Performance Analysis of Current Parallel Programming Models for Many-core Systems

Yangjie Cao
School of Software Technology
Zhengzhou University
Zhengzhou, China.
caoyj@zzu.edu.cn

Baodong Wu, Yongcai Tao, Lei Shi
School of Information Engineering
Zhengzhou University
Zhengzhou, China.
{ieyctao,shilei}@zzu.edu.cn

***Abstract*—Recent developments in microprocessor design show a clear trend towards multi-core and many-core architectures. Nowadays, processors consisting of dozens of general-purpose cores are already available in the market, and with the rapid development of semiconductor technology and increasing computing demand, it will be very common to have a processor consisting of hundreds or even thousands of cores in the near future. This kind of processors is commonly referred as many-core processors. In the many-core processor era, however, exploiting all the advantages offered by these processors will not be trivial. In this paper, we focus on studying and analyzing several typical parallel programming models for many-core systems, and summarize the common performance-optimized problems faced by current parallel programming models.**

***Index Terms*—Many-core processor, Parallel programming model, Performance analysis**

I. Introduction

Recent developments in microprocessor design show a clear trend towards multi-core and many-core architectures. This radical shift in processor design results from diminishing returns of continuously increasing processor frequencies limited by manufacturing cost, power consumption, and heat dissipation problems, etc [1]. Nowadays, processors consisting of dozens of general-purpose cores are already available in the market, and with the rapid development of semiconductor technology and increasing computing demand, it will be very common to have a processor consisting of hundreds or even thousands of cores in the near future. This kind of processors is commonly referred as many-core processors. In the many-core processor era, however, exploiting all the advantages offered by these processors will not be trivial, and one of the great challenges is how to efficiently utilize the available on-chip computing power [2,3].

To solve the problems faced by current many-core processor systems, extensive and intensive research is carried out on parallel programming models [4]. A parallel programming model is a concept that enables the expression of parallel programs which can be compiled and executed. The advantages of a programming model is usually judged on its generality: how well a range of different problems can be expressed and how well they execute on a range of different architectures. Therefore, the implementation of a programming model can take several forms such as libraries, language extensions, or invent new programming model. A good programming model can provide an essential bridge between hardware and software, which means that high-level languages can be efficiently compiled and executed on specific hardware [5].

Generally speaking, parallel programming models can be broadly divided into two kinds: data-parallel programming model and task-parallel programming model. Data-parallel programming model focuses on solving data-independent problems and try to efficiently distribute data on different parallel computing nodes to achieve better performance. Data-parallel models are widely used in traditional high-performance computing. A most popular data-parallel programming model is MPI (Message Passing Interface) [6], which is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. Another popular data-parallel programming model is the MapReduce model, which is proposed for processing large data sets, and the name of an implementation of the model by Google [7]. In contrast to data-parallel programming model, task-parallel programming model, also called function-parallel programming model, expresses parallelism from the perspective of computations. In task-parallel programming model, programmers focus on decomposing the problem into sub-tasks, that can run in parallel. How parallel tasks are scheduled and executed on the architecture is the responsibility of the underlying runtime systems. Therefore, the task-parallel programming model provides more productivity due to such separation of concerns between task decomposition and scheduling methods. In recent years, task-parallel programming model is considered as essential tool to improve the productivity of generally parallel programming. There many task-parallel programming model are developed in the past decade, such as Java Fork-Join framework [8], Intel Threading Building library (TBB) [9], Cilk/Cilk++ [10,11], and OpenMP 3.0 [12].

II. The basic framework of task-parallel programming models

Task-parallel programming model is becoming increasingly popular in the past decade, and various task-parallel programming languages or libraries have been developed. When using a task-parallel programming model, programmers can easily generate parallel tasks without worrying about how these tasks are generated and mapped to the underlying hardware. From the programmers view, task scheduling and load balancing are transparently completed by the runtime systems. For example, The basic framework of the OpenMP programming model is shown as in Fig. 1.

As shown in Fig. 1, when using the task-parallel programming model, the programmer first write programs applying

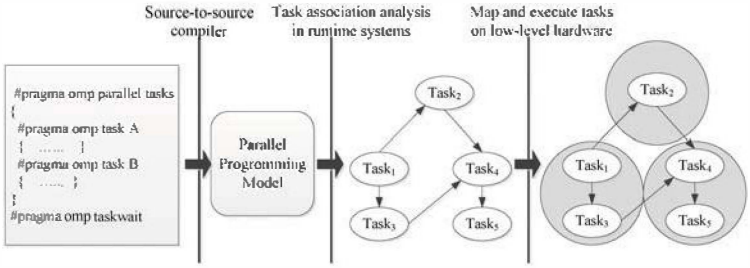

Fig. 1. The basic framework of task-based parallel programming model.

the specific parallel API provided by parallel programming model, and complete the codes for a specific application scenarios; After that, the original codes will be turned into intermediate code by the source-to-source compiler to meet the needs of specific programming model. Next, the runtime system, which provided by the programming model, conducts automatically the associated analysis of among generated tasks which are further represented as a directed acyclic graph (DAG). Finally, in the task execution stage, the scheduler of runtime system take charge of mapping tasks on low-level hardware to execute and load balancing are also automatically handled by the runtime system.

III. Performance analysis of typical parallel programming models

In this section, we first briefly introduce three typical task-parallel programming models: Intel Threading Building library (TBB), MIT Cilk and OpenMP 3.0. Then, we conduction intensive experiments to evaluate and analyze the performance of different programming models.

A. Cilk

Cilk is task-based parallel programming language proposed by Massachusetts Institute of Technology (MIT) [10]. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk's runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details like load balancing, synchronization, and communication protocols. Recently, Cilk++ is implemented based on traditional Cilk programming model, and it has been integrated into the the Intel parallel development environment as well as the latest version of GCC 4.7 [11]. The advantages of Cilk programming model are as follows: Firstly, Simplicity. Cilk only provides three keywords to write parallel programs, namely, cilk, spawn and sync. Secondly, it is easy to understand. The Cilk programs have serial semanticsthus this is more similar to traditional serial programming when writing the Cilk programs. Thirdly, high-level abstraction. The programmer programming only need to express the parallelism in the computation, without caring about the underlying task scheduling, load balancing and locality, which automatically and transparently complete by the runtime system. Finally, the work-stealing scheduling strategy applied by Cilk runtime system is very efficient when executing Cilk programs in practice.

B. TBB

Threading Building library (TBB) is a parallel programming library developed by Intel Corporation [9]. TBB, which is based on task parallel programming model, is mainly used the way of task instead of system thread to express parallelism. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner. Specifically, when using TBB, it does not require special languages or compilers. In TBB parallel libraries, task is a special kind of C++ objects, the object-oriented features of TBB tasks allows the TBB model to explore a variety of parallelism, such as DOALL, Recursive, Reduction even Pipeline. TBB is designed to promote scalable data parallel programming. It also fully supports nested parallelism, so you can build larger parallel components from smaller parallel components.

The scheduling algorithm applied in TBB runtime system is an improved work-stealing strategy. Unlike Cilk in which continuation is constructed upon task suspension, TBB does not have continuation support. When a task is suspended and the worker goes stealing, the old stack frames remain on the runtime stack. As a result, the suspended task can only be resumed by the same worker that suspends the task; TBB also restricts the stolen task to those deeper in the spawn tree than the suspended task in order to avoid stack over now in the worst case. These two restrictions will reduce the efficiency the work-stealing load balancing. In addition, TBB allows the programmer to manually create continuation tasks. However, this approach is essentially trading productivity for performance. In contrast to the Cilks work-first execution of all spawned tasks, TBB uses the help-first policy upon task creation.

C. OpenMP

OpenMP, which stands for Open Multiprocessing, was published by the OpenMP Architecture Review Board, a group of major hardware and software vendors [12]. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. The main aim of OpenMP is to simplify the programming of multi-platform, shared-memory, parallel programs by allowing insertions of pragmas into C/C++ code or special comments in Fortran code to instruct the compiler which parts of the code should be parallelized. There exist several compilers that support OpenMP, such as the GNU Compiler Collection, the Intel compilers, and IBM XL.

Currently there are two kinds of OpenMP standards widely used in practice: OpenMP 2.5 and OpenMP 3.0. The OpenMP 2.5 API mainly uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that will execute correctly both as parallel programs and as sequential programs. The recent OpenMP specification version 3.0 introduced a new task model to handle irregular and dynamic parallelism. In the view of OpenMP, tasks are units of work which are executed in parallel by threads. Tasks have their own data environment and can share data with other tasks. A disadvantage of OpenMP is that it lacks fine-grained control mechanisms to tie tasks to threads, and threads to cores.

D. Performance analysis

In this section, we conduct experiments to evaluate and compare the performance of different parallel programming models. The metric of speedup is applied to compare the performance of different applications. The experimental platform used is a Sun Fire X4600 M2 server, which equips 32 cores and 256G memory. The operating system is Linux and the kernel version is 2.6.28. The benchmarks used in our experiments are shown in Table 1, 2, 3.

TABLE I
THE DESCRIPTION AND INPUT SETS OF CILK BENCHMARKS

Benchmark	Description	Input sets
CK	Rudimentary checkers	-b 10 -w 13
Fib	Fibonacci numbers	46
FFT	Fast Fourier Transform	-n 2^{26}
LU	LU decomposition	-n 4096
Heat	Jacobi-type iteration to Solve a finite-difference	-g 10 -nx 4096 -ny 4096 -nt 500
Strassen	Multiplies two matrices	-n 4096

TABLE II
THE DESCRIPTION AND INPUT SETS OF TBB BENCHMARKS

Benchmark	Description	Input sets
Blackscholes	Option Pricing Model	65,536 options
Bodytrack	Body tracking	4 frames, 4,000 particles
Streamcluster	Online clustering	16,384 points per block
Swaptions	Pricing of a portfolio	64 swaptions

TABLE III
THE DESCRIPTION AND INPUT SETS OF OPENMP BENCHMARKS

Benchmark	Description	Input sets
Alignment	Aligns sequences of proteins	100 proteins
Health	Simulates a country health system	4 levels with 38 cities
Sort	A mixture of sorting algorithms	128M integers
SparseLU	LU factorization of a sparse matrix	500x7500
Strassen	Multiplies two n $\times$ n matrices	8192x8192

Using these benchmarks described in Table 1, 2, and 3, we conducted several experiments on the real many-core system, to evaluate and analyze the performance of current programming models. The experiment results are shown in Fig. 2, Fig. 3, and Fig. 4.

From the experimental results we can clearly see that, with the increase in the number of processor cores used by the application, the vast majority of applications are able to obtain different levels of performance speedup. Especially, when the number of cores in the system is less, most applications achieve nearly linear speedup with the increasing number of cores. When the number of core is further increased, however, no matter what kinds of programming models are used, the performance speedup of applications becomes more slower and even some application's performance speedup is experiencing a sharp downturn, such as Strassen in Fig. 2, Bodytrack in Fig. 3, and Sort and Health in Fig. 4. The experiment results indicate that the currently used parallel programming model such Cilk, TBB, and OpenMP, by themselves, do not scale well with great number of cores, particularly in the presence of many-core platforms.

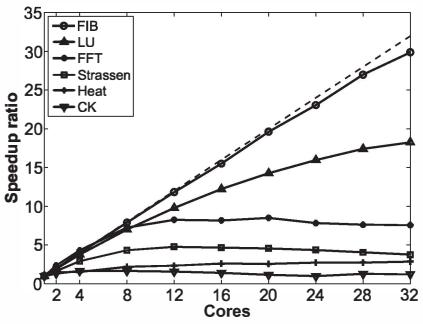

Fig. 2. Performance comparison of different Cilk benchmarks.

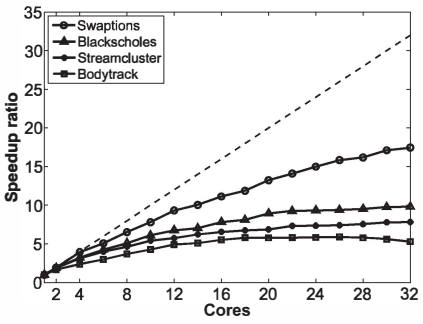

Fig. 3. Performance comparison of different TBB benchmarks.

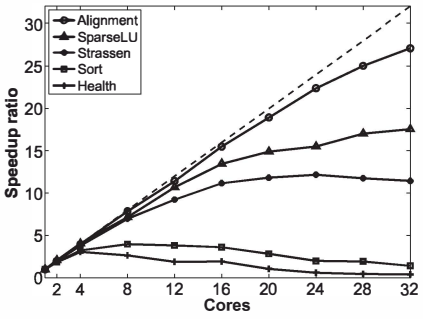

Fig. 4. Performance comparison of different OpenMP benchmarks.

IV. CONCLUSION

The prevalence of many-core processors is bound to drive most kinds of software development towards parallel pro-

gramming. To limit the difficulty and overhead of parallel software design and maintenance, it is crucial that parallel programming models allow an easy-to-understand, concise and dense representation of parallelism. Parallel programming models such as Cilk, OpenMP and Intel TBB attempt to offer a better, higher-level abstraction for parallel programming than threads and locking synchronization. Experiment results conducted in this paper, however, show that there are still many problems and inadequacies with traditional multi-core programming model in practical applications:

- Poor scalability. It is a typical requirement in most many-core runtime systems to explicitly or implicitly (via function calls) specify the number of cores to use for the execution of an application. As more cores are becoming available, many applications will start to experience diminishing returns with increased processor allocation. Without knowing the execution characteristic of the application on a particular hardware platform, simply allocating all available cores to the application may not ensure satisfying performance. As we can see in Fig.2, only a couple of applications have nearly linear speedup when increasing the number of allocated cores. The runtime of major applications
- Low utilization. The experimental results conducted in this paper show that the scheduling strategy provided by runtime system, can not make application take full use of their assigned processor resources, where the performance of each application is expected to improve according with the available increasing resources. With the progress of semiconductor technology and the development of the processor architecture, the future many-core platform will integrate more and more core in one die. Therefore, how to improve the efficiency of large-scale available cores becomes more prominent.
- Resource competition. Another issue faced by current programming model is that competitions for processor resources are unavoidable in current many-core runtime systems. Nowadays, it is very common for multiple users or applications to share a high-performance computing platform. Using current solutions by themselves, the performance may not scale well with increasing number of cores, particularly in the presence of concurrently running parallel applications.

In conclusion, the many-core architectures are the next big turning-point in the development of future processor architectures. It is not trivial, however, to efficiently utilize the available computing power offered by increasing number of cores. The currently used programming models by themselves do not scale well with great number of cores, which has been verified in this paper. Therefore, the search for efficient and productive parallel programming models for software developers has taken on a new urgency. In addition, how to greatly improve the efficiency and scalability of the underlying runtime system should be also studied intensively which allows to exploit the advantages offered by the many-core architectures.

ACKNOWLEDGMENT

This work is partially supported by China National Hi-tech Research and Development Program (863 Project) under the grants No. 2008AA01A202, 2009AA01A131 and Natural Science Foundation of China under the grant No.61073011,61133004, 61173039.

REFERENCES

[1] Borkar S, Chien AA. The future of microprocessors. Communications of the ACM, 2011, 54(5):67-77.
[2] Mack C. Fifty Years of Moores Law. IEEE Transactions on Semiconductor Manufacturing, 2011, 24(2):202-207.
[3] Cao YJ, Sun HY, Qian DP, Wu WG. Stable Adaptive Work-Stealing for Concurrent Many-core Runtime Systems.IEICE Transactions Information and Systems, 2012, E95D(5):1-10.
[4] Patterson D. The trouble with multi-core. Spectrum, IEEE, 2010, 47(7):28-32.
[5] Vajda A. Practical Many-Core Programming. Programming Many-Core Chips, 2011:175-211.
[6] Gropp W, Thakur R. Thread-safety in an MPI implementation: Requirements and analysis. Parallel Computing, 2007, 33(9):595-604.
[7] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1):107-113.
[8] Zambas C, Lujn M. Introducing aspects to the implementation of a Java fork/join framework. Algorithms and Architectures for Parallel Processing, 2008, 5022:294-304.
[9] Pheatt C. Intel threading building blocks. Journal of Computing Sciences in Colleges, 2008, 23(4):298-298.
[10] Blumofe R D, Joerg C F, Kuszmaul B C, et al. Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 1996, 37(1):55-69.
[11] Leiserson C. The Cilk++ concurrency platform. The Journal of Supercomputing, 2010, 51(3):244-257.
[12] Ayguad E, Copty N, Duran A, et al. The design of OpenMP tasks. IEEE Transactions on Parallel and Distributed Systems, 2008, 20(3):404-418.