LinkedList_append( LinkedList* list, char* data )`
  Diese Funktion erzeugt ein neues Listenelement, dass den Zeiger `data` speichert, und hängt das neue Listenelement am Ende der Liste an.
- `void LinkedList_delete( LinkedList* list )`
  Diese Funktion zerstört eine doppelt verlinkte Liste. Sie gibt auch den Speicher aller enthaltenen Listenelemente und der Datenelemente frei.
- `LinkedListNode* LinkedList_getFirst( LinkedList* list )`
  Diese Funktion gibt einen Zeiger auf das erste Element der Liste zurück. Ist die Liste leer, wird `NULL` zurückgegeben.
- `LinkedListNode* LinkedList_getLast( LinkedList* list )`
  Diese Funktion gibt einen Zeiger auf das letzte Element der Liste zurück. Ist die Liste leer, wird `NULL` zurückgegeben.
- `LinkedListNode* LinkedList_getPrevious( LinkedListNode* node )`
  Diese Funktion gibt einen Zeiger auf das vorhergehende Element der Liste zurück. Ist `node` das erste Element der Liste, wird `NULL` zurückgegeben.
- `LinkedListNode* LinkedList_getNext( LinkedListNode* node )`
  Diese Funktion gibt einen Zeiger auf das nächste Element der Liste zurück. Ist `node` das letzte Element der Liste, wird `NULL` zurückgegeben.

- `char* LinkedList_getData( LinkedListNode* node )`
  Diese Funktion gibt den in `node` gespeicherten Zeiger zurück.
- `unsigned int LinkedList_getSize( LinkedList* list )`
  Diese Funktion gibt die Anzahl der Listenelement zurück.
- `char* LinkedList_getDataAt( LinkedList* list, unsigned int index )`
  Diese Funktion gibt den gespeicherten Zeiger aus dem Element an der Position `index` zurück. Das erste Element der Liste hat die Position 0.

b) (5 Punkte) Im zweiten Schritt soll ein Programm erstellt werden, dass eine Textdatei öffnet und einliest. Der Text soll in 16 kB großen Blöcken in der doppelt verketteten Liste aus Aufgabe 1a gespeichert werden. Jeder dieser Textblöcke soll nur vollständige Wörter enthalten. Wenn ein Wort nicht mehr vollständig in den 16 kB Block hineinpasst, soll das gesamte Wort im nächsten Textblock gespeichert werden. Die Aufteilung des Textes in mehrere Blöcke wird später bei der Parallelisierung benötigt. Der Name der Textdatei soll als Programmargument übergeben werden.

Die Datei `file_reader.h` enthält die Deklaration der Funktion, die zum Öffnen und Einlesen implementiert werden soll. Die `main()`-Funktion soll in einer anderen Quelldatei implementiert werden.

- `LinkedList* read_text_file( const char* filename, int blockSize )`
  Diese Funktion öffnet und liest eine Textdatei, deren Name im Parameter `filename` angegeben ist. `blockSize` spezifiziert die maximale Größe der Textblöcke. Die Funktion gibt eine doppelt verlinkte Liste zurück, die den Text enthält.

c) (6 Punkte) In dieser Teilaufgabe soll ein Textparser implementiert werden, der in einem Textblock einzelne Wörter identifiziert. Die dafür benötigten Datenstrukturen und Funktionen sind in der Datei `parser.h` deklariert. Die Idee ist, dass zunächst ein Parserstruktur erstellt wird, das die aktuelle Position in dem zu parsenden Text speichert. Mit Hilfe des Parser-Objektes kann dann das jeweils nächste Wort identifiziert und zurückgegeben werden.

Um das Parsen zu vereinfachen nehmen wir an, dass Wörter nur aus den Buchstaben A-Z bestehen. Alle anderen Zeichen, inklusive Umlaute, ß und andere Sonderzeichen trennen zwei Wörter. Weiterhin soll der Parser die Wörter in Großbuchstaben zurückgeben, um die Weiterverarbeitung im Wörterbuch zu vereinfachen.

- `Parser`
  Dies ist die Datenstruktur, die die aktuelle Position im zu parsenden Text speichert.
- `Parser* Parser_create( const char* text )`
  Diese Funktion erzeugt ein neues Parserobjekt. Die Position ist am Anfang des Textes. Der Text darf sich bis zum Ende des Parsens nicht verändern.
- `int Parser_getNextWord( Parser* parser, char* nextWord, unsigned int bufferLength )`
  Diese Funktion schreibt das nächste Wort an die durch `nextWord` angegebene Speicherstelle. Es werden maximal `bufferLength` Zeichen geschrieben. Der Nutzer ist dafür verantwortlich, dass genügend Speicher allokiert wurde und `nextWord` auf eine gültige Speicherstelle zeigt. Wenn das Ende des Textes erreicht wurde, gibt die Funktion 0 zurück. Ansonsten ist der Rückgabewert ungleich 0. Alle Buchstaben in `nextWord` sind in Großbuchstaben umgewandelt.
- `void Parser_delete( Parser* parser )`
  Diese Funktion zerstört ein Parserobjekt.

d) (12 Punkte) In dieser Teilaufgabe soll das Baum-basierte Wörterbuch implementiert werden. Hierbei handelt es sich um einen Baum, bei dem jeder Knoten eine Zeichenkette speichert, die der längste gemeinsame Prefix für alle in diesem Subbaum gespeicherten Wörter ist. Die Kindknoten speichern dann eine Zeichenkette, die sich im nächsten Zeichen von den anderen Kindknoten unterscheidet. Nehmen wir zum Beispiel an, wir haben ein Wörterbuch mit den Wörtern `WHALE`,

`WOOD` and `WOODFIRE`. Dann würde das Wörterbuch die in Abbildung 2 gezeigte Struktur haben. Die Wurzel speichert die Zeichenkette `W`, da dies der gemeinsame Prefix aller 3 Wörter ist. Die Wurzel hat zwei Kindknoten, die eine Zeichenkette speichern, die sich im 2. Buchstaben unterscheidet. Ferner muss bei jedem Knoten gespeichert werden, ob es sich bei der Zeichenkette um ein Wort im Wörterbuch handelt. Zum Beispiel ist `WOOD` ein Wort auch wenn es Kindknoten hat, `W` aber nicht.

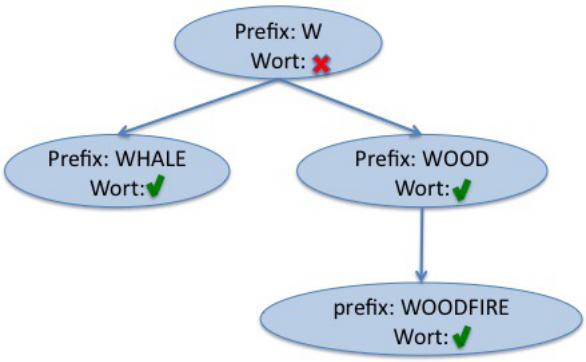

Figure 2: Beispiel für ein Baum-basiertes Wörterbuch mit den drei Worten WHALE, WOOD und WOODFIRE.

Um ein neues Wort dem Wörterbuch hinzuzufügen, muss der längste gemeinsame Prefix gefunden werden. Das neue Wort wird dann ein Kindknoten des gemeinsamen Prefix. Existiert der Prefix noch nicht als eigenständiger Knoten, muss dieser auch eingefügt werden. Wenn dem Wörterbuch zum Beispiel das Wort `WOLF` hinzugefügt wird, ist der längste gemeinsame Prefix `WO`. Daher muss zunächst ein Knoten mit der Zeichenkette `WO` zwischen `W` und `WOOD` eingefügt werden. Dann kann `WOLF` ein neues Kind von `WO` werden. Abbildung 3 zeigt die neue Struktur.

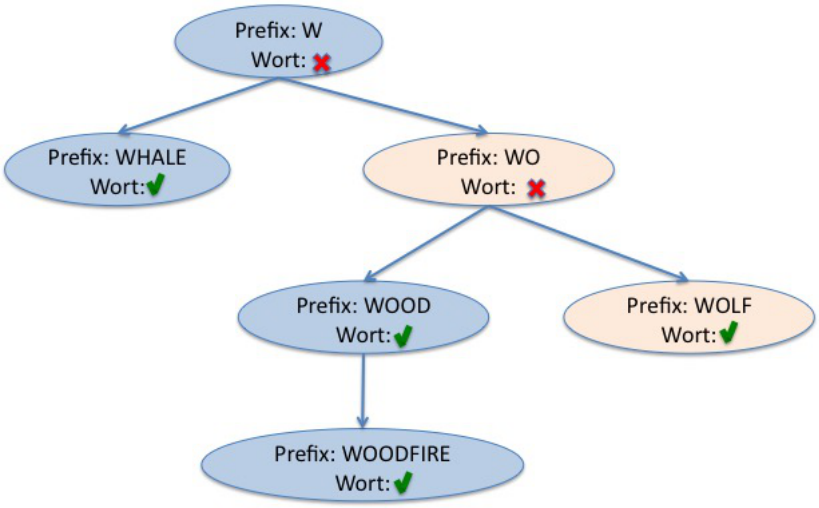

Figure 3: Fortsetzung des Beispiels aus Abbildung 2. Das Wörterbuch nachdem das Wort WOLF eingefügt wurde.

Die Datei `dictionary.h` enthält die Deklaration der zu implementierenden Funktionen. Beachten Sie, dass sie vermutlich weitere lokale Hilfsfunktionen benötigen, die aber von Funktionen in anderen Quelldateien nicht aufgerufen werden sollen. Diese Hilfsfunktionen können Sie in ihrer Quelldatei selbst definieren.

- `Dictionary`
  Dieser Datentyp repräsentiert ein Wörterbuch.
- `Dictionary* Dictionary_create()`
  Diese Funktion erzeugt ein neues leeres Wörterbuch.
- `void Dictionary_delete( Dictionary* dict )`

Diese Funktion zerstört ein Wörterbuch und gibt auch alle enthaltenen Knoten und Zeichenketten frei.

- `void Dictionary_insert( Dictionary* dict, const char* word )`
  Diese Funktion fügt ein Wort in das Wörterbuch ein.
- `int Dictionary_isIn( Dictionary* dict, const char* word )`
  Diese Funktion überprüft ob ein Wort im Wörterbuch enthalten ist. Ist das Wort enthalten wird ein Wert ungleich 0 zurückgegeben. Ansonsten wird 0 zurückgegeben.
- `void Dictionary_print( Dictionary* dict )`
  Diese Funktion gibt alle im Wörterbuch gespeicherten Wörter in alphabetischer Reihenfolge auf dem Bildschirm aus.
- `void Dictionary_merge( Dictionary* destination, Dictionary* source )`
  Diese Funktion fügt alle in `source` enthaltenen Wörter in `destination` ein.

Erstellen Sie auch ein Programm, dass eine Textdatei öffnet, den Text in einer der doppelt verlinkten Liste speichert und anschließend parst, um das Wörterbuch aufzubauen. Zum Schluss sollen allen Wörter im Wörterbuch in alphabetischer Reihenfolge auf dem Bildschirm ausgegeben werden.

e) (5 Punkte) Im letzten Schritt dieser Teilaufgabe soll ein Programm erstellt werden, dem zwei Textdateien als Programmargumente übergeben werden sollen. Aus der ersten Textdatei soll ein Wörterbuch aufgebaut werden, wie in der vorherigen Teilaufgabe beschrieben. Danach soll die zweite Textdatei eingelesen werden und ebenfalls in einer doppelt verlinkten Liste gespeichert werden. Anschließend sollen alle Wörter aus der zweiten Textdatei gezählt werden, die nicht im Wörterbuch enthalten sind. Diese Zahl soll auf dem Bildschirm ausgegeben werden.

Aufgabe 2

In dieser Aufgabe soll das Zählen der Wörter, die nicht im Wörterbuch sind, mit verschiedenen Ansätzen parallelisiert werden. Als Ausgangspunkt kann dafür das Programm aus 1e verwendet werden. Behalten sie eine Kopie des Programms aus 1e und jeder parallelisierten Version.
Für die Zeitmessungen kann die Funktion `omp_get_wtime()` verwendet werden. Ein Programm mit einer anderen Anzahl von Threads laufen zu lassen, sollte keine Modifikation des Quellcodes erfordern.
Um ein Programm mit OpenMP zu kompilieren muss die OpenMP-Unterstützung explizit aktiviert werden. Beim `gcc` muss dazu `-fopenmp` als Programmparameter angegeben werden.

a) (6 Punkte) Zuerst soll das Zählen der Worte, die nicht im Wörterbuch enthalten sind, mit einer parallelen `for`-Schleife parallelisiert werden. Jede Schleifeniteration soll einen der Textblöcke verarbeiten. Dadurch verarbeitet also jeder Thread einen Teil der Textblöcke in der doppelt verlinkten Liste. Der Zugriff auf eine gemeinsame Zählervariable verursacht einen Datenkonflikt. Dieser wird aufgelöst, indem jeder Thread einen privaten Zähler besitzt, und am Ende das Gesamtergebnis mit einer Reduktion ermittelt wird.

b) (4 Punkte) Messen sie die Laufzeit der parallelen Region aus Aufgabe 2a. Die gemessene Zeit soll die Zeit zum Laden der Textdateien oder dem Aufbau des Wörterbuchs nicht mit einschließen, sondern nur die Zeit zum Zählen der Worte, die nicht im Wörterbuch vorkommen.

Das Programm aus 2a soll mit 1, 2, 4, 8 und 16 Threads ausgeführt werden. Stellen Sie die Ergebnisse graphisch dar und vergleichen sie die gemessenen Zeiten.

c) (10 Punkte) Nun soll die Zählung der Worte, die nicht im Wörterbuch enthalten sind, mit OpenMP Tasks anstatt mit einer `for`-Schleife parallelisiert werden. Dabei soll jeder Task einen Textblock verarbeiten. Ein Thread muss dazu über die verkettete Liste iterieren und für jeden Textblock einen Task erzeugen. Die Tasks werden dann von allen Threads bearbeitet. Beachten Sie, dass es zu Datenkonflikten führt wenn mehrere Threads denselben Zähler inkrementieren. Daher sollte jeder Thread einen privaten Zähler besitzen. Nachdem alle Tasks bearbeitet wurden, kann jeder Thread seinen privaten Zähler zu dem Gesamtergebnis addieren. Der Zugriff auf die Variable mit dem Gesamtergebnis sollte aber durch eine `atomic`-Direktive geschützt werden.

Messen Sie die Ausführeungszeit der Parallelen Region für 1, 2, 4, 8 und 16 Threads. Stellen Sie die Ergebnisse graphisch dar.

d) (10 Punkte) In dieser Teilaufgabe soll eine feingranulare Taskparallelisierung der Zählung der Worte, die nicht im Wörterbuch stehen, implementiert werden. Anstatt einen Textblock zu bearbeiten, soll für jedes Wort ein eigener Task erstellt werden. Ein Thread soll also den gesamten Text parsen und für jedes Wort einen Task erstellen, der überprüft ob dieses Wort im Wörterbuch steht.

Messen sie sie Ausführungszeit der parallelen Region mit 1, 2, 4, 8 and 16 Threads. Stellen Sie die Ergebnisse graphisch dar. Vergleichen Sie die Messungen mit den Messungen aus Aufgabe 2c.

Aufgabe 3

In der dritten Aufgabe soll der Aufbau des Wörterbuchs aus den Wörtern einer Textdatei parallelisiert werden. Da Änderungen an dem Wörterbuch zu Datenkonflikten führen können wenn mehrere Threads auf des Wörterbuch zugreifen, ist der Aufbaus des Wörterbuchs komplizierter zu parallelisieren.

a) (10 Punkte) Der erste Ansatz zur Parallelisierung ist eine parallele `for`-Schleife zu verwenden und in jeder Schleifeniteration einen Textblock zu bearbeiten. Um Datenkonflikte zu vermeiden soll der Zugriff auf das Wörterbuch exklusiv sein. Der Aufruf jeder Wörterbuchfunktion muss daher mit einer OpenMP `critical`-Region geschützt werden.

Messen Sie die Zeit, die zum Aufbau des Wörterbuchs benötigt wird für 1, 2, 4, 8 und 16 Threads. Die Zeitmessung sollte nicht die Zeit zum Öffnen und Lesen der Textdatei einschließen. Vergleichen Sie die Zeiten für die verschiedenen Thread-Anzahlen. Stellen Sie die Ergebnisse graphisch dar.

b) (10 Punkte) Der exklusive Zugriff auf das Wörterbuch verursacht eine Serialisierung des parallelen Programms wodurch die Parallelisierung ihren Sinn verliert. Um also den Flaschenhals beim Zugriff auf das Wörterbuch zu vermeiden, soll jeder Thread sein privates Wörterbuch aufbauen. Dann können die Textblöcke auf die verschiedenen Threads verteilt werden und die Zugriffe auf das private Wörterbuch sind konfliktfrei. Allerdings bedeutet das, dass am Ende die privaten Wörterbücher in ein gemeinsames Wörterbuch integriert werden müssen. Der Zugriff auf das gemeinsame Wörterbuch muss exklusiv sein.

Messen Sie wieder die Laufzeit der parallelen Region um das Wörterbuch aufzubauen mit 1, 2, 4, 8 und 16 Threads. Vergleichen Sie die Messungen mit den Messungen aus Aufgabe 3a. Stellen Sie die Ergebnisse graphisch dar.

c) (10 Punkte) Als letztes soll jetzt noch das Zusammenführen der privaten Wörterbücher aus Aufgabe 3b parallelisiert werden. Dazu soll ein rekursiver Algorithmus mit Tasks verwendet werden.

Der Kern des Algorithmus ist die Funktion `dict_merge`, der ein Array von n Wörterbüchern übergeben wird, die zusammengeführt werden sollen:

- Ist $n < 2$, muss die Funktion nichts machen.
- Ist n genau gleich 2, fügt die Funktion die Wörter des zweiten Wörterbuchs in das Erste ein.
- Ist $n > 2$, erzeugt die Funktion 2 Tasks. Der erste Task ruft `dict_merge` auf und übergibt die erste Hälfte der Wörterbücher. Der zweite Task ruft ebenfalls `dict_merge` auf und übergibt die zweite Hälfte der Wörterbücher. Nachdem beide Tasks ausgeführt wurden, werden alle Wörter aus dem resultierenden Wörterbuch des zweiten Tasks in das Ergebnis des ersten Tasks eingefügt.

Abbildung 4 veranschaulicht wie die Vereinigung mit 8 Threads funktioniert.

Messen Sie die Laufzeit der parallelen Region zum Aufbau des Wörterbuchs und vergleichen sie die gemessenen Zeiten mit den Ergebnissen aus Aufgabe 3b. Stellen Sie die Ergebnisse graphisch dar.

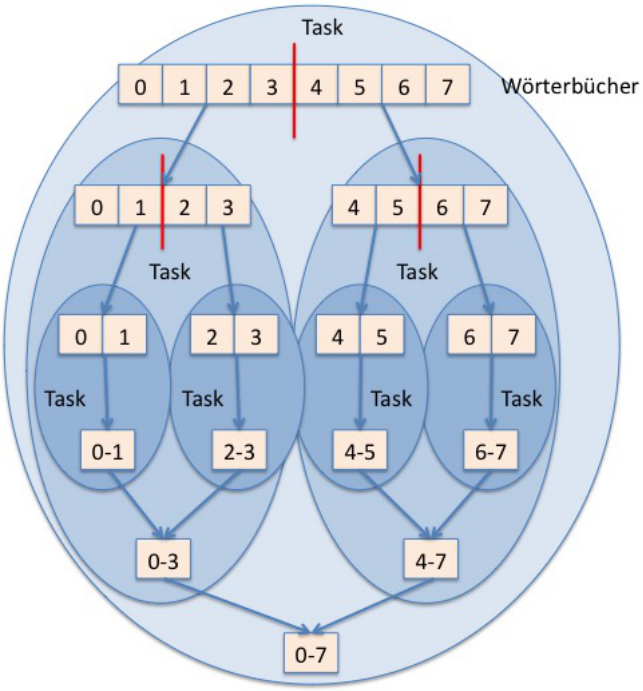

Figure 4: Rekursive Vereinigung der Wörterbücher mit Tasks.