PARALLEL AND DISTRIBUTED SYSTEMS: COMPARING PTHREADS, OPENCILK AND MULTICILK

Antoniou Antonios

Aristotle University of Thessaloniki
9482

Kaimakamidis Anestis

Aristotle University of Thessaloniki
9627

1. INTRODUCTION

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[1]. 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[2].

3.1. Axpy

Having two arrays, x and y, axpy performs the operation:
$y = a \cdot x + y$
where, a is a constant value given to the operation by the program.

3.2. Sum

Sum takes an array x and a similar $a$parameter and performs:
for i in size(x) **do**
$sum = sum + a \cdot x[i]$
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.

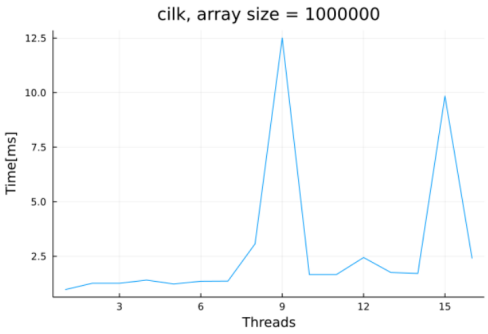
5.2. Conducted tests

For the purposes of our research, we conduct tests for various sizes, number of threads, and number of arrays to be produced and consumed.

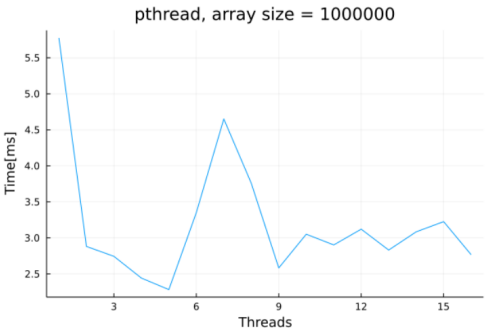
5.2.1. Pthreads vs openCilk

Our main test consists of of **10 arrays** of **sizes 1000000, 2000000, and 5000000**. The **thread numbers** (n) are **2, 4, 6, 8, 12, and 16**. We will observe the differences between the two schemes a little closer, for a **selected number of threads: 2, 4, 8, and 16**

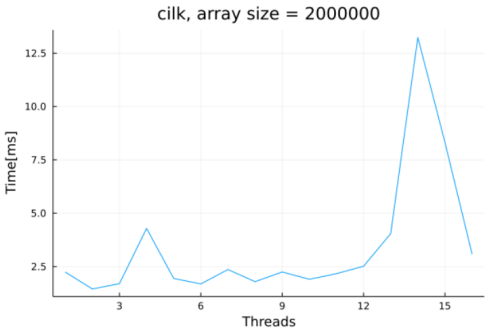
Fig. 1. openCilk vs pthreads



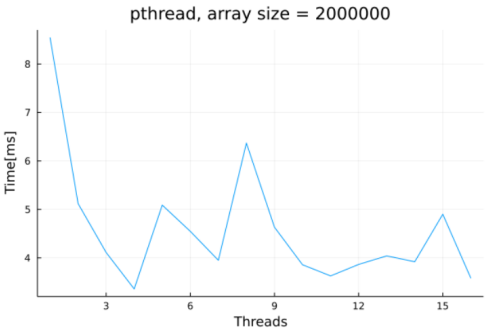
(a) openCilk - 1000000 elements



(b) pthread - 1000000 elements

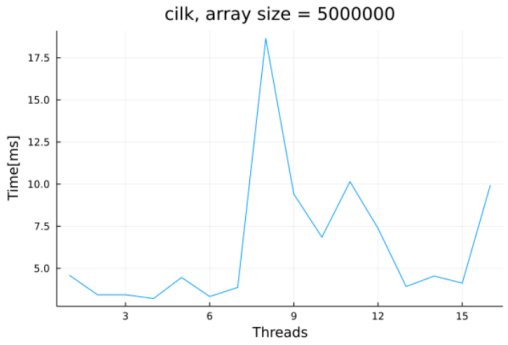


(a) openCilk - 2000000 elements

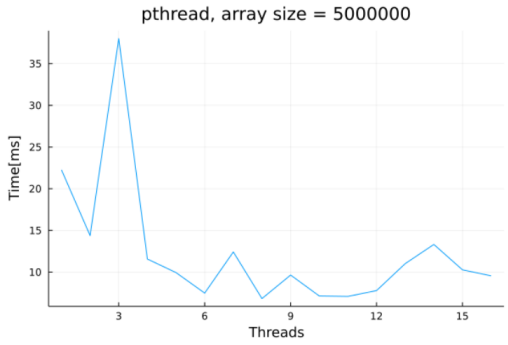


(b) pthread - 2000000 elements

Fig. 2. openCilk vs pthreads

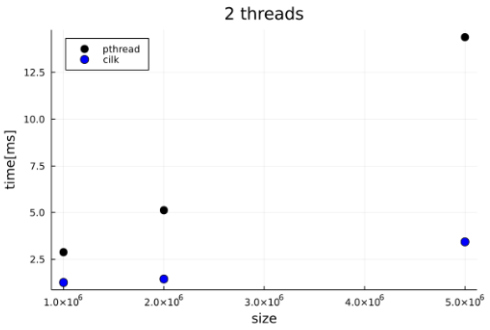


(a) openCilk - 5000000 elements

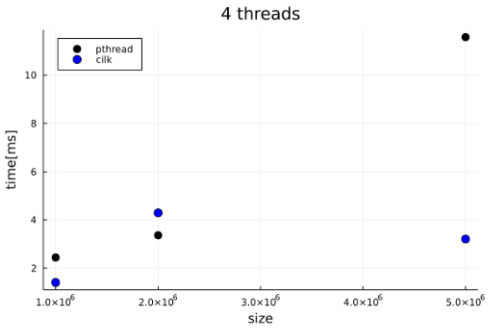


(b) pthread - 5000000 elements

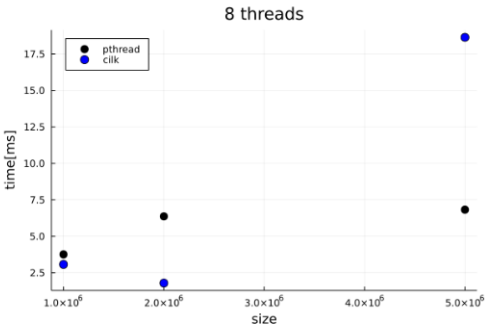
Fig. 3. openCilk vs pthreads



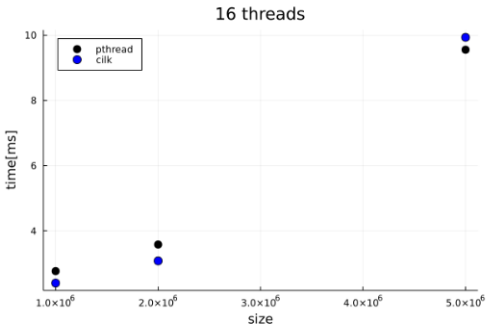
(a) Comparison for 2 threads



(b) Comparison for 4 threads



(a) Comparison for 8 threads

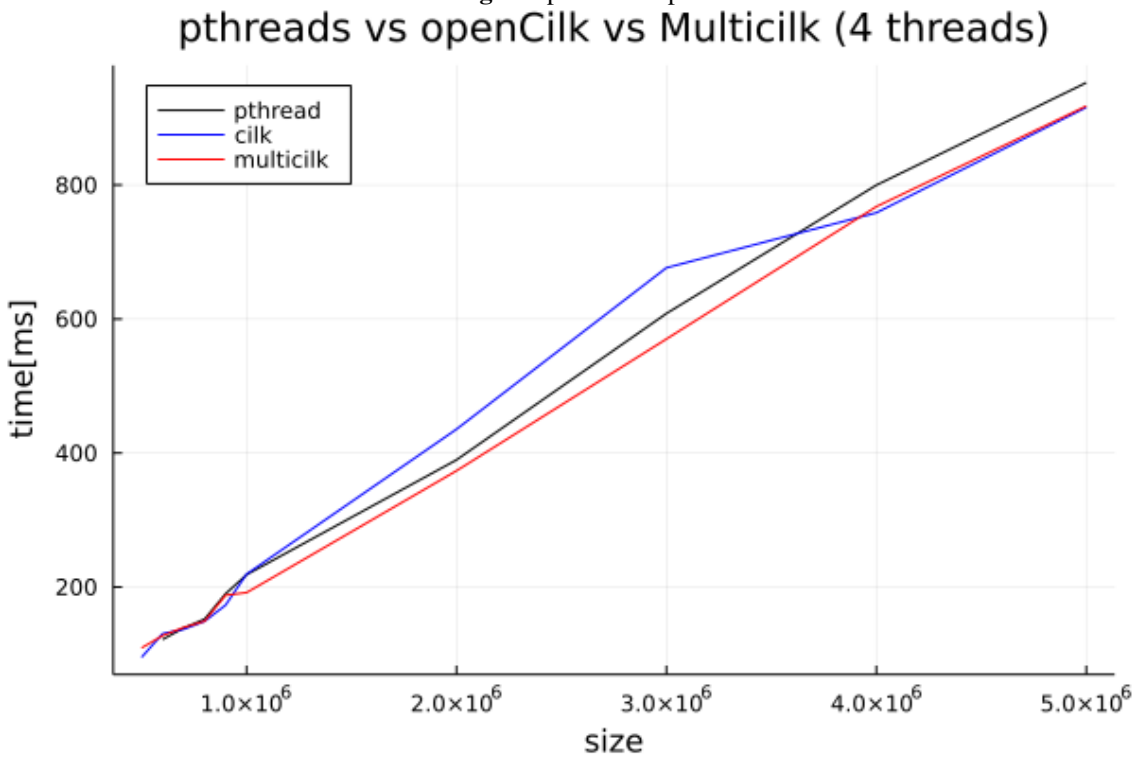


(b) Comparison for 16 threads

5.2.2. Benchmarks between the three schemes

The programs construct **10 arrays** of varying sizes, using $n = 4$, which was shown to be the best performing parameter in the analysis above. For that parameter, we use arrays of **size 500000, 600000, 700000, 800000, 900000, 1000000, 2000000, 3000000, 4000000, and 5000000.**

Fig. 4. openCilk vs pthreads vs Multicilk



6. CONCLUSION

The tests above show that openCilk's work stealing feature can offer more room for optimization, but less potential for deterministic behavior and predictability. All pthreads implementations seem to be scaling in a more predictable and linear manner, at least up until a number of threads that is in par with the specifications of the machine on which the tests were conducted.

However, on the first batch of tests, for a large enough size, when work stealing actually makes sense, it is easy to see that openCilk is able to outperform pthreads, where the workload is statically assigned and prone to imbalance.

Regarding the second batch, larger sizes and more threads than the ones natively built into the hardware lead openCilk to larger execution times, while more ideal conditions render openCilk capable of outperforming pthreads once again.

Last, but not least, the **Multicilk comparison** leaves much to be desired. Generally speaking, the conducted tests are the perfect way for the average programmer to realize that no scheme, paradigm or design is a panacea. Each one of them needs to be carefully engineered and fit into a problem that is fit for its advantages and hides its disadvantages. Multicilk needs a large number of threads, a large enough problem, and most importantly a vast amount of data communication between underlying threads. Even so, one could be a little skeptical about its ease and merits of use. This would be the reason why there is little to no bibliography found on the matter[3].

7. REFERENCES

[1] Ensar Ajkunic, Hana Fatkic, Emina Omerovic, Kristina Talic, and Novica Nosovic, "A comparison of five parallel programming models for c++," in *2012 Proceedings of the 35th International Convention MIPRO*. IEEE, 2012, pp. 1780–1784.

[2] Solmaz Salehian, Jiawen Liu, and Yonghong Yan, "Comparison of threading programming models," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 766–774.

[3] Sai Sameer Pusapaty, *Combining Task Parallelism and Multithreaded Concurrency*, Ph.D. thesis, Massachusetts Institute of Technology, 2022.