Future Generation Computer Systems 30 (2014) 202–215

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A memory access model for highly-threaded many-core architectures☆

Lin Ma*, Kunal Agrawal, Roger D. Chamberlain

Department of Computer Science and Engineering, Washington University in St. Louis, United States

HIGHLIGHTS

- We design a memory model to analyze algorithms for highly-threaded many-core systems.
- The model captures significant factors of performance: work, span, and memory accesses.
- We show the model is better than PRAM by applying both to 4 shortest paths algorithms.
- Empirical performance is effectively predicted by our model in many circumstances.
- It is the first formalized asymptotic model helpful for algorithm design on many-cores.

ARTICLE INFO

Article history:
Received 25 January 2013
Received in revised form
14 May 2013
Accepted 17 June 2013
Available online 15 July 2013

Keywords:
PRAM
TMM
All Pairs Shortest Paths (APSP)
Highly-threaded many-core
Memory access model

ABSTRACT

A number of highly-threaded, many-core architectures hide memory-access latency by low-overhead context switching among a large number of threads. The speedup of a program on these machines depends on how well the latency is hidden. If the number of threads were infinite, theoretically, these machines could provide the performance predicted by the PRAM analysis of these programs. However, the number of threads per processor is not infinite, and is constrained by both hardware and algorithmic limits. In this paper, we introduce the Threaded Many-core Memory (TMM) model which is meant to capture the important characteristics of these highly-threaded, many-core machines. Since we model some important machine parameters of these machines, we expect analysis under this model to provide a more fine-grained and accurate performance prediction than the PRAM analysis. We analyze 4 algorithms for the classic all pairs shortest paths problem under this model. We find that even when two algorithms have the same PRAM performance, our model predicts different performance for some settings of machine parameters. For example, for dense graphs, the dynamic programming algorithm and Johnson's algorithm have the same performance in the PRAM model. However, our model predicts different performance for large enough memory-access latency and validates the intuition that the dynamic programming algorithm performs better on these machines. We validate several predictions made by our model using empirical measurements on an instantiation of a highly-threaded, many-core machine, namely the NVIDIA GTX 480.

1. Introduction

Highly-threaded, many-core devices such as GPUs have gained popularity in the last decade; both NVIDIA and AMD manufacture general purpose GPUs that fall in this category. The important distinction between these machines and traditional multi-core machines is that these devices provide a large number of low-overhead hardware threads with low-overhead context switching between them; this fast context-switch mechanism is used to hide the memory access latency of transferring data from slow large (and often global) memory to fast, small (and typically local) memory. Researchers have designed algorithms to solve many interesting problems for these devices, such as GPU sorting or hashing [1–4], linear algebra [5–7], dynamic programming [8,9], graph algorithms [10–13], and many other classic algorithms [14,15]. These projects generally report impressive gains in performance. These devices appear to be here to stay. While there is a lot of folk wisdom on how to design good algorithms for these highly-threaded machines, in addition to a significant body of work on performance analysis [16–20], there are no systematic theoretical models to analyze the performance of programs on these machines. We are interested in analyzing and characterizing performance of algo-

* Corresponding author. Tel.: +1 3144986029.
E-mail address: lin.ma@cse.wustl.edu (L. Ma).

0167-739X/$ – see front matter © 2013 The Authors. Published by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.06.020

rithms on these highly-threaded, many-core machines in a more abstract, algorithmic, and systematic manner.

Theoretical analysis relies upon models that represent underlying assumptions; if a model does not capture the important aspects of target machines and programs, then the analysis is not predictive of real performance. Over the years, computer scientists have designed various models to capture important aspects of the machines that we use. The most fundamental model that is used to analyze sequential algorithms is the Random Access Machine (RAM) model [21], which we teach undergraduates in their first algorithms class. This model assumes that all operations, including memory accesses, take unit time. While this model is a good predictor of performance on computationally intensive programs, it does not properly capture the important characteristics of the memory hierarchy of modern machines. Aggarwal and Vitter proposed the Disk Access Machine (DAM) model [22] which counts the number of memory transfers from slow to fast memory instead of simply counting the number of memory accesses by the program. Therefore, it better captures the fact that modern machines have memory hierarchies and exploiting spatial and temporal locality on these machines can lead to better performance. There are also a number of other models that consider the memory access costs of sequential algorithms in different ways [23–29].

For parallel computing, the analogue for the RAM model is the Parallel Random Access Machine (PRAM) model [30], and there is a large body of work describing and analyzing algorithms in the PRAM model [31,32]. In the PRAM model, the algorithm's complexity is analyzed in terms of its *work* – the time taken by the algorithm on 1 processor, and *span* (also called *depth* and *critical-path length*) – the time taken by the algorithm on an infinite number of processors. Given a machine with P processors, a PRAM algorithm with work W and span S completes in $\max(W/P, S)$ time. The PRAM model also ignores the vagaries of the memory hierarchy and assumes that each memory access by the algorithm takes unit time. For modern machines, however, this assumption seldom holds. Therefore, researchers have designed various models that capture memory hierarchies for various types of machines such as distributed memory machines [33–35], shared memory machines and multi-cores [36–40], or the combination of the two [41,42].

All of these models capture particular capabilities and properties of the respective target machines, namely shared memory machines or distributed memory machines. While superficially highly-threaded, many-core machines such as GPUs are shared memory machines, their characteristics are very different from traditional multi-core or multiprocessor shared memory machines. The most important distinction between the multi-cores and highly-threaded, many-core machines is the number of threads per core. On multi-core machines, context switch cost is high, and most models nominally assume that only one (or a small constant number of) thread(s) are running on each machine and this thread blocks when there is a memory access. Therefore, many models consider the number of memory transfers from slow memory to fast memory as a performance measure, and algorithms are designed to minimize these, since memory transfers take a significant amount of time. In contrast, highly-threaded, many-core machines are explicitly designed to have a large number of threads per core and a fast context switching mechanism. Highly-threaded many-cores are explicitly designed to hide memory latency; if a thread stalls on a memory operation, some other thread can be scheduled in its place. In principle, the number of memory transfers does not matter *as long as there are enough threads* to hide their latency. Therefore, if there are enough threads, we should, in principle, be able to use PRAM algorithms on such machines, since we can ignore the effect of memory transfers which is exactly what PRAM model does.

However, the number of threads required to reach the point where one gets PRAM performance depends on both the algorithm and the hardware. Since no highly-threaded, many-core machine allows an infinite number of threads, it is important to understand both (1) how many threads does a particular algorithm need to achieve PRAM performance, and (2) how does an algorithm perform when it has fewer threads than required to get PRAM performance? In this paper, we attempt to characterize these properties of algorithms. To motivate this enterprise and to understand the importance of high thread counts on highly-threaded, many-core machines, let us consider a simple application that performs Bloom filter set membership tests on an input stream of biosequence data on GPUs [3]. The problem is embarrassingly parallel, each set membership test is independent of every other membership test. Fig. 1 shows the performance of this application, varying the number of threads per processor core, for two distinct GPUs. For both machines, the pattern is quite similar, at low thread counts, the performance increases (roughly linearly) with the number of threads, up until a transition region, after which the performance no longer increases with increasing thread count. While the location of the transition region is different for distinct GPU models, this general pattern is found in many applications. Once sufficient threads are present, the PRAM model adequately describes the performance of the application and increasing the number of threads no longer helps.

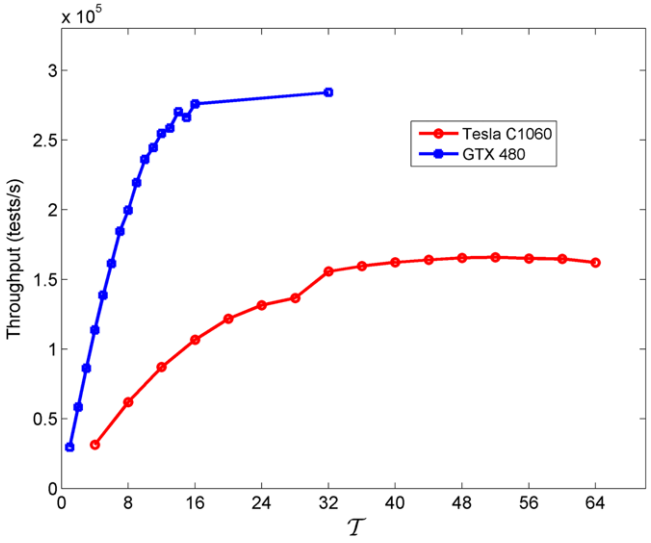

Fig. 1. Throughput of Bloom filter algorithm for set membership testing on biosequence data. Performance (in membership tests per second) is plotted vs. number of threads per processor both for a Tesla C1060 and a GTX 480 GPU.

In this work, we propose the *Threaded Many-core Memory (TMM)* model that captures the performance characteristics of these highly-threaded, many-core machines. This model explicitly models the large number of threads per processor and the memory latency to slow memory. Note that while we motivate this model for highly-threaded many-core machines with synchronous computations, in principle, it can be used in any system which has fast context switching and enough threads to hide memory latency. Typical examples of such machines include both NVIDIA and AMD/ATI GPUs and the YarcData uRiKA system. We do not try to model the Intel Xeon Phi, due to its limited use of threading for latency hiding. In contrast, its approach to hide memory latency is primarily based on strided memory access patterns associated with vector computation.

If the latency of transfers from slow memory to fast memory is small, or if the number of threads per processor is infinite, then this model generally provides the same analysis results as the PRAM analysis. It, however, provides more intuition. (1) Ideally, we want to get the PRAM performance for algorithms using the fewest