Die Programmiersprache Cilk

David Soria Parra

2. Juli 2010

Abstract

Cilk 5 ist eine Erweiterung der Programmiersprache C zur Parallelprogrammierung. Die Sprache implementiert einen effizienten Work-Stealing-Algorithmus zur parallelen Ausführung von Funktionen auf einem System. Ich werde im Rahmen dieser Arbeit die Spracherweiterungen von Cilk besprechen und zeigen, wie Cilk den Work-Stealing-Algorithmus implementiert.

1 Einleitung

Cilk 1 wurde 1994 am MIT Laboratory for Computer Science mit dem Ziel entworfen Studenten die nötigen Datenstrukturen und Mechanismen zur Parallelisierung eines Programmes auf Shared-Memory Systemen aufzuzeigen. Cilk 2 führte high-level Anweisungen ein um den Sprachkern zu vereinfachen und die Parallelisierung transparenter zu machen.

Im weiteren Verlauf der Entwicklung von Cilk konzentrierte sich die Cilk Gruppe um Charles E. Leiserson auf die Weiterentwicklung des Work-Stealing-Algorithmus und die Einführung weiterer Anweisungen um spekulative Berechnungen zu ermöglichen. Die aktuelle Version von Cilk, Version 5.3.2, wurde 2007 veröffentlicht [CWe]. Während bei vorherigen Cilk-Versionen die Erweiterung der Sprache im Vordergrund stand, legte man bei Entwicklung von Cilk 5 Wert auf einen stabilen Sprachkern für den produktiven Einsatz.

Seit 2007 wird Cilk vom Unternehmen Cilk Inc. weiterentwickelt. Eine C++ Variante von Cilk, Cilk++, wird von Intel entwickelt, die 2009 Cilk Inc. aufkauften [CPP].

2 Die Programmiersprache Cilk

Cilk erweitert die Programmiersprache ANSI-C um die fünf Anweisungen `cilk`, `spawn`, `sync`, `inlet` und `abort` [Sup]. Diese Anweisungen kontrollieren die parallele Ausführung des Programmes. Cilk operiert ausschließlich auf Funktionen. Andere Anweisungenblöcke können nicht parallelisiert werden und müssen bei Bedarf in eine Funktion ausgelagert werden. Cilk kann ein Programm nur auf einem Rechner parallelisieren. Eine Verteilung

der Arbeit auf mehrere Rechnerknoten, wie beispielsweise in Erlang, findet nicht statt. Werden alle von Cilk eingeführten Anweisungen in einem Programm entfernt, erhält man ein valides C-Programm. Das so entstandene C-Programm wird *C-Elision* genannt.

Im Verlauf der Arbeit werde ich eine *Cilk-Funktion* kurz als Funktion bezeichnen. Eine nich parallelisierbare Funktion wird als *C-Funktion* bezeichnet. Sie folgt den Aufrufskonventionen des verwendeten C-Compilers. Eine parallel ablaufende Instanz einer Funktion wird als *Thread* bezeichnet. Ein *Thread* wird nicht auf einen System-Thread abgebildet.

2.1 cilk und spawn

Mit dem Schlüsselwort `cilk` wird eine Funktion als parallelisierbar ausgewiesen. Eine Funktion kann nicht wie eine C-Funktion aufgerufen werden, sondern muss mit dem Schlüsselwort `spawn` gestartet werden. Die Funktion wird an den Cilk Scheduler (siehe Kapitel 4.2) übergeben. Sie blockiert nicht. Die aufrufende Funktion wird als *Elternfunktion* bezeichnet. Die aufgerufene Funktion wird als *Kindfunktion* bezeichnet.

Das Schlüsselwort `spawn` darf ausschließlich in Zuweisungen der Form `=`, `+=`, `-=`, `*=`, `/=`, alleine oder als erster Parameter eines *Inlets*-Aufruf (siehe 2.3) verwendet werden. Diese Restriktion ist nötig um Datenanbhängigkeiten zwischen parallel laufenden Funktionen zu verhindern (siehe 3.1.3)[DD01].

2.2 sync

Die `sync` Schlüsselwort dient als lokale Barriere. Sie stellt sicher, dass alle durch `spawn` aufgerufenen Funktionen beendet wurden und die Rückgabewerte zur Verfügung stehen. Die Verwendung von Rückgabewerten ohne vorheriges `sync` ist unsicher. Eine abgearbeitete Funktion impliziert an ihrem Ende einen `sync`.

Beispiel 1 zeigt die parallelen Berechnung der Fibonacci-Zahlen mit Cilk. In dem Beispiel muss auf die Integer-Werte `a` und `b` mithilfe der `sync` Anweisung gewartet werden um fehlerhafte Berechnungen bei dem Rückgabe zu vermeiden.

Example 1.

```
cilk int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    int a, b;
    a = spawn fib(n-1);
    b = spawn fib(n-2);
    sync;
    return a + b;
  }
}
```

2.3 inlet

Sollen Rückgabewerte eines `spawn` Aufrufs weiterverarbeitet und der Synchronisationsaufwand vermieden werden, können *Inlets* verwendet werden. Inlets sind C-Funktionen, die ausgeführt werden, sobald der Aufruf einer Funktion beendet ist. Dafür wird der `spawn` Aufruf als erster Parameter des Inlets angegeben.

2.3.1 Definition

`Inlets` werden durch das Schlüsselwort `inlet` markiert und als C-Funktion definiert. Ein Inlet muss innerhalb einer Cilk-Funktion definiert werden und ist in deren lokalen Scope verwendbar.

Inlets sind in ihrer Semantik eingeschränkt und dürfen keine `sync`, `spawn` oder weitere Inlet-Anweisungen enthalten. Ein Inlet darf auf alle Variablen innerhalb der umschließenden Funktion zugreifen und diese verändern. Dabei stellt Cilk sicher, dass das Inlet eines Threads atomar auf die Variablen zugreift. Cilk stellt *nicht* sicher, dass Inlets verschiedener Threads derselben Funktion atomar auf die Variablen zugreifen [Sup].

2.3.2 Aufruf

Ein Inlet wird wie eine C-Funktion aufgerufen. Das erste Argument ist ein valider spawn Aufruf (siehe 2.1). Alle weiteren Parameter sind valide C-Argumente und werden vor dem `spawn` Aufruf evaluiert. Das Inlet wird aufgerufen sobald die Ausführung der übergebenen Cilk-Funktion beendet ist. Beispiel 2 zeigt die Verwendung eines Inlets anhand des bekannten Fibonacci-Beispiel.

Example 2.

```
cilk int fib(int n) {
  int res = 0;
  inlet void summer(int result) {
     res += result;
     return;
  }

  if (n < 2) {
    return n;
  } else {
    summer(spawn fib(n-1));
    summer(spawn fib(n-2));
    sync;
    return res;
  }
}
```

2.4 abort

Die Anweisung `abort` ermöglicht es laufende Funktion zu unterbrechen. Die ist unter anderem für spekulative Suche nützlich.

Cilk stellt sicher, dass alle Kindfunktionen, die von einem Thread erstellt wurden, terminiert werden. Die Rückgabewerte der Funktionen sind in diesem Fall undefiniert. Cilk garantiert nicht, dass die Threads sofort beendet werden.

3 Scheduling

Der Cilk Compiler und die Cilk Runtime sind für das Scheduling von Threads auf die vorhandenen Prozessoren verantwortlich. Dabei verwendet Cilk einen Work-Stealing-Algorithmus. Ein Prozessor, auch als *Worker* bezeichnet, arbeitet zuerst alle ihm zugewiesenen Threads ab. Sind alle Threads abgearbeitet versucht der Worker *(Thief)* ausstehende Prozesse von anderen Workern *(Victim)* zu stehlen (*Steal*). Durch den verwendeten Algorithmus kann die Anzahl der Steals bei aussreichender Parallelität des Programmes gering und der Synchronisationsaufwand klein gehalten werden. Im folgenden Kapitel gehe ich auf die Funktionsweise des Work-Stealing-Algorithmus ein und betrachte dessen Laufzeiteigenschaften.

3.1 Work-Stealing-Algorithmus

3.1.1 Worker

Zur Implementierung des Work-Stealing-Algorithmus verwaltet Cilk eine Ready-Deque für jeden Worker. Ein Worker verwendet dabei seine zugehörige Deque als Stack und legt vor der Erzeugung eines neuen Threads (spawn) einen *Frame* auf den Stack. Ein Frame enthält dabei Informationen über den Zustand der Elternfunktion vor dem `spawn` Aufruf. Die Ready-Deque enthält alle noch auszuführenden Frames eines Workers.

Ist ein Thread beendet, wird geprüft ob der aktuell ausgeführte Frame von einem anderen Worker gestohlen wurde. Wurde der Frame nicht gestohlen, wird er von der Deque heruntergenommen und die Ausführung des Eltern-Threads wird fortgeführt. Cilk gewährleistet dabei, dass ein Worker einen Frame ohne Konflikte auf die Deque legen kann. In dem Fall, dass kein Frame von einem anderen Worker gestohlen wurde, führt der Algorithmus Funktionen genauso aus wie ein C-Compiler. Cilk speichert die Aktivierungs-Frames manuell ab um im Falle eines Steal den Zustand der Funktion auf einem anderen Prozessor wiederherstellen zu können.

3.1.2 Thief

Wird ein Worker zu einem Thief, wählt er ein Victim zufällig aus und nimmt einen Frame vom Kopf der Ready-Deque des Vicitims. Der parallele Zugriff mehrerer Worker auf dieselbe Ready-Deque wird durch zählende Semaphore und Shared-Memory organisiert [MF98].

Der Thief legt den gestohlenen Frame an das Ende seiner Ready-Deque und bearbeitet diese als Worker. Der Thief verwendet hierfür eine spezielle Variante der aufzurufenden Funktion, um Datenabhängigkeiten zwischen Thief und Victim zu synchronisieren (siehe Kapitel 4).

3.1.3 Directed Acyclic Execution Graph

Die Ausführung eines Cilk Programmes kann durch einen Graphen visualisiert werden. Anhand dieses *Ausführungsgraphen* (*Directed Acyclic Execution Graph)* kann gezeigt werden, welche Einschränkungen bei der Ausführung von Threads nötig sind um ein effektives Scheduling zu ermöglichen.

In der folgenden Visualisierung werden Threads als Kreise dargestellt, die in Funktionen gruppiert sind. Die Erstellung eines neuen Threads wird als nach unten gerichtete gerade Kante zwischen zwei Threads dargestellt. Eine horizontale Kante beschreibt den Aufruf eines Nachfolge-Threads. Datenabhängigkeiten, beispielsweise die Verwendung eines Rückgabewertes durch den Eltern-Thread, werden als wellenförmige Kanten dargestellt. Abbildung 3.1 zeigt einen so entstandenen gerichteten, azyklischer Graph (DAG).

Es kann gezeigt werden, dass für einen beliebig angeordnete DAG kein guter Scheduling-Algorithmus existiert. Durch die von Cilk vorgenommenen Einschränkungen in der Sprache, kann ein effektiver Scheduling-Algorithmus erreicht werden [MF98]. Cilk erlaubt Datenabhängigkeit lediglich zwischen der aufrufenden und der aufgerufenen Funktion, d.h. nur auf oben gerichtete Kanten in der Visualisierung. Ein Graph mit diesen Eigenschaften nennt man *wohlgeordnet* [RDB94, DD01].

In Cilk wird ein wohlgeordneter Ausführungsgraph erreicht indem nur die Elternfunktionen Rückgabewerte der erstellten Kindfunktionen verwenden dürfen und `spawn` Aufrufe nur in Zuweisungen oder im kontrollierten Ausführungsbereich eines Inlets erlaubt sind (Inlets erlauben keinen `spawn` Aufruf). Andere Formen der Datenabhängigkeiten müssen explizit über Locking modelliert werden. Hierfür können Funktionen aus der Cilk Library verwendet werden.

3.2 Laufzeitbetrachtungen

Die Laufzeit eines Cilk Programmes wird durch die folgenden Faktoren bestimmt:

1. T_P Die Ausführungszeit eines Cilk Programmes auf P Prozessoren.
2. T_∞ *(Critical Path Length)* Ausführungszeit für den längsten Pfad von Abhängigkeiten. Dies entspricht der Ausführungszeit eines Cilk Programms auf unendlich vielen Prozessoren.
3. T_1 Die Ausführungszeit eines Cilk Programmes auf einem Prozessor.

Dabei gilt $T_P \geq \frac{T_1}{P}$, d.h. die nötige Gesamtarbeitszeit wird durch Parallelität nicht kleiner, und kann bestenfalls gleichmäßig auf P Prozessoren aufgeteilt werden. Zusätzlich gilt $T_P \geq T_\infty$, d.h. endlich viele Prozessoren können ein Programm nicht schneller ausführen als eine unendlichen Anzahl von Prozessoren.

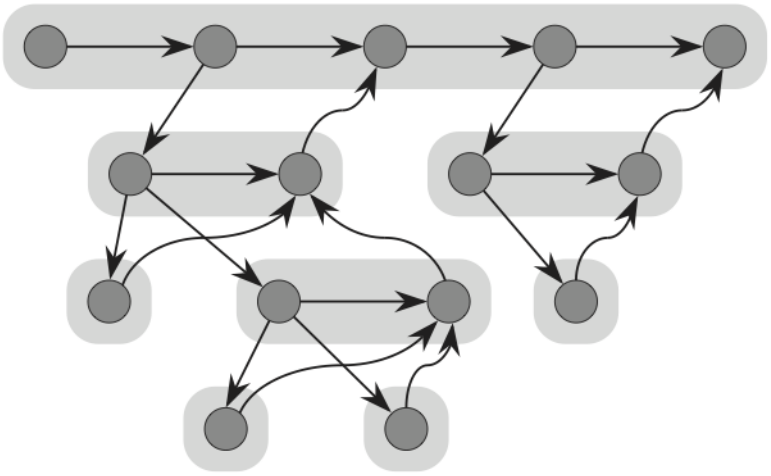

Abbildung 3.1: Ausführungsgraph [DD01].

Der Work-Stealing-Algorithmus von Cilk führt ein Cilk Programm in der erwarteten Zeit T_P mit

$$T_P = \frac{T_1}{P} + O(T_\infty) \tag{3.1}$$

aus [MF98]. Die Parallelität eines Cilk Programmes wird festgelegt als $Par = \frac{T_1}{T_\infty}$. Sie gibt an wie gut die Ausführung eines Cilk Programms parallelisierbar ist. Ist die Anzahl der Prozessoren deutlich geringer als die durschnittliche Parallelität eines Programmes ist $O(T_\infty)$ vernachlässigbar gegenüber $\frac{T_1}{P}$ und somit $T_P \approx \frac{T_1}{P}$. M. Frigo und C. Leiserson geben an, dass die durchschnittliche Parallelität der von ihnen getesteten Cilk Programme bei 200 lag und die Programme empirisch fast linear gegenüber ihrer C-Elision [MF98] skalieren.

4 Compilation

Die Compilation eines Cilk Programmes ist in drei Abschnitte unterteilt. Im Preprozessor Schritt wird der Cilk Quelltext mithilfe des Programmes `cilk2c` in einen C Quelltext umgewandelt. Cilk Anweisungen werden dabei durch C Preprozessor Macros ersetzt, die in der Cilk Library definiert sind. Der so entstanden C Quelltext wird im zweiten Schritt mit einem C Compiler - in der Regel der GNU C Compiler (`gcc`) - compiliert und abschließend gegen die Cilk Library gelinkt [Sup]. Cilk fügt dem entstandenen Programm zusätzliche Kommandozeilenparameter zur Steuerung des Scheduling und zur Generierung von Laufzeitstatistiken hinzu.

4.1 Compilierte Varianten einer Funktion

Ist die Parallelität eines Programmes hoch, ist die Anzahl der erwartenden Steals gering, da immer genügend Frames pro Prozessor verfügbar sind. Der Worker stiehlt deshalb selten Frames von anderen Workern. Die Compilationsstrategie von Cilk geht davon aus, dass ein normales Cilk Programm eine ausreichend hohe Parallelität besitzt und Steals selten vorkommen [MF98]. Der Cilk Preprozessor generiert deshalb für jede Funktion eine *schnelle* und eine *langsame* Variante, um den Synchronisationsaufwand gering zu halten. Die langsame Variante wird aufgerufen, wenn ein Frame gestohlen wurde und auf einem anderen Worker ausgeführt wird. Der gestohlene Frame stellt die Synchronisation der Daten zwischen Thief und Victim sicher. Alle weiteren Funktionsaufrufe verwenden wieder die schnelle Variante. Durch die Ordnung des Ausführungsgraphen werden alle `spawn` Aufrufe innerhalb einer langsamen Variante durch diese synchronisiert. Die langsame Variante macht ihren Rückgabewert dem ursprünglichen Worker verfügbar, wenn alle von ihre erstellten Kindfunktionen beendet sind.

Bei der häufiger aufgerufenen schnellen Variante entfällt dies. Die schnelle Variante entspricht großteils der C-Elision der Funktion, beinhaltet aber Anweisungen zur Speicherung des aktuellen Frames in der Deque des Workers. Synchronsiationsanweisungen wie `sync` werden ignoriert. Wird eine Funktion aufgerufen, initialisiert diese zunächst eine Frame Struktur, um den aktuellen Zustand zu sichern. Diese Struktur wird der Deque angehängt. Nach der Abarbeitung der Funktion wird der Frame wieder von der Deque genommen [Sup]. Beispiel 3 zeigt den C Quelltext einer schnellen Variante.

Example 3.

```
int fib(int n) {
  struct _fib_frame *_frame;
  _INIT_FRAME(_frame, sizeof(struct _fib_frame), _fib_sig);
  {
    if (n < 2) {
      _BEFORE_RETURN_FAST();
      return (n);
    } else {
      int x;
      int y;
      {
        _frame->header.entry=1;
        _frame->scope0.n=n;
        x=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,0,x);
      }
      {
        _frame->header.entry=2;
        _frame->scope1.x=x;
        y=fib(n-2);
```

```
            _XPOP_FRAME_RESULT(_frame,0,y);
        } /* sync */;
        {
            _BEFORE_RETURN_FAST();
            return (x+y);
        }
    }
  }
}
```

Die langsame Variante erweitert die schnelle Variante um Macros zur Synchronisation von Threads. Dafür speichert es die Rückgabewerte der Kind-Threads temporär ab. An jedem Synchronisationspunkt prüft die langsame Variante, ob Rückgabewerte von laufenden Threads ausstehen. Ist dies der Fall wird die langsame Variante an den Scheduler übergeben und vom System in einen wartenden Zustand überführt.

4.2 Scheduler

Die nötigen Scheduling Instruktionen werden in beide Varianten des Funktionen mithilfe von Macros eincompiliert. Nachdem eine Funktion beendet wurde, prüft die compilierte Cilk-Funktion, ob noch Frames in der zugehörigen Ready-Deque des Prozessors verfügbar sind. Ist dies nicht der Fall wird ein Steal wie beschrieben vorgenommen. Die Anweisungen zur Ausführungen einzelner Funktionen und das speichern der Aktivierungs-Frames wird ebenfalls direkt in den transformierten C-Quelltext übernommen.

4.3 Overhead

Im Gegensatz zu der C-Elision hat die Cilk Variante eines Programmes einen Overhead für die Scheduling-Entscheidungen und die Synchronisation von Workern. Diese sind

1. Aktivierungs-Frames müssen manuell allokiert werden und auf die Ready-Deque gelegt werden.
2. Variablen müssen vor einem `spawn` Aufruf im aktuellen Frame gespeichert werden.
3. Eine Funktion muss nach jedem `spawn` Aufruf prüfen ob ihr Frame gestohlen wurde.
4. Aktivierungs-Frames müssen vor der Rückgabe freigegeben werden.

5 Fazit

Mit dem Aufkommen von günstigen Multiprozessosystemen in den letzten fünf Jahren gewinnen parallelisierbare Programmiersprachen an Bedeutung. Cilk Ansatz die Programmiersprache C durch wenige Anweisungen zu erweitern ermöglicht dabei einfach und kostengünstig Legacy Code zu parallelisieren.

Kern von Cilk ist dabei der Work-Stealing-Algorithmus. Es hat sich gezeigt, dass er robust ist und bei ausreichender Parallelität eines Programmes einen nahezu linearen Geschwindigkeitsgewinn ermöglicht. Work-Stealing-Algorithmen haben deshalb ihren Weg auch in andere Programmiersprachen, wie beispielsweise Fortress [FWe], gefunden.

Dennoch wird Cilk hauptsächlich im akademischen Rahmen eingesetzt. Gründe hierfür können sein, dass Cilk erst seit 2007 professionell vermarktet wird und sich spät ein stabiler Sprachkern entwickelt hat. Zudem werden vermehrt andere Programmiersprachen wie Java oder C++ eingesetzt um parallele Programme zu entwickeln.

Es ist nicht zu erwarten, dass Cilk in den nächsten Jahren vermehrt eingesetzt wird.

Literatur

[CPP] *Intel Cilk++ Software Development Kit.* `http://software.intel.com/en-us/articles/intel-cilk/`.

[CWe] *The Cilk Website.* `http://supertech.csail.mit.edu/cilk/`.

[DD01] Don Daily, Charles E. Leiserson: *Using Cilk to Write Multiprocessor Chess Programs.* Technischer Bericht, MIT Laboratory for Computer Science, 2001.

[FWe] *First Impressions of the Fortress Language.* `http://software.intel.com/en-us/articles/first-impressions-of-the-fortress-language/`.

[MF98] Matteo Frigo, Charles E. Leiserson, Keith H. Randall: *The Implementation of the Cilk-5 Multithreaded Language.* Technischer Bericht, MIT Laboratory for Computer Science, 1998.

[RDB94] Robert D. Blumofe, Charles E. Leiserson: *Scheduling multithreaded computations by work stealing.* In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Seite 356 368, November 1994.

[Sup] Supercomputing Technologies Group: *Cilk 5.4.6 Reference Manual.*