J. Parallel Distrib. Comput. 74 (2014) 2574–2581

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

What is ahead for parallel computing

Wen-mei Hwu

University of Illinois at Urbana-Champaign, United States

HIGHLIGHTS

- A clear articulation of what makes parallel algorithms hard in multicores and manycores.
- A practical classification of levels of challenges in parallel algorithms.
- An overview of practical techniques for improving parallel algorithm scalability and efficiency.
- A detailed real example of a highly optimized parallel GPU algorithm.

ARTICLE INFO

Article history:
Received 2 October 2013
Received in revised form
13 February 2014
Accepted 13 February 2014
Available online 12 March 2014

Keywords:
Parallel algorithms
Parallel data structures
Locality
Memoryicore bandwidth
GPU
Multicore
Algorithm optimization
Algorithm library
Data layout

ABSTRACT

With the industry-wide switch to multicore and manycore architectures, parallel computing has become the only venue in sight for continued growth in application performance. In order for the performance of an application to grow with future generations of hardware, a significant portion of its computation must be done with scalable parallel algorithms. It is therefore important to develop and deploy as many scalable parallel algorithms as possible. This paper takes a critical look at the major challenges involved in the development of scalable parallel algorithms and points to needs for compiler tool innovations to help address these challenges.

0. Introduction

With the industry-wide switch to multicore and manycore architectures, parallel computing has become the only venue in sight for continued growth in application performance. In order for the performance of an application to grow with future generations of hardware, a significant portion of its computation must be done with scalable parallel algorithms. It is therefore important to develop and deploy as many scalable parallel algorithms as possible.

So, why this is different from what the high-performance computing community has been working on in the past several decades? A simple answer is that the bulk of the prior work was along the lines of performing domain partitioning and use serial algorithms on each partition. While this approach has been very effective in allowing the science community to address increasingly larger problems using increasingly larger clusters, it has not made significant progress in parallelizing the algorithms used in each computing node. A good example is that many Intel Math Kernel Libraries are still only in sequential form when asked to solve a single large problem. What we need to do this time around is to introduce parallel algorithms into each computing node.

The next question is what makes the parallel algorithms for multicore and manycore processors challenging? As it turns out, the main challenges arise from the fact that the rate at which memory data can be accessed by the processor chips is much lower than the rate at which arithmetic operations can be performed on the chip. A closer look into the standard organization of the memory system further reveals that data accesses need to be highly regular in order for the data to be delivered to the processor at a rate close to the advertised rate. Today, these deficiencies are compelling parallel algorithm designers to resort to highly sophisticated data locality and regularization techniques, thereby drastically complicating their algorithms. A legitimate question is whether such deficiency will simply disappear as the technologies advance. Unfortunately, the trend is in the opposite direction, as I will explain in more detail below.

E-mail address: w-hwu@illinois.edu.

http://dx.doi.org/10.1016/j.jpdc.2014.02.005
0743-7315/

1. Three important trends in computing platforms

In general, I see three important trends in technologies and computing platforms. First, the on-chip execution resource is growing faster than the off-chip memory bandwidth. While the on-chip execution resources have been growing with Moore's law, the off-chip memory bandwidth has been held back by the slow evolution of DRAM architecture and the monitory and energy cost of I/O pins of chip packaging. We are at the point where about eight arithmetic operations need to be performed for each byte of data being brought on chip in order to fully utilize the execution capacity of manycore chips. This is in sharp contrast to the long-held rule of one floating-point operation for each byte of data in the High-Performance Computing (HPC) community. Unfortunately, the gap is expected to increase rather than decrease in future generations of computing devices. The reason is that the cost of increasing off-chip memory bandwidth is increasing much faster than the cost of increasing on-chip execution resource. This compels algorithm designers to keep as much data in the on-chip memory as possible for re-use, both across parallel threads and within each thread. Unfortunately, this is an increasingly challenging endeavor due to the next important trend.

The second trend is that the amount of on-chip memory is growing slower than the amount of on-chip execution resource. Unfortunately, traditional cache memories have shown to have diminishing return, where doubling the cache size does not cut the miss rate by two but rather by square root of two [4]. Due to the pressure to provide more computing power to demanding applications, vendors of both CPUs and GPUs are increasing on-chip execution rates faster than on-chip memory. This has incurred high pressure on algorithm designs. For many-core chips, the recent NVIDIA Kepler GK110 GPU has 256 kB of registers, 64 kB of configurable L1/Shared Memory, and 48 kB in each Streaming Multiprocessor (SMX). It also has 1536 kB of L2 cache that is shared among 15 Streaming Multiprocessors [9]. Among all 15 Streaming Multiprocessors, there is a total of 1904 kB of on-chip memory. While this looks like a large number, this amount of on-chip memory is shared by up to 30,720 active threads. This leave us with only 62 bytes of on-chip memory capacity per thread. In many applications, the most effective way of cutting down the use of memory bandwidth is to load tiles of data into the on-chip memory and have all parallel threads to perform their computation on these data from the shared memory. This approach becomes more difficult when the amount of available on-chip memory is small.

The third trend is that vectors are playing an increasingly important role in both memory accesses and arithmetic operations. In particular, vectorized memory accesses are becoming a key to high data delivery rate to manycore processors. This is a direct consequence of the increasing DRAM burst size required to achieve specified data delivery rates. Localized, vector-style data accesses are a well understood programming style that takes advantage of DRAM bursts. Programs that are well vectorized in memory accesses can perform an order of magnitude better than those that are not in today's manycore processors. The difference is expected to increase in the future, which will increasingly compel algorithm designers to regularize the memory access patterns of their algorithms.

2. Life is not fair in parallel computing

Fig. 1 illustrates the efforts to cover parallel portions of the applications with multicore CPU architectures and manycore GPU architectures. The center (core) of the cartoon peach picture represents the sequential portions of the applications. These sequential portions have been the target of modern instruction-level parallelism techniques that wring limited amount of parallelism out of these portions. The latency reduction mechanisms in modern CPUs, including cache memories, branch predictors, and data forwarding are important in preserving the instruction-level parallelism in these portions.

The orange flesh of the cartoon peach represents the data parallel portions of the applications. These portions typically process large data sets whose elements can be processed in parallel. Modern media and data analysis applications tend to have large data parallel portions. The traditional CPUs do not have sufficient amount of execution resources to harvest the data parallelism and achieve dramatic increase in performance. The bump coming out of the peach core depicts the SIMD or vector extensions to the CPU instruction set in order to exploit a much higher level of data parallelism in CPUs. The trend is to increase the width of the SIMD instructions to exploit a higher level of parallelism.

The meshed layer in the peach flesh represents the types of data parallel applications that can be efficiently covered by manycore architectures today. These applications typically are based on parallel algorithms that have high-level of data re-use and vector memory accesses, such as matrix–matrix multiplication, convolution, and particle–grid calculations. The arrows coming out of the layer represent efforts to develop new algorithms and/or to add hardware resources such as more on-chip memory to broaden the types of applications that can be effectively covered by manycore architectures.

The main point of Fig. 1 is that there is a large population of parallel applications that are currently covered neither by CPUs nor manycore processors. Although these applications are rich in data parallelism, they lack the regular accesses and data re-use needed for effective execution by the manycores. The dividing line between haves and have-nots is whether the algorithms used have data re-use and regular accesses. New efforts to design algorithms that exhibit more regular accesses and data re-use are needed for these applications to run effectively on manycore processors. In fact, I argue that the same efforts are also needed for these applications to run effectively on multicore CPUs with wide SIMD extensions.

3. Five levels of challenges in developing parallel algorithms

As we seek to design new parallel algorithms for multicore and manycore processors, we must understand that applications present at least five levels of difficulties. I argue that all these difficulties are equally present whether one programs a manycore GPU or a multi-core CPU. As long as one is trying to achieve high performance, energy efficient execution of highly parallel applications, these challenges must be met. Unfortunately, there is currently little technology in main stream compilers to help programmers to meet these challenges.

The most difficult level is that some applications do not have work-efficient algorithms that have sufficient parallelism. That is, we simply do not know how to solve the problem with a large number of parallel execution units without significantly increasing the computation complexity. Examples of such problems include finding shortest paths between two points in a graph and Delaunay triangulation of a mesh. In the shortest path problem, all known parallel algorithms with high levels of parallelism have significantly higher complexity, or equivalently much lower work efficiency, compared to the most efficient sequential algorithms. For large data sets, the increased computational complexity unfortunately results in so much work that the execution time of parallel algorithms can be slower than sequential algorithms. Unfortunately, large data sets are what make parallel algorithms compelling and thus make the work-inefficient parallel algorithms impractical.

In Delaunay triangulation, we have speculative algorithms that exhibit a small amount of parallelism due to the lack of known methods to divide the mesh into independent parts for large-scale