Parallel Computing 51 (2016) 79–87

Contents lists available at ScienceDirect

Parallel Computing

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

A task-uncoordinated distributed dataflow model for scalable high performance parallel program execution

Lucas A. Wilson [a,b,*], Jeffery von Ronne [a]

[a] Department of Computer Science, The University of Texas at San Antonio, San Antonio, Texas, USA
[b] Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas, USA

ARTICLE INFO

Article history:
Available online 6 November 2015

Keywords:
Distributed dataflow
Task-uncoordinated parallelism
Single assignment
Task coalescence
Vectorization

ABSTRACT

We propose a distributed dataflow execution model which utilizes a distributed dictionary for data memoization, allowing each parallel task to schedule instructions without direct inter-task coordination. We provide a description of the proposed model, including autonomous dataflow task selection. We also describe a set of optimization strategies which improve overall throughput of stencil programs executed using this model on modern multi-core and vectorized architectures.

1. Introduction

Current models for parallel program execution assume a level of reliability that will not be achievable as systems push closer to exascale. In order to be effective at such scales, execution models must be capable of elastically recovering from host failure mid-computation.

In order to meet these challenges, we have developed the Relentless Execution Model (REM) [1,2], a distributed dataflow model for task-uncoordinated execution on distributed memory parallel architectures where hardware volatility is assumed. REM uses uncoordinated scheduling processes running in parallel to execute dataflow program graphs as deterministic compute tasks that interact through a distributed, eventually-consistent single-assignment key-value store. The advantages of this approach are that new scheduling processes can be added to a computation elastically, and processes which exist on failing hardware can fall out of the execution pool without fatally effecting other processes.

A naive implementation of REM, where each atomic operation is scheduled independently as a fine-grained task would not be performant. Modern architectures are designed to handle sequential streams of instructions, managing instruction-level parallelism within the hardware. Additionally, modern architectures employ wide Single Instruction, Multiple Data (SIMD) units which enable greater exploitation of data parallelism by the hardware. Any execution model targeting modern architectures must be able to take advantage of these capabilities. This can be achieved by applying REM to course-grained, coalesced tasks.

In this paper, we provide a description of the proposed Relentless Execution Model, how it is able to effectively share work without explicit inter-task coordination, and how it performs autonomous task selection. We also describe several coalescing strategies – both on tasks and on dictionary labels – which yielded significantly improved performance on vectorized multi-core architectures. Experiments using a stencil test application demonstrate reasonable performance (37 GFLOPS single node, 429 GFLOPS multi-node) on Intel Xeon E5-2680v2 "Sandy Bridge" processors with dynamically scheduled tasks.

* Corresponding author at: Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas, USA. Tel.: +15122327351.
E-mail addresses: lucaswilson@acm.org (L.A. Wilson), vonronne@acm.org (J. von Ronne).

http://dx.doi.org/10.1016/j.parco.2015.10.013
0167-8191/

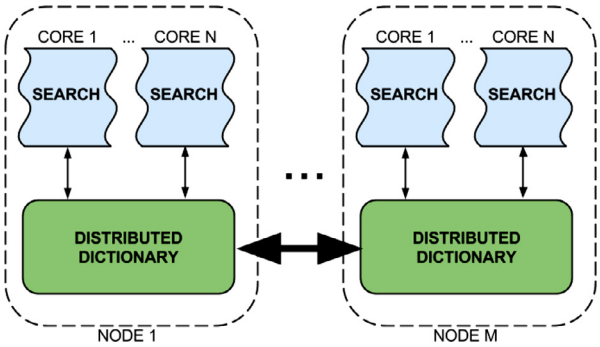

Fig. 1. The Relentless Execution Model.

2. Background

The relentless execution model is a dataflow execution model whose current implementation utilizes the memcached distributed key-value store. These existing building blocks are described in this section.

Dataflow Execution Models. Dataflow models focus exclusively on the availability of inputs to particular "functions" associated with nodes in a dataflow graph. When the input data is available, the code contained within the appropriate graph node is activated, and the computation is performed. This makes it well suited for the representation of parallel computations, where the inclusion of a program counter would make it necessary for the programmer to ensure the consistency of the system through various coordination mechanisms, such as mutual exclusion locks or synchronization barriers.

Distributed Dictionaries. Distributed Hash Tables (DHTs) [23–28] provide a limited subset of the tuple space data model [29], namely associative arrays as pair-based mappings from a key to a value. These systems solve the problem of distributing the tuple space into different address spaces by eliminating much of the complex querying associated with fully featured tuple space implementations.

Memcached. While the distributed dictionary projects already mentioned utilize the filesystem for storage, memcached [3] is a memory resident distributed key-value store. Remaining memory resident is an important and attractive feature for several reasons. First, remaining in memory will improve the response time of the distributed key-value store, as requests do not have to be either retrieved from or committed to disk. Second, existing work improves memcached for HPC platforms by taking advantage of Remote Direct Memory Access (RDMA) [4].

3. Distributed dataflow execution

Our proposed model executes dataflow programs in a distributed fashion using a novel scheduling algorithm which takes advantage label-value pairs in an eventually-consistent distributed dictionary to intelligently inform the traversal of the dataflow graph.

The Relentless Execution Model (REM) [1,2] uses uncoordinated scheduling processes running in parallel to execute dataflow program graphs as individual compute tasks that interact through a distributed, eventually-consistent single-assignment dictionary (Fig. 1). The advantages of this approach are that new scheduling processes can be added to a computation elastically, and processes which exist on failing hardware can fall out of the execution pool without fatally effecting other processes.

3.1. Explicit data dependency description

REM executes programs expressed as explicit data dependency descriptions. The description of a program consists of

1. a finite set T of tasks to be performed;
2. a finite set L of labels, which are used as keys in the shared dictionary;
3. a set $R \subseteq L$ of result labels whose association with values completes the REM program's execution;
6. a surjective function *producer*: $L \to T$ that maps dictionary labels to the task that produces the value for that label;
5. a function *requires* : $T \to \mathcal{P}(L)$ that maps each task to the labels of the inputs the task requires before it can be executed; and
6. a function *computes:* $T \to (L \nrightarrow V) \nrightarrow (L \nrightarrow V)$ that maps each task to the partial function it computes.

In a valid REM program, the *computes* function must be consistent with the *producer* and *requires* functions. That is, for every task $\tau \in T$, *computes*(τ) is required to be defined over the domain *requires*$(\tau) \to V$ and to have its image contained in $\{\ell \in L \mid \textit{producer}(\ell) = \tau\} \to V$. Furthermore, we require there to exist a partial ordering $\preceq$ among the tasks, such that for any $\tau_1 \in T$, $\tau_2 \in T$, and $\ell \in L$, $\tau_1 = \textit{producer}(\ell)$ and $\ell \in \textit{requires}(\tau_2)$ only if $\tau_1 \preceq \tau_2$.