Future Generation Computer Systems 84 (2018) 22–31

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

On the adequacy of lightweight thread approaches for high-level parallel programming models

Adrián Castelló [a,*], Rafael Mayo [a], Kevin Sala [b], Vicenç Beltran [b], Pavan Balaji [c], Antonio J. Peña [b]

[a] Universitat Jaume I de Castelló, 12071 Castelló de la Plana, Spain
[b] Barcelona Supercomputing Center (BSC), 08034 Barcelona, Spain
[c] Argonne National Laboratory, Lemont, IL, USA

HIGHLIGHTS

- Design and implementation of OpenMP and OmpSs on top of lightweight threads.
- Analysis of the relationship between programming models and lightweight threads.
- Performance evaluation in different OpenMP and OmpSs scenarios.

ARTICLE INFO

Article history:
Received 17 August 2017
Received in revised form 15 December 2017
Accepted 8 February 2018
Available online 21 February 2018

Keywords:
Lightweight threads
OpenMP
OmpSs
GLT
POSIX threads
Programming models

ABSTRACT

High-level parallel programming models (PMs) are becoming crucial in order to extract the computational power of current on-node multi-threaded parallelism. The most popular PMs, such as OpenMP or OmpSs, are directive-based: the complexity of the hardware is hidden by the underlying runtime system, improving coding productivity. The implementations of OpenMP usually rely on POSIX threads (pthreads), offering excellent performance for coarse-grained parallelism and a perfect match with the current hardware. OmpSs is a task oriented PM based on an ad hoc runtime solution called Nanos++; it is the precursor of the tasking parallelism in the OpenMP tasking specification. A recent trend in runtimes and applications points to leveraging massive on-node parallelism in conjunction with fine-grained and dynamic scheduling paradigms. In this paper we analyze the behavior of the OpenMP and OmpSs PMs on top of the recently emerged Generic Lightweight Threads (GLT) API. GLT exposes a common API for lightweight thread (LWT) libraries that offers the possibility of running the same application over different native LWT solutions. We describe the design details of those high-level PMs implemented on top of GLT and analyze different scenarios in order to assess where the use of LWTs may benefit application performance. Our work reveals those scenarios where LWTs overperform pthread-based solutions and compares the performance between an ad hoc solution and a generic implementation.

1. Introduction

In the past few years, the number of cores per processor has increased steadily, reaching impressive counts such as the 260 cores per socket in the Sunway TaihuLight supercomputer [1], which was ranked #1 for first time in the June 2016 TOP500 List [2].

The trend followed in that list indicates that future exascale systems will support massive on-node parallelism, deploying thousands of cores per socket. Extracting the computational power of those machines will thus require efficient libraries and programming models (PMs). The most popular approaches to obtain acceptable on-node performance rely on POSIX threads (pthreads) application programming interface (API) [3] or directive-based PMs such as OpenMP [4] or OmpSs [5].

Directive-based PMs are usually implemented on top of the pthreads API, which matches perfectly the current hardware and coarse-grained parallelism. Because of the high cost of management, however, it fails to accommodate new software paradigms that target dynamically scheduled, fine-grained parallelism.

Several lightweight thread (LWT) libraries have been implemented in the last years to tackle fine-grained and dynamic software requirements [6]. Each LWT solution features its own PM and target environment. Some of these solutions are implemented for

* Corresponding author.
E-mail addresses: adcastel@uji.es (A. Castelló), mayo@uji.es (R. Mayo), ksala@bsc.es (K. Sala), vbeltran@bsc.es (V. Beltran), balaji@anl.gov (P. Balaji), antonio.pena@bsc.es (A.J. Peña).

https://doi.org/10.1016/j.future.2018.02.016
0167-739X/

a specific Operating System (OS), such as Windows Fibers [7] and Solaris Threads [8]. Compared with those, ConverseThreads [9] and Nanos++ [10] support a specific high-level PM; Charm++ [11] and OmpSs [5], respectively. There are also general-purpose solutions such as MassiveThreads [12], Qthreads [13], and Argobots [14]. The Generic Lightweight Threads (GLT) API [15], [16] is an effort to unify these LWT solutions under a unique PM in order to foster productivity and portability with negligible overhead. This lightweight layer offers the common functionality of LWT solutions and is currently implemented on top of MassiveThreads, Qthreads, and Argobots. As a result, a runtime or application based on GLT requires no changes in order to be executed on top of any of these three LWT solutions.

In this paper we analyze common OpenMP and OmpSs parallel patterns and discuss how LWTs deal with them, in comparison with traditional approaches. While OpenMP is the most widely-adopted directive-based PM, OmpSs is the precursor of task-parallelism and features a runtime which leverages a custom LWT implementation. We evaluate our implementations and compare their performances with those obtained when using the original runtimes.

In order to perform the comparison, we have implemented the OpenMP and OmpSs runtimes on top of the GLT API, called Generic Lightweight Thread OpenMP (GLTO) and Generic Lightweight Thread OmpSs (GOmpSs), respectively. Our OpenMP implementation is based on the open-source BOLT project [17], which is in turn based on LLVM [18]. The LLVM OpenMP runtime shares the code developed in the Intel OpenMP [19] solution. Our OmpSs version is based on the Nanos++ library [10] from the Barcelona Supercomputing Center (BSC).

Our study reveals that the use of LWTs instead of pthread-based approaches in the OpenMP PM may yield performance benefits, depending on the application nature. In addition, our results expose that the performance with the OmpSs runtime implemented on top of GLT is close to that obtained with an ad-hoc implementation, improving the task management in fine-grained code tasks.

In summary, the main contributions of this paper are as follows: (1) design of OpenMP and OmpSs runtimes on top of a generic LWT API; (2) analytical study of the relationship between high-level PMs and LWT solutions; and (3) the experimental performance evaluation of that relationship in different OpenMP and OmpSs scenarios.

The rest of the paper is organized as follows. Section 2 provides some background information about OpenMP, OmpSs, and GLT. Section 3 reviews a few related works. Section 4 details the GLTO implementation and Section 5 describes the GOmpSs implementation. Section 6 provides an in-depth performance analysis of the distinct scenarios. Finally, Section 7 contains our conclusions.

2. Background

In this section we review the OpenMP and OmpSs PMs and describe the GLT implementation and its interaction with the underlying LWT libraries.

2.1. OpenMP

The OpenMP API supports multiplatform shared-memory multiprocessing programming, and current implementations cover most architectures and operating systems. OpenMP offers a directive-based PM to parallelize a code by means of "pragmas". Intel and GNU offer two common OpenMP implementations that rely on pthreads in order to exploit concurrency.

The OpenMP runtimes are commonly composed of two main parts: the work-sharing constructs and task parallelism. In contrast

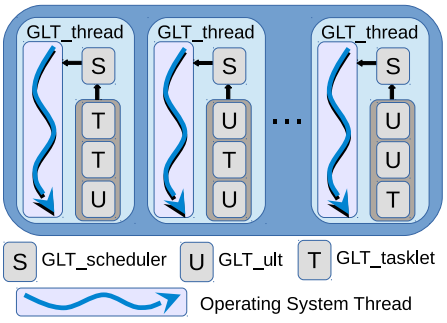

Fig. 1. PM offered by the GLT library.

to with work-sharing constructs, where all the OpenMP implementations follow a similar policy, distinct OpenMP implementations leverage different mechanisms for task management. In particular, while the GNU version implements a single task queue shared by all the threads, the Intel implementation incorporates one task queue for each thread and integrates workstealing for load balance control. In both solutions, the task management is separated from the work-sharing implementations because task directives were added in the OpenMP 3.0 specification.

2.2. OmpSs

OmpSs [20], developed at BSC, aims to provide an efficient programming model for heterogeneous and multicore architectures. It embraces a task-oriented execution model similar to the OpenMP tasking features.

OmpSs detects data dependencies between tasks at execution time, with the help of directionality clauses embedded in the code, and leverages this information to generate a task graph during the execution. This graph is then employed by the runtime to exploit the implicit task-parallelism, via a dynamic out-of-order, dependency-aware schedule. This mechanism provides a means to enforce the task execution order without the need for explicit synchronization. This PM is task-oriented and, therefore, it does not support work-sharing constructs.

2.3. Generic lightweight threads

GLT is a common API that was designed with the aim of unifying, under the same PM, a variety of LWT libraries. It is currently defined and implemented for three general-purpose LWT solutions: MassiveThreads, Qthreads, and Argobots.

Fig. 1 illustrates the PM offered by this API. Specifically, `GLT_thread` refers to the OS thread itself, while `GLT_ult` represents the user-level threads (ULTs). In addition, `GLT_tasklet`, a lighter work unit that does not own a stack (preventing migration or yield operations), is offered as part of the common API. While tasklets are natively supported by Argobots only, these are implemented on top of ULTs for Qthreads and MassiveThreads. `GLT_scheduler` acts differently depending on the underlying library and it may affect the performance of the PM but not the final result of the execution.

In principle adding an extra software layer between the user application and the underlying libraries may impact performance; however, GLT does not add any significant overhead because it offers a header-only version that allows the compilers to avoid the extra calls by embedding the LWT code by means of `static inline` declarations [21].

Despite some LWT solutions offer an API of more than 300 functions, GLT offers just 52 functions grouped in 7 modules: