Behavioral Analysis of Cilk Multithreading Programs Running on Multiprocessor Computer Systems

Iván A. David
Advisor: Jaime Seguel

Electrical and Computer Engineering Department
University of Puerto Rico, Mayagüez Campus
Mayagüez, Puerto Rico 00681-9042
ivan.david@ece.uprm.edu

Abstract

This paper describes the study about behavior of multithreading programming running over multiprocessor computer systems and extraction of data during multithreaded program executions for posterior analysis. The programming language selected for implementation of algorithms and study of work stealing scheduling method was Cilk, where we expect to obtain rough data about multithreaded computations, perform data analysis and finally obtained information from program execution must be represented graphically in a post mortem fashion to ease the programmer to visualize partially how the written code is processed.

1. Introduction

In programming languages and software development it is very important to know the behavior of codes during execution. This appears to be essential for many software engineering activities including program slicing, testing, debugging, reverse engineering and maintenance. [Zhao99]

Parallel programming (in all languages, not just logic languages) is still a highly experimental science in which the design of both user and system programs is undergoing simultaneous study. System implementors tune systems in response to experimental data from users. Furthermore, users try to understand the effects of implementor's decisions as well as the effects of variations in the parallel algorithms being executed by their programs. [Foster et al. 98]

Collecting timing statistics and measuring speedups is often insufficient for understanding the right meaning of these results. This is because in a sequential program, we know the sequence of events, whereas in a parallel program, not only is the precise sequence of events unknown to us, but it changes from one run to the next. In most cases, we have some expectation of the rough sequence of events, the overall "behavior" of the program. However, in the parallel case it is very difficult to deduce from readily available statistics whether the program actually behaved according to our expectations or not. A typical situation is one in which our intuition has told us that near-linear speedups should occur, but execution times (easy to measure) tell us that we are not getting them. It is often not at all clear what to do next. [Foster et al. 98]

This study will need both research and implementation work. The research will be into the nature of useful multithreaded parallel program behavior based on POSIX Threads; and implementation to design and establish mechanisms for incorporating application specific events visualization methods. We have observed statistical reports from Cilk after running programs and the expected amount of threads mismatch from live execution.

We expect basically discriminate which threads reported by Cilk statistical options, correspond to programmer expected model and its behavior during execution. We have that enough new information will be generated offering alternative ways to understand and decide over parallel programs implementations.

2. The Cilk Language

Cilk is an algorithmic multithreaded language. The philosophy behind Cilk is that a programmer should concentrate on structuring his/her program to expose parallelism and exploit locality leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Cilk's runtime system takes care of details such as load balancing and communication protocols. Unlike other multithreaded languages, Cilk is algorithmic in that the runtime system's scheduler guarantees probably efficient and predictable performance. [Leiserson97]

The basic Cilk language is extremely simple. It consists of C with the addition of three new keywords to indicate parallelism and synchronization. A Cilk program when run on one processor has the same semantics as the C program that results when the Cilk keywords are deleted. In addition, the Cilk system extends serial C semantics in a natural way for parallel execution. For example C's stack memory is implemented as a "cactus" stack in Cilk. [Cilk 5.3.2 Reference Manual]

Cilk is a simple extension of the C language with fork/join parallelism. Portability of Cilk programs derives from the observation, based on "Brent's theorem", that any Cilk computation can be characterized by two quantities: its work T_1, which is the total time needed to execute the computation on one processor, and its critical-path length T_∞, which is the execution time of the computation on a computer with an infinite number of processors and a perfect scheduler (imagine God's computer).

Work and critical-path are properties of the computation itself, and they do not depend on the number of processors executing the computation. In previous work, Blumofe and Leiserson designed a Cilk's "work-stealing" scheduler and proved that it executes a Cilk program on P processors in time Tp, where

$$T_p \le T_1/P + O(T_\infty)$$

This equation suggests both an efficient implementation strategy for Cilk and an algorithmic design that only focuses on work and critical path. [Frigo 99].

2.1 Efficient parallelization for ANSI C

Cilk's language support makes it easy to express operations on shared memory. The user can declare shared pointers and can operate on these pointers with normal C operations, such as pointer arithmetic and dereferencing. The type-checking preprocessor automatically generates code to perform these operations. The user can also declare shared arrays which are allocated and deallocated automatically by the system. As an optimization, also provide register shared pointers, which are a version of shared pointers that are optimized for accesses to the same page. [Bluemofe et al. 96]

2.2 Multithreading model

Cilk use a fully strict model for multithreaded computations, which is an adaptation of theorems of Brent and Graham on Directed Acyclic Graph (DAG) scheduling. A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-time tasks. The tasks of a thread must execute in this sequential order from the first (leftmost) task to the last (rightmost) task. In order to execute a thread, we allocate for it a chunk of memory, called activation frame, that the tasks of the thread can use to store the values on which they compute. [Blumofe et al. 94]

2.3 The Cilk runtime

The Cilk runtime system implements a efficient scheduling policy based on randomized work-stealing. During the execution of a Cilk program, when a processor runs out of work, it asks another processor chosen at random for work to do. Locally, a processor executes procedures in ordinary serial order (just like C programs), exploring the spawn tree in a depth-first manner. When a child procedure is spawned, the processor saves local variables of the parent on the bottom of a stack and commences work on the child. (Here, we use the convention that the stack grows downward, and that items are pushed and popped from the "bottom" of the stack.) When the child returns, the bottom of the stack is popped (just like

C) and the parent resumes. When another processor requests work, however, work is stolen from the top of the stack, which is, from the end opposite to the one normally used by the worker. [Cilk 5.3.2 Reference Manual]

3. Scheduling Schema

An execution schedule for a multithreaded computation determines which processor on a parallel computer executes which tasks at each step. An execution schedule depends on the particular multithreaded computation and the number of processors in the parallel computer. In any given step of an execution schedule, each processor executes at most one task. During the course of its execution, a thread may create, or spawn, other threads. Spawning a thread, where spawned threads to be children of the thread that did the spawning, and a thread can spawn as many children as it desires. In this way, threads are organized into an activation thread hierarchy implemented over a Deque structure due to allow insertion and extraction operations at top and bottom of itself. [Blumofe 94]

3.1 Execution measurements

Using the notion of parallelism, which is defined as $\overline{P} = T_1 / T_\infty$. The parallelism is the average amount of work for every step along the critical path. Whenever $P << \overline{P}$, that is, the actual number of processors is much smaller than the parallelism of the application, we have equivalently that $T_1 / P >> T_\infty$. Thus, the model predicts that $T_p \approx T_1 / P$, and therefore the Cilk program is predicted to run with almost perfect linear speedup. The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs over the entire range of possible parallel machine sizes. [Cilk 5.3.2 Reference Manual]

3.2 Pthreads implementation

Threads are often called *lightweight* processes and while this term is somewhat of an over simplification, it is a good starting point. Threads are cousins to UNIX processes though they are not processes themselves. In UNIX, a process contains both an executing program and a bundle of resources such as the file descriptor table and address space. Thus a thread is essentially a program counter, a stack, and a set of registers; all the other data structures belong to the task. [Zalewski97]

4. General Project Steps

Considering implied compilation options by Cilk language, to get statistical information about program execution, taking care for wall clock time, time elapsed, total work, total cumulative work, critical path and pallelism, getting detailed information about each of obtained results.

Get intermediate real C language code generated by the Cilk compiler during pre-compilation stage from Cilk user source code; this sequence of code will be modified implementing a new layer where new functions will be added to ease the generation of events threads measurements. Acquired information must be delivered to an automatic model to export data according execution real time and total work generating statistical information into a standardized format.

Design a graphical model where the user can to visualize the program execution respects time, threads and processors units involved during process; this will be implemented into well known graphical tool or developing a prototype graphical tool.

4.1 Implementation model

To get enough information about Cilk behavior we have performed a dissection over the open source Cilk Package; we observe that was not enough and we have reoriented to intermediate C language code generated during compilation of Cilk programs.

We have developed a well structured framework preserving the Unix/Linux file system style where little applications can deliver specific information about current hardware and operational system specifications and system variables. This

framework must be used by the users to store their own programs and get their executions results.

Using scripting languages like Bash, Kornshell and Ruby we have done a set of scripts to set some needed system variables and separate the different stages of Cilk programs compilations by means of undocumented flags and options obtaining C files, C files with libraries inclusions, Assembly code files and object files, and finally executable files.

The intermediate C language source file includes a massive number of declarations, data structures, functions calls and numerously tags to be used by Cilk runtime, we are currently studying what means each of them.

5. Conclusions

The comprehension of behavior of multithreading programming appears as a very interesting topic at present, when multiprocessors systems and clustering systems offers more flexibility and expandability. the study of languages with multithreading capabilities provides us a wide range for exploit efficiently the multiprocessing and parallel programming.

After evaluating the Cilk package source code and studying detailed statistical results, the next step has been to get specific execution measurements. We have decided to implement as solution to this problem the modification of precompiled code inserting specific function calls and appropriate identifiers for every read or write access and generate a log file taking care to cause a minimal impact on the performance of program.

Getting multithreading events and their interdependencies during execution respects the time; offer us the opportunity of implementing a visual component to display mechanism that promotes reconstruction of the sequence of events and an understanding of how it was caused by the program specification bring out to a static or dynamic graphical model; of course implementation of such a program currently involves decisions about graphics languages, window systems and appropriate representation.

References

[Zhao99] Zhao, J. Multithreaded dependence graphs for concurrent Java programs. Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on, 1999. 13 -23

[Foster et al. 98] Foster, Ian., Lusk, Ewing., Stevens, Rick. Performance Visualization (a white paper) Mathematics and Computer Science Division. Argonne National Laboratory. 1998

[Leiserson97] Leiserson, Charles E., Plaat, Aske. Programming Parallel Applications in Cilk. MIT Laboratory for Computer Science, Cambridge, MA, USA. 1997.

[Cilk 5.3.2 Reference Manual] Cilk 5.3.2 Reference Manual. Supercomputing Technologies Group. MIT Laboratory for Computer Science. November 2001 [online]. Citing: The Cilk Language. Available from World Wide Web: (http://supertech.lcs.mit.edu/cilk)

[Frigo 99] Frigo,Matteo. Portable High-Performance Programs.. Ph. D. Thesis, MIT Department of Electrical Engineering and Computer Science. June 1999.

[Bluemofe et al. 96] Blumofe, R.D., Frigo, M.; Joerg, C.F., Leiserson, C.E., Randall, K.H. DAG-consistent distributed shared memory. Parallel Processing Symposium, 1996. Proceedings of IPPS '96, The 10th International, 1996. 132 -141

[Blumofe et al. 94] Blumofe, R. D., Leiserson, C. E. Scheduling multithreaded computations by work stealing. Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on , 1994

[Zalewski97] Zalewski, J. Pthreads Programming [Book Review] IEEE Concurrency [see also IEEE Parallel & Distributed Technology, Volume: 5 Issue: 1, Jan.-March 1997 85 -86