Crude Synchronization-Efficiency Queueing Model

Paul E. McKenney
Linux Technology Center
IBM Beaverton
paulmck@linux.vnet.ibm.com

ABSTRACT

Many people are still in the habit of thinking of individual machine instructions as having constant computational complexity. In fact, modern systems have decidely non-constant computational complexity for the atomic instructions that are commonly used to implement synchronization primitives. This paper presents a very crude (but effective) queueing model demonstrating how the varying overhead of atomic instructions can influence the performance of shared-memory parallel programs.

1. INTRODUCTION

The non-constant nature of atomic instructions on modern CPUs can be seen in Figure 1, which plots the time in nanoseconds to execute an atomic increment of a single global variable on a system containing recent x86 CPUs. As can be seen, the latency of an atomic-increment instruction is roughly 25 nanoseconds *times the number of CPUs*. This figure unmistakeably shows that we cannot take for granted that conventional global synchronization mechanisms based on atomic instructions will perform or scale well with increasing numbers of CPUs.

Instead, we need to carefully consider the synchronization efficiency of our designs. To this end, this paper derives a crude queueing model that approximates the synchronization efficiency of synchronization mechanisms that operate on a single global shared variable.

2. DERIVATION

This section derives a crude queueing model for the efficiency of synchronization mechanism that operate on a single shared global variable, based on an M/M/1 queue. M/M/1 queuing models are based on an exponentially distributed "inter-arrival rate" λ and an exponentially distributed "service rate" μ. The inter-arrival rate λ can be thought of as the average number of synchronization operations per second that the system would process if the synchronization were free, in other words, λ is an inverse measure of the overhead of each non-synchronization unit of work. For example, if each unit of work was a transaction, if each transaction took one millisecond to process, not counting synchronization overhead, then λ would be 1,000 transactions per second.

The service rate μ is defined similarly, but for the average number of synchronization operations per second that the system would process if the overhead of each transaction was zero, and ignoring the fact that CPUs must wait

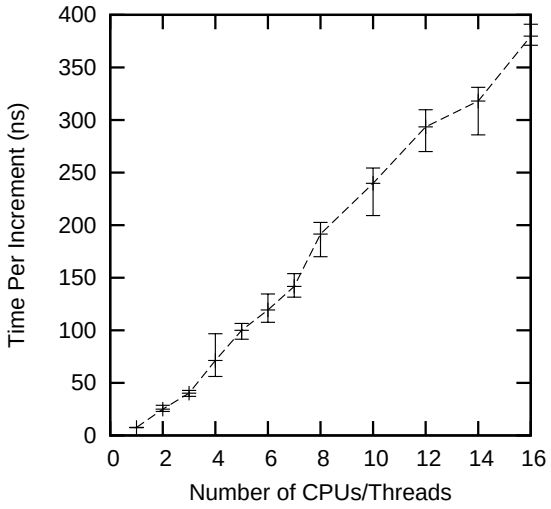

Figure 1: Atomic-Increment Overhead as Function of CPUs/Threads

on each other to complete their increment operations. Our example system from Figure 1 consumed about 25 nanoseconds per atomic increment. The value of μ is therefore about 40,000,000 atomic increments per second.

Of course, the value of λ increases with increasing numbers of CPUs, as each CPU is capable of processing transactions independently (again, ignoring synchronization):

$$\lambda = n\lambda_0 \tag{1}$$

where n is the number of CPUs and λ_0 is the transaction-processing capability of a single CPU. Note that the expected time for a single CPU to execute a single transaction is $1/\lambda_0$.

Because the CPUs have to "wait in line" behind each other to get their chance to increment the single shared variable, we can use the M/M/1 queueing-model expression for the expected total waiting time:

$$T = \frac{1}{\mu - \lambda} \tag{2}$$

Substituting the above value of λ:

$$T = \frac{1}{\mu - n\lambda_0} \tag{3}$$

Now, the efficiency is just the ratio of the time required to process a transaction in absence of synchronization to the time required including synchronization:

$$e = \frac{1/\lambda_0}{T + 1/\lambda_0} \tag{4}$$

Substituting the above value for T and simplifying:

$$e = \frac{\frac{\mu}{\lambda_0} - n}{\frac{\mu}{\lambda_0} - (n-1)} \tag{5}$$

But the value of μ/λ_0 is just the ratio of the time required to process the transaction (absent synchronization overhead) to that of the synchronization overhead itself. If we call this ration f, we have:

$$e = \frac{f - n}{f - (n-1)} \tag{6}$$

Figure 2 plots the synchronization efficiency e as a function of the number of CPUs/threads n for a few values of the overhead ratio f. As can be seen from this plot, synchronization mechanisms based on atomic manipulation of a single global shared variable cannot be expected to scale well on current commodity hardware.

3. CONCLUSIONS

Given current commodity hardware, developers should use synchronization mechanisms based on atomic manipulation of multiple global shared variables rather than on a single global shared variable. A common Linux-community slogan that corresponds to this advice is "lock data, not code", so that the number of atomically manipulated variables rises with the load on the system, avoiding queuing. Alternatively, lighter-weight synchronization mechanisms should be used where they apply.

Legal Statement

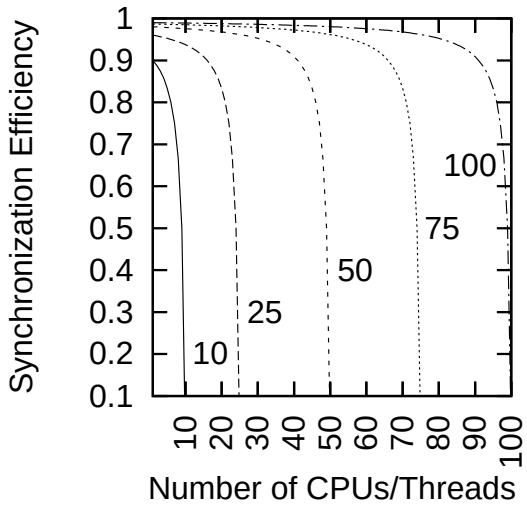

Figure 2: Synchronization Efficiency