Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering

Antoniou, Antonios - 9482
Kaimakamidis, Anestis - 9627

Parallel and Distributed Systems

Comparing pthreads, openCilk and Multicilk

1 Abstract

1.1 Symmetric Multiprocessing (SMP)

Symmetric Multiprocessing, or SMP, is a programming paradigm meant for distributing a program's workload across a machine's threads, instead of the sequential execution of instructions, in the expense of added source code complexity. The concurrent processing is realized by the use of both shared and private entities in the context of each thread. In this paper, we will be taking a closer look at three ways of setting up a piece of SMP software: pthreads, openCilk and Multicilk.

However, before getting into detail, it's crucial that we clear up some terminology and basics of SMP. Since some of the variables -or more generally memory regions- are shared between threads (can be interchangeably called "processes" as well), there is need for a consistent methodology utilized to make sure that both access to and processing of them without being led to race conditions.

1.2 Important terminology

• **Race conditions**: A program is susceptible to race conditions, or race hazards, when the order in which threads execute the given instructions can have inconsistent influence on the outcome of the program. There are two main ways of maintaining consistency:
• **Mutexes**: When a thread is about to update a variable -or entity- it obtains a lock designated for access to a certain memory space. When updating is over, the lock is released -or unlocked- again. While a mutex is locked, no other thread is allowed to update, or read a variable. In order to gain access to the mutex-ed variable, a thread must ask for it again, until it finds it an unlocked state.
• **Condition variables**: They work much like mutexes with a significant change in granting access to all threads. When a lock is unlocked after the successful update of a value, a signal is put out to all threads so that they can access it immediately. This means that no thread needs to persistently ask for permission to a variable until it is finally given to it.
• We wrap regions of the code that process an entity that's shared among processes around some mutex logic. This region is oftentimes referred to as a **critical section**.

2 Different multithreading schemes

Pthreads makes use of condition variables for managing critical sections, while openCilk is limited to mutexes. However, one important advantage of the latter is work stealing. More specifically, when a pthread is created, it is very strictly assigned a certain piece of workload. Work stealing introduces flexibility and dynamic scheduling to the work assigned to all threads. This is more easily understood using a cilk_for loop as an example.

When a thread is done with the iteration of the loop assigned to it, it checks if any iteration is free. If so, it immediately starts executing it. So, thread with ID 0 could be executing the body of a loop for iteration $i = 10$, while thread 1 could still be executing iteration $i = 4$.

On the other hand, Multicilk tries to take advantage of both software architectures. Cilk threads (often called cilks) are created, with each one of them having the permission to delegate some piece of their work to pthreads. This gives Multicilk the ability to use both work stealing and condition variables, depending on the region of the code the programmer needs to optimize. For that reason, we have constructed, roughly, the same algorithm for all three of the parallelization methods in order to compare their performances. Below are the specifics of each procedure.

3 Brief description of the problem

We will construct a problem that utilizes instructions for which both condition variables and work stealing makes sense, so that we can quantify their influence on the execution times. More specifically, our algorithm is a classic **producer-consumer** problem. We use a cyclical array that produces random seeds (we will call this number alpha). This seed is multiplied by random numbers to form an array of a constant user-given size. At this point, the producer has completed their work. Now it's up to the consumers to take that array and perform a basic linear algebra operation on it. The operations we have programmed for the purposes of this project are axpy and a general sum of the elements of the array, sequentially, via pthreads and via openCilk.

3.1 Axpy

Having two arrays, x and y, axpy performs the operation:

$$y = \alpha \cdot x + y, \tag{1}$$

where α is a constant value given to the operation by the program.

3.2 Sum

Sum takes an array x and a similar α parameter and performs:

for `i in size(x)` **do**
  $sum = sum + \alpha \cdot x[i]$
end for

4 Algorithm implementation

4.1 Pthreads and openCilk

The program is written to work using a number of threads -we will call it n-. Let's suppose $n = 4$. For the first two implementations, this means $n^2 = 16$ threads will be utilized. All of them will be both producers and consumers. They will enter a for loop and produce-consume until a counter is reached. Every array that is produced and then consumed, increments that counter by one, obviously using condition variables or mutexes to maintain consistency and overall deterministic behavior. When the counter reaches a user-given constant, all threads are joined and the program returns.

4.2 Multicilk

For Multicilk, we took advantage of the ability for more detailed work assignment, while still maintaining some of the much-helpful concept of work stealing. So, n cilks are invoked, and start producing arrays. Each one of them can use n pthreads to consume their produced arrays. Again, on the completion of an arithmetic operation on an array, a counter is incremented, until a certain value is reached. Using the design above, we call the same number of threads as the simpler implementations, with a difference in the roles they take up.

5 Testing

5.1 Test parameters

The parameters given to the various resulting executables are:
• **n**: Essentially the root of the number of threads to be invoked, in the fashion that has been

mentioned above
• **Size**: The size of each individual array, in elements.
• **Cases**: Number of arrays that must be produced and consumed.
• **Alpha**: The constant float value, used to perform both axpy and sum.

5.2 Conducted tests

For the purposes of our research, we conduct tests for various sizes, number of threads, and number of arrays to be produced and consumed. More specifically, our main test consists of of **100 arrays** of **size 1000**. The **thread numbers** (n) are **2, 3, and 4**. We also use a test case for $n = 2$ and varying sizes. An extra to our research is the compilation, and testing of the **sequential counterpart** of the operations, so we can verify the scaling of the algorithm.