Parallel Computing 40 (2014) 328–343

Contents lists available at ScienceDirect

Parallel Computing

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

Chunks and Tasks: A programming model for parallelization of dynamic algorithms

Emanuel H. Rubensson *, Elias Rudberg

Division of Scientific Computing, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden

ARTICLE INFO

Article history:
Available online 4 October 2013

Keywords:
Distributed memory parallelization
Dynamic data distribution
Dynamic load balancing
Fault tolerance
Parallel programming model
Determinism

ABSTRACT

We propose Chunks and Tasks, a parallel programming model built on abstractions for both data and work. The application programmer specifies how data and work can be split into smaller pieces, chunks and tasks, respectively. The Chunks and Tasks library maps the chunks and tasks to physical resources. In this way we seek to combine user friendliness with high performance. An application programmer can express a parallel algorithm using a few simple building blocks, defining data and work objects and their relationships. No explicit communication calls are needed; the distribution of both work and data is handled by the Chunks and Tasks library. This makes efficient implementation of complex applications that require dynamic distribution of work and data easier. At the same time, Chunks and Tasks imposes restrictions on data access and task dependencies that facilitate the development of high performance parallel back ends. We discuss the fundamental abstractions underlying the programming model, as well as performance, determinism, and fault resilience considerations. We also present a pilot C++ library implementation for clusters of multicore machines and demonstrate its performance for irregular block-sparse matrix–matrix multiplication.

1. Introduction

Parallel computing can be difficult. In order to have a functioning parallel program, one first needs to find algorithms that can be executed in parallel. Then, the algorithms must be expressed in such a way that the parallelism is exposed so that separate, independent parts can be identified. Furthermore, the work must be distributed among the available processors/nodes and data required by each processor must be communicated as needed. We present in this article a parallel programming model intended to facilitate the above process of developing parallel programs.

Our aim is to enable scalability up to any number of processors, on distributed memory machines and possibly on heterogeneous computers, not only for algorithms with static distribution of work and data but also for hierarchical, recursive algorithms where communication patterns are not known beforehand, and may change dynamically. To make this feasible without overwhelming programmers, a clean and easy-to-use model is required. Fault tolerance is also important, especially considering the trends in modern supercomputers, where hardware errors are unavoidable in any large parallel calculation.

1.1. Parallel programming models

Much research has been devoted to parallel programming models. To give some background to the design choices made while developing our new interface, we mention below some of the more well-known previous models. A programming model usually exists in the form of a tool, e.g. a library or language, that supports program development within the model.

* Corresponding author. Tel.: +46 184715717.
E-mail addresses: emanuel.rubensson@it.uu.se (E.H. Rubensson), elias.rudberg@it.uu.se (E. Rudberg).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.09.006

A crucial point here is the interface separating the concerns of the application programmer from the inner workings of the library or language.

Parallel computing has traditionally been dominated by threads, light-weight processes that share memory. Access to shared memory needs to be synchronized in order to achieve consistent behavior of the program. Unfortunately, proper synchronization is difficult to ensure; bugs are easy to introduce but often difficult to detect. Because of their nondeterministic behavior, threaded programs can be difficult to understand [1]. Also, although the shared memory model facilitates the implementation of dynamic algorithms, other models are needed to make use of distributed memory systems.

In message passing programming models (e.g. MPI and PVM) the application programmer has to decide how the data should be distributed and use explicit communication to make sure that the needed data is available whenever a task is to be performed. This gives the programmer control, but a drawback is that implementation of complex algorithms, for example requiring dynamic load balancing, is difficult.

An alternative is to use some programming model where data is shared via a distributed shared memory. The Linda programming model [2] is built on an associative logically shared memory called a tuple space. In implementations of Linda for distributed memory each processor manages a portion of the tuple space. There is in principle no way for the Linda implementation to know in advance which processes that will access a particular tuple. However, efficient implementations attempt to optimize the distribution of data based on observation of the tuple traffic between processes [2]. In case of static traffic patterns the runtime system can then quickly set up efficient communication channels. As in threaded programs it is easy to introduce nondeterministic behavior in Linda. You may for example in a conforming Linda application code request to read a tuple from tuple space for which several matching but different tuples exist. (The Linda runtime system will then arbitrarily choose one of them.)

Concurrent Collections (CnC) [3] is a programming model which similarly to Linda is built on associative containers. A key difference compared to Linda is that CnC is deterministic. Determinism is achieved by a single assignment rule which says that an entry in a data collection is not allowed to be overwritten. CnC differs from Linda also in that there is a distinction between the tags used to index a data collection and the data. However, tags are still chosen by the application programmer which means that CnC for distributed memory shares Linda's difficulties to make data available efficiently as outlined above.

Languages based on a Partitioned Global Address Space (PGAS) such as Unified Parallel C (UPC) [4] and CoArray Fortran [5] provide a layer of abstraction for distributed shared data objects, e.g. distributed arrays. While UPC and CoArray Fortran rely on static distributions more recent languages like X10 [6] and Chapel [7] also provide means to dynamically control the location of data and work.

In the Sequoia [8] and Hierarchical Place Trees [9] models, an application program is formulated as a hierarchy of tasks. The application is mapped to the memory hierarchy of different architectures using a hardware specification separate from the application source code combined with static or dynamic mapping of data and work to the hardware.

The actor model [10] can be seen as an extension of object oriented programming to support concurrency. Each actor has private state and a mailbox, and a program is driven by messages between such actors. When an actor receives a message it can change its state, create new actors, and/or send messages to other actors. One of the more well-known languages based on actors is Erlang, initially developed for telecom applications [11]. Charm++ is an actor-like programming model based on C++, targeting scientific computing applications [12].

In what we here will refer to as task-based approaches, the computation is driven by the registration of tasks that are executed by a runtime system. The application programmer writes the program in terms of tasks, usually specifying dependencies in one way or another, and a runtime engine schedules the tasks on available resources. In some task-based systems, e.g. SMPSs [13] and StarPU [14], tasks can only be registered in a single master thread while others such as Cilk [15], OmpSs [16], Scioto [17], SuperGlue [18], and XKaapi [19] allow recursive nesting with varying support for synchronization between tasks. Dependencies are usually determined to achieve sequential consistency [20] on the basis of annotations that indicate whether task parameters are read, written or both [13,14,16,18,19]. An alternative is to let the programmer express the task graph by explicitly specifying dependencies [14].

Most task-based programming models have initially been developed for multicore architectures, in some cases with support for accelerators. The application of task-based approaches to clusters of computers involves additional challenges concerning scalable distribution of data and work and resolution of remote dependencies between tasks. There have been several interesting efforts in this direction.

Scioto [17] and HotSLAW [21] are task-based programming models for distributed memory machines that capitalize on features provided by one-sided communication libraries and partitioned global address space models. In Scioto, dependencies between tasks are disallowed but tasks may synchronize their data accesses using primitives provided by the communication library. StarPU includes features to facilitate integration with MPI [22]. In short, tasks registered by the user are executed by the StarPU runtime which automatically ensures that the needed data is available. The user decides the distribution of data which also dictates the distribution of work. OmpSs uses a master/slave approach without synchronization between slaves [16]. Data may be distributed among the master and the slaves but all data traffic between slaves is directed by the master. Cilk-NOW is a runtime system for Cilk programs for networks of workstations [23]. The scheduling among workstations is based on work stealing, which is a scalable approach to achieve load balanced computations for dynamic algorithms. However, the work in [23] did not include a solution for the distribution of data. The DAGuE framework is a distributed memory task-based approach targeting dense linear algebra operations [24]. To achieve scalability, DAGuE