Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 73–84

www.elsevier.com/locate/entcs

Generative Version of the FastFlow Multicore Library

Zalán Szűgyi[1] Norbert Pataki[2]

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Abstract

Nowadays, one of the most important challenges in programming is the efficient usage of multicore processors. Many new programming languages and libraries support multicore programming.
FastFlow is one of the most promising multicore C++ libraries. Unfortunately, a design problem occurs in the library. One of the most important methods is pure virtual function in a base class. This method supports the communication between different threads. Although, it cannot be template function because of the virtuality, hence, the threads pass and take argument as a `void*` pointer. The base class is not template neither. This is not typesafe approach. We make the library more efficient and safer with the help of generative technologies.

Keywords: multicore programming, C++, FastFlow, template

1 Introduction

The recent trend to increase core count in processors has led to a renewed interest in the design of both methodologies and mechanisms for the effective parallel programming of shared memory computer architectures. Those methodologies are largely based on traditional approaches of parallel computing.

Usually, low-level approaches supplies the programmers only with primitives for flows-of-control management (creation, destruction), their synchronization and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). For instance, POSIX thread library can be used to this purpose. Programming parallel complex applications in this way is certainly hard; tuning them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in the core's caches.

[1] Email: lupin@ludens.elte.hu
[2] Email: patakino@elte.hu

doi:10.1016/j.entcs.2011.11.039

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g. streaming) parallel applications. Avoiding memory fences means not only avoiding locks but also avoiding any kind of atomic operation in memory (e.g. Compare-And-Swap, Fetch-and-Add). While there exists several assessed fence-free solutions for asynchronous symmetric communications, these results cannot be easily extended to asynchronous asymmetric communications that are necessary to support arbitrary streaming networks.

The important approach to ease programmer's task and improve program efficiency consist in to raise the level of abstraction of concurrency management primitives. For example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimize cache flushing or maximize load balancing of cores. Synchronization primitives can be also abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc.

This kind of abstraction significantly simplify the hand-coding of applications. However, it is still too low-level to effectively automatize the optimization of the parallel code: the most important weakness here is in the lack of information concerning the intent of the code (idiom recognition); inter-procedural/component optimization further exacerbates the problem.

Recently, there has been a trend of generating programs from high-level specifications. This is called the generative approach, which focuses on synthesizing implementations from higher-level specifications rather than transforming them. From this approach, programmers' goal is captured by the specification. In addition, technologies for code generation are well developed (staging, partial evaluation, automatic programming, generative programming) [7]. FastFlow [4], TBB [13] and OpenMP [8] follow this approach. The programmer needs to explicitly define parallel behaviour by using proper constructs, which clearly bound the interactions among flows-of-control, the read-only data, the associativity of accumulation operations and the concurrent access to shared data structures.

FastFlow is a parallel programming framework for multi-core platforms based upon non-blocking lock-free/fence-free synchronization mechanisms. The framework is composed of a stack of layers that progressively abstracts out the programming of shared-memory parallel applications. The stack has two different goals: to ease the development of applications and make them very fast and scalable. FastFlow is particularly targeted to the development of streaming applications.

Templates are key elements of C++ programming language [15]. They enable data structures and algorithms be parameterized by types thus capturing commonalities of abstractions at compile time without performance penalties at runtime. *Generic programming*, a recently emerged programming paradigm for writing reusable components – most cases data structures and algorithms – is implemented in C++ with heavy use of templates.

The worker threads of FastFlow are passing data between each other via `void*` pointers, which are then processed by the receiving thread. This is rather necessary for implementation reasons. At the core of the FastFlow framework there is an