Ausnutzung von Mehrkernpotenzialen durch asynchrone Iterationen

René Franke, Wolf Zimmermann

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik
Von-Seckendorff-Platz 1, 06120 Halle, Deutschland

{rene.franke, wolf.zimmermann}@informatik.uni-halle.de

Zusammenfassung

Hierarchisch gewachsene Software ist in aktuellen Mehrkernumgebungen meist ineffizient, weil die verfügbaren Hardwareressourcen nicht ausgenutzt werden können. Synchronisationsmechanismen limitieren die parallele Programmabarbeitung. Ziel dieser Arbeit ist die Minimierung von Synchronisationsanforderungen durch asynchrone Iterationen. Die Leerlaufzeiten von Prozessen werden dabei minimiert, indem auf veraltete Daten zurückgegriffen wird, statt auf die Bereitstellung aktueller Daten zu warten. Wir zeigen, welches Potenzial die asynchrone Abarbeitung bietet und unter welchen Bedingungen das Vorgehen anwendbar ist.

1 Einleitung

Aktuelle handelsübliche Rechner sind typischerweise mit mindestens vier Prozessoren ausgestattet. Die Prozessorhersteller wollen so dem Wunsch nach immer mehr Rechenleistung gerecht werden. Doch hierarchisch gewachsene Software, die unter den Bedingungen von Einkernumgebungen entwickelt und optimiert wurde, kann diese zusätzlichen Ressourcen nicht ausnutzen. Ein Grund liegt in der Serialisierung von Zugriffsoperationen durch Sperrmechanismen. In Mehrkernumgebungen sollten Sperren möglichst vermieden werden, um die Parallelität zu erhöhen. Für die Überführung von Software auf Mehrkernsysteme ist deshalb ein Reengineering-Prozess notwendig.

Die Arbeit ist wiefolgt gegliedert: Kapitel 2 motiviert die Problemstellung, bevor in Kapitel 3 die Idee asynchroner iterativer Methoden aufgegriffen wird. Anschließend werden in Kapitel 4 erste Resultate präsentiert und in Kapitel 5 zukünftige Untersuchungen vorgestellt.

2 Synchronisation als Flaschenhals

Das Reengineering besteht dabei aus einer effizienten Aufteilung von Teilfunktionalitäten der Software auf die zur Verfügung stehenden Hardwareressourcen. Die Kommunikation der Prozesse zum Austausch von Daten und zur Synchronisation ist ein limitierender Faktor in der Paralleliserbarkeit von Anwendungen [4]. Je häufiger ein Prozess auf Daten eines anderen Prozesses warten muss, desto geringer ist der parallele Anteil der Software. Da die einzelnen Prozessoren unabhängig voneinander arbeiten und ggf. auch unterschiedlich getaktet werden können, ergeben sich Herausforderungen im Hinblick auf die Ausnutzung der Prozessoren. Insbesondere können die Ausführungszeiten von Prozessen auf unterschiedlichen Prozessoren stark voneinander abweichen. Müssen alle Prozesse der Software auf einen *langsamen* Prozess warten, verschlechtert das die Performance der Software erheblich.

Synchronisation ist notwendig, wenn ein Prozess auf die Daten eines anderen Prozesses warten muss. Neben der Form der expliziten Synchronisation durch die Software, erfolgt eine implizite Synchronisation durch die Sicherstellung der Kohärenz in den Zwischenspeichern der einzelnen Prozessoren. Das Lesen eines Datums liefert dabei immer den zuletzt geschriebenen Wert zurück. Befindet sich der aktuelle Wert im Zwischenspeicher eines anderen Prozessors, so muss dieser erst über weitere Zwischenspeicherebenen zum betroffenen Prozessor-Cache gesendet werden.

3 Asynchrone Iterationen

Asynchrone iterative Verfahren sind gekennzeichnet durch die Verwendung aktuell verfügbarer Daten innerhalb eines Iterationsschrittes zur Ausführung von Operationen auf den Daten. In einem asynchronen Iterationsschritt ist es unerheblich, ob ein verwendetes Datum aktuell gültig ist oder bereits veraltet ist und die Änderung nur noch nicht im ausführenden Prozess angekommen ist. Die Verfahren manipulieren solange in jedem Iterationsschritt eine nichtleere Teilmenge der Daten, bis keine Änderung mehr vorgenommen werden kann und ein Fixpunkt der Iteration erreicht ist. Asynchrone Iterationen vermeiden dadurch Leerlaufzeiten, in denen Operationen lediglich auf die Verfügbarkeit von neueren Daten warten und erhöhen somit die Parallelisierbarkeit.

Durch die fehlende Synchronisation der Prozesse können Operationen auf den veralteten Daten Inkonsistenzen erzeugen. [2] weist nach, dass die Existenz ei-

ner vollständigen Halbordnung auf dem Lösungsraum des Problems ein hinreichendes Kriterium zur Konvergenz der Fixpunktiteration ist und die asynchrone Iteration zum gleichen Fixpunkt gelangt wie die synchrone Iteration. In jedem Iterationsschritt berechnet eine nichtleere Teilmenge aller Prozesse eine neue Lösung, die sich dem Fixpunkt annähert. Asynchrone iterative Verfahren lassen sich auf softwaretechnische Problemfelder anwenden: Graphalgorithmen lassen sich so nicht nur asynchron parallel ausführen, sondern können etwa auch Elemente, die einen großen Einfluss auf die Lösung haben, priorisieren [5].

4 Erste Ergebnisse

Numerische Verfahren bilden einen Spezialfall asynchroner Iterationen ab und sind bereits gut erforscht [1, 3]. Zur Motivation asynchroner iterativer Verfahren wurden daher Näherungsverfahren zur Lösung linearer Gleichungssysteme untersucht. Abbildung 1 zeigt das Potenzial asynchroner iterativer Verfahren zur numerischen Lösung linearer Gleichungssysteme.

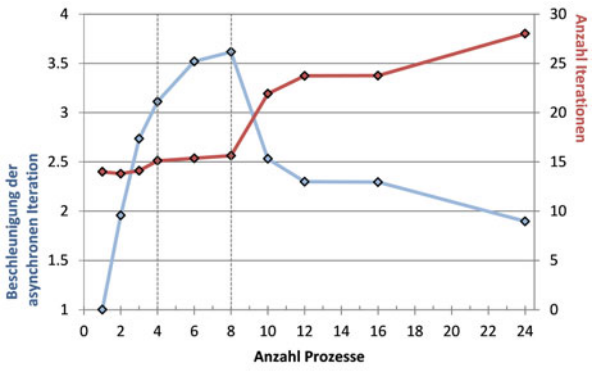

Abbildung 1: Asynchrone Iteration zur Lösung linearer Gleichungssysteme

Auf einem aktuellen Vierkernrechner kann ein Performancegewinn von etwa Faktor 3 im Vergleich zur sequentiellen Lösung erreicht werden. Werden mehr als acht parallele Prozesse verwenden, müssen sich einige Prozesse Rechenkerne teilen, was die Performance mindert. Außerdem steigt die Anzahl der benötigten Iterationen durch die Verwendung veralteter Werte nur unwesentlich an. Erst ab der zuvor erwähnten Architekturgrenze von acht Prozessen erhöht sich dieser Wert deutlich, da sich Prozesse gegenseitig in ihrer Ausführung verhindern.

Die Idee asynchroner iterativer Verfahren lässt sich auf Kürzeste-Wege-Probleme in Graphen anwenden. Dort existiert eine vollständige Halbordnung, da in jedem Iterationsschritt der gefundene kürzeste Weg höchstens noch kürzer wird. Asynchrone iterative Verfahren liefern ähnliche Resultate hinsichtlich der Performance wie das vorgestellte numerische Verfahren. Kanten mit geringem Kantengewicht können priorisiert werden, weil sie mit hoher Wahrscheinlichkeit in einem kürzesten Weg enthalten sind [5].

5 Ausblick

Asynchrone Iterationen können zur Steigerung der Softwareperformance in Mehrkernumgebungen beitragen. Notwendig dafür ist allerdings die Identifikation geeigneter Strukturen bzw. die Transformation in geeignete Strukturen. Aus den manuellen Überführungen können zukünftig Erkenntnisse gewonnen werden, die zu einer halbautomatischen Parallelisierung durch Annotationen im Quellcode beitragen können. Perspektivisch ist auch eine Implementierung in Compiler denkbar, um Programmteile automatisch zu parallelisieren. Jede Schleife eines Programms kann dabei als Fixpunktiteration betrachtet werden.

In einem nächsten Schritt soll die Skalierbarkeit der Resultate für andere Architekturen untersucht werden. Dabei sollen insbesondere Abschätzungen zur Anzahl der benötigten Iterationen getroffen werden, die hauptsächlich verantwortlich für die Performance der Lösung ist.

Zugleich liefern die Ergebnisse eine Motivation für die Abschwächung der Konsistenzbedingungen in aktuellen Prozessor-Caches. Asynchrone Iterationen sind Szenarien, in denen die Sicherstellung der Cache-Kohärenz nicht notwendig ist. Durch Reengineering geeigneter Strukturen in asynchrone Prozesse können Mehrkernpotenziale effizient ausgenutzt werden.

Danksagung

Diese Arbeit wurde gefördert vom Bundesministerium für Bildung und Forschung (BMBF) unter Kennzeichen 01IS11026.

Literatur

[1] D. P. Bertsekas and J. N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the 3rd International Conference on Supercomputing (ICS)*, pages 461–470, 1989.

[2] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *ACM SIGPLAN Notices*, 12(8):1–12, 1977.

[3] A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(1-2):201–216, 2000.

[4] D. Tetzlaff and S. Glesner. Making MPI Intelligent. In *Software Engineering 2012: Workshopband*, pages 75–88. Gesellschaft für Informatik, 2012.

[5] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *6th biennal Conference on Innovative Data Systems Research (CIDR)*, 2013.