High-throughput task parallelism with concurrency and locality control

Mein Name
Mein.Name@tu-cottbus.de

Ein geeignetes Thema für die eigene Abschlussarbeit zu finden kann sehr zeitaufwendig sein. Dieses Dokument enthält eine Vorlage für die Vorbereitung auf ein Arbeitsthema. Diese kursiv gesetzen Textabschnitte können später entfernt werden.

In der Regel werden Themen oder Ideen durch die Betreuer vorgeschlagen. Allerdings befreit dies nicht davon, das eigene Thema und die eigene Aufgabenstellung selbst zu verstehen. Eine Strategie um späteren Überraschungen vorzubeugen ist, die Aufgabenstellung und eine Einführung in das Thema selbst schriftlich auszuarbeiten – natürlich mit Unterstützung durch die Betreuer.

Dein Inhalt in diesem Dokument wird sicherlich mehrmals überarbeitet werden. Deshalb keine Angst: Nur fehlendes Material ist schlechtes Material! Eine Seite Text für Zusammenfassung+Einleitung ist ausreichend.

***Abstract*—*1) Was ist das Problem?* The parallel execution of tasks requires reasonable concurrency and locality control. *2) Warum ist das Problem interessant?* To achieve high throughput on many-core architectures, their high number of hardware threads has to be utilized due to a low instruction throughput of individual hardware threads. Consequently, parallel execution with efficient control mechanisms is crucial. *3) Was will ich in der Arbeit machen?* This thesis explores efficient concurrency and locality control mechanisms for a parallel task execution model and evaluates implementation variants. *4) Welches Ergebnis wird angestrebt?* The evaluation shows that thread groups for locality control combined with delegation queues for concurrency control provide a significant improvement over a static assignment of task to specific hardware threads.**

***Index Terms*—many-core, event-driven, asynchronous, concurrency, locality, tasklet**

I. Introduction/Einleitung

Die Einleitung wiederholt die Zusammenfassung von oben. Hier ist nun mehr Platz für Begründungen, Erklärungen, Begriffsdefinitionen und Literaturverweise.

1–2) Was ist das Problem? Welches Anwendungsgebiet wird betrachtet? Warum ist eine Problemlösung interessant? Für wen?

Many-Core Architectures (MCAs) provide a large number of "low frequency, low complexity cores" [1, p. 27]: high frequencies would lead to over-proportional higher power consumption [1, p. 2]; the space of complex cores can be traded for a higher number of simple cores [2]. In consequence, the instruction throughput of a single thread is low and high performance can be achieved only through parallelism.

To compensate for the lower instruction throughput, MCAs often provide parallelism on the instruction level (SIMD) and application-specific CPU extensions. Not every task can profit from this kind of acceleration; this is especially true for resource management tasks found, for example, in operating systems and middleware. Thus, the overhead generated by resource management on MCAs may take a larger share of the runtime even in single threaded applications.

However, applications are expected to be highly concurrent in order to exploit the large number of threads provided by MCAs. Ideally, the scalability of such applications should not be limited by slow, sequential resource management tasks. Conclusively, support for parallelism on the task level is needed.

Unfortunately, not every set of tasks can be parallelized effortlessly: shared resources may only support certain kinds of concurrent access or even no concurrent access at all. Concurrency control subsumes all mechanisms that enable obtaining correct results in concurrent programs; it includes mutual exclusion as well as optimistic transaction-based mechanisms.

Another factor becoming more important when dealing with MCAs is the placement of tasks relative to resources. Some resources are accessible only by certain threads (a specific cache or TLB), whereas other resources are accessible by all threads but the access costs depend on the thread (NUMA, data in caches). Analogously to concurrency control, locality control is used here to subsume all mechanisms that allow to specify where a task can be executed. This includes assigning each task to a specific thread as well as more abstract mechanisms like executing a task on any thread of an arbitrary chosen group.

3) Was will ich in der Arbeit machen? Welchen Aspekt des Problems will ich lösen? Was ist die grobe Lösungsidee? Welche Aspekte sollen/müssen nicht behandelt werden?

In this thesis, existing task parallel programming models will be examined with a focus on concurrency and locality control mechanisms. Based on the results, a programming model using parallel tasklets will be developed. Tasklets are lightweight tasks which can not be suspended and run to completion. However, more complex task representations can easily be implemented on top of tasklets. The model provides thread groups as a locality control mechanism, that enforce hard locality constraints but also allows implementations to employ online algorithms for dynamic work sharing in order to improve the task throughput. Moreover, the model includes concurrency

control mechanisms which enable common synchronization patterns as well as sophisticated parallel algorithms.

4) Was soll erreicht werden? Wie soll dies experimentell nachgewiesen werden?

In order to examine the implications of the parallel execution of tasklets within thread groups, the model will be implemented in C++11. Primarily, the parallel execution of tasks with shared resources will be evaluated against controlling concurrency and locality by statically assigning tasks to individual threads; secondarily, implementation variants for work sharing and mutual exclusion will be evaluated.

5) Was wird durch meine Lösung vermutlich möglich? (in Bezug auf das gesamte Thema und eventuell darüber hinaus)

It is expected that the parallel tasklet model is well-suited to increase the parallelism of tasks by providing reasonable concurrency control mechanisms and allowing implementations to employ efficient work sharing algorithms. Delegation locking, a locality-aware locking mechanism, is expected to improve the performance of lock-based synchronization further.

II. Domain Analysis / Stand der Technik

Welche Aspekte des Problems sind vermutlich schon gelöst? Welche Publikationen behandeln welches Teilproblem? Zu welchen Teilproblemen wurde keine Literatur gefunden? Hier wird eine kompakte Übersicht über den Stand der Technik benötigt. Dabei geht es noch nicht um detaillierte Inhalte der Publikationen sondern um ihre grobe Einordnung. Dazu genügt es in der Regel, die Abstracts und die Conclusions am Ende der Artikel zu lesen. Dieses Kapitel wird später das wichtigste der schriftlichen Ausarbeitung, denn es ist für die Einordnung der eignen Ergebnisse und Erfolge notwendig.

- locality: caches, NUMA, consistency islands, distributed memory systems
- programming models: fork/join (Cilk+, OpenMP 4 Tasks), futures (Rust, Taco), event-based (node.js, libevent), thread synchronization primitives, Linux kernel tasklets, TinyOS, CSP (rust/go/StacklessPhython channels), Ada rendevous+monitor, Apple Grand Central Dispatch, Actor Models, Chapel, X10
- stackless execution: continuation passing, protothreads
- flat and hierarchical workstealing
- mutex implementations: flat combining, delegation locks, spinlocks. procedure chaining

III. Concept / Lösungsansatz

Wie sieht die Lösungsidee grob aus? Zum Beispiel: Welche Modelle, Datenstrukturen, Algorithmen, oder Software-Komponenten müssen entwickelt werden? Welche bekannten Lösungen sollen übertragen werden? Das zweitwichtigste Kapitel der späteren Ausarbeitung.

- Design Criteria: light-weight, concurrent execution of task, building blocks for concurrency control that allow easy implementation of typical synchronization pattern (monitor, reader/writer) but also allows for sophisticated algorithms (lock-/wait-free algorithms)
- locality control: Thread groups as shared task queues. Any thread of a group can dequeue and process tasks from the group's queue. Enforce strict locality constraints. Match NUMA locality. Threads can be in multiple groups.
- concurrency control: Mutexes by delegation queues and combiner. Tasks that failed to acquire a mutex are enqeued at the mutex. When releasing a mutex, next waiting enqueued task is activated.
- usage examples: Monitor, Reader/Writer locks, light-weight fork/join, Coroutines, nested locks, condition variables

IV. Implementation

Was ist notwendig, um einen Prototypen für die Evaluation und Bewertung der Lösung zu erreichen? Was kann dafür weggelassen werden? Was muss für die Umsetzung des Konzeptes programmiert werden?

- Mapping of the concept to a C++11 interface
- implement tasklets, queues, locking mechanisms
- implementation variants: shared MWMR queues, non-hierarchical workstealing, hierarchical workstealing, pseudo groups with single thread
- functionality tests and benchmark applications: fibonacci function, fork-join multicast, shared atomic increment object, ...

V. Evaluation

Welche Experimente, Messungen, Vergleiche sind notwendig, um eine Lösung im Vergleich zu anderen Lösungen zu bewerten? Welche Hypothesen sollen belegt bzw. widerlegt werden? Was wird benötigt um diese Experimente durchführen zu können?

Main question: are task groups better than static thread binding? How many threads can go into one group. Implementation variants are evaluated against each other: Delegation Queues against naive implementation; work Stealing against naive implementations.

VI. Conclusions

Kurzzusammenfassung: Was wurde gemacht? Was kam dabei raus? Was wird dadurch möglich? Welche Fragen und Probleme können auf dieser Grundlage als nächstes untersucht werden?

The chosen execution model provide a solid foundation for efficient concurrent systems on many-core architectures. Concurrent task execution on shared resources has been shown to be advantageous compared to sequentialization on fixed hardware threads.

References

[1] A. Vajda, *Multi-core and Many-core Processor Architectures*. Springer US, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-9739-5

[2] J. Manferdelli, N. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808–815, May 2008. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2008.917730