Features of Parallel Programming Languages

Stephen J. Plautz

The University of Alabama

CS403 - Programming Languages

12/14/2018

INTRODUCTION

Parallel programming is a programming style that is meant to achieve parallel execution on multicore processor hardware. This parallel execution is desirable because it allows large problems to be broken up into smaller problems, which can then be solved simultaneously via the use of multiple threads. Breaking up the problem in this way has significant advantages with respect to both economic efficiency and processing speed.

Over the last ten years, the rate at which clock speed has increased for processors has plateaued. While the number of transistors contained in modern processors has increased exponentially, the physical size of most processors has not. Moore's law has slowed down, and the physical limits of hardware spatial optimization are becoming increasingly relevant. There simply is a limit to the number of transistors that can be fit into such a small area without sacrificing structural integrity. As such, it has become less financially viable to focus primarily on the physical optimization of single core speed.

Fortunately, multicore computing has addressed these issues. Multicore computing is a computing method that relies on multiple processors working in conjunction, rather than an individual processor. These cores are placed on the same individual processor socket, allowing for faster communication between each other. Through this method, a dual core processor with two individually cheap cores each reaching 1.5GHz could theoretically outperform a more expensive individual processor reaching 2.5GHz, though in practice the speed would likely be quite similar. Most high-end processors currently being manufactured have at least four cores, allowing for parallel execution of four simultaneous threads. While these cores could each have more than one thread assigned to them for concurrent execution, the fact remains that they can each only execute one of the individually assigned threads at a time with the operating system deciding when to switch execution a different concurrently assigned thread. While individually, the clock speeds on these cores are lower than the more optimized single core processors, together they can achieve much faster execution times for properly written applications.

The key to harnessing this multicore computing power is to write applications with this processor structure in mind. Thus, the parallel programming paradigm has been devised. Parallel programming in applications is largely concerned with the assignment of execution threads to executing parallelizable problems (problems that can be broken down into smaller parts which can be solved individually), and the combination of these threads for output. Many different libraries for existing languages have been created to reduce the burden of the programmer from knowing very in-depth details about their operating system and scheduler to achieve parallelization. The duty of the programmer is instead to recognize

when a given problem is indeed parallelizable, and making use of either the built-in language constructs or library API's that provide for parallel programming. Additionally, they must be aware of the functionality provided by their specific language's implementation of parallelization, and the potential weaknesses or advantages of that language relative to other candidates for the job.

Throughout this paper I will be going over some of the different languages that have language constructs for implementing parallelization in an effort to compare their respective merits and shortcomings. The languages I have selected include Haskell, Java/MPJexpress, and C/Cilk. I will first provide a description of these languages and the way that they implement parallelization. I will then be comparing these languages according to their ease of use, the quality and completeness of their libraries, and the weaknesses or difficulties associated with the language's implementation.

HASKELL

Haskell first appeared in 1990 as the result of committee decision to standardize current existing functional languages into a consolidated common language that could serve as a basis for future research in functional language design. Haskell is a lazy, non-strict, pure functional language that featured the very first type class system. Being lazy and non-strict, Haskell has the capability of call-by-need evaluation, allowing it to reduce the running time of certain functions, as well as the ability to define infinite lists, one of the main features of Haskell. Also, Haskell provides for type classes, which were initially devised as a means of implementing overloaded arithmetic and equality operators, however their uses have increased greatly over the years.

It is particularly interesting to view Haskell through the lens of parallel programming because it is a pure functional language. This means that functions written in Haskell have no side effects, unless intentionally designed to do so. Haskell implements two forms of parallelism, Pure Parallelism (provided in the Control.Parallel library), and Concurrency (provided in the Control.Concurrent library). The distinction between these methods of parallelization with respect to Haskell is that pure parallelism speeds up the non-I/O parts of the program, specifically computations, which are assigned to multiple processors, and concurrency can be used for parallelizing a wider variety of things that may produce side effects, including I/O. While pure parallelism in Haskell is guaranteed deterministic with no race conditions or deadlocks, concurrency is not free of these concerns, and results from concurrent threads are combined non-deterministically. The relationship between this

original purity, and the necessarily impure nature of concurrency's nondeterminism is particularly interesting.

By using parallelism, performance gains can be made on both programs that do and do not make use of concurrent design. Haskell programs can be run in parallel on SMP's, or symmetric multiprocessors, and whether concurrent or not parallelism will use multiple processors. This brings about the distinction of pure parallelism and traditional parallelism. Pure parallelism will not make use of concurrent program design, while traditional parallelism will. In other words, when one encounters the term "pure parallelism" they could alternatively substitute it for "deterministic parallelism".

There are a few ingredients to parallelize a regular Haskell program. The first is to compile the program with the -threaded flag as this compiles the runtime system with the ability to spawn OS threads. To run a Haskell program on multiple CPUs, the user must specify the RTS -N flag. The -N[x] flag allows the user to specify the number of cores that they would like the program to be run on, however omitting the flag will result in the runtime using as many threads as there are cores in the user's machine. Additionally, there are flags that will affect the way the runtime schedules threads on CPUs. The runtime will typically try to load balance via thread migration and work stealing, and these flags: -qa, -qm, and -qw; will either modify that migration pattern or prevent it depending on the flag. The final step is to expose the program's parallelism to the compiler, which can be done concurrently by using the concurrent package, or deterministically by using either the Control.Parallel library, Strategies, or the monad parallel library.

The parallel library specifies two constructs that allow the programmer to expose deterministic parallel execution to the compiler. These constructs are "par" and "pseq". The "pseq" construct is semantically similar to the "seq" construct, which guarantees that value a and b will be evaluated to weak head normal form before seq returns a value. Briefly, this evaluation is necessary to avoid stack overflows due to the lazy nature of the Haskell programming language. Rather than evaluating a fold expression that would reach all the way to the most extreme element before evaluating, and therefore forcing the stack to contain far too many "thunks" of unevaluated data while it is waiting to be processed, the seq command allows for these values to be evaluated and accumulated on the way. However, according to the Haskell user guide: "A note on evaluation order: the expression seq A B does *not* guarantee that A will be evaluated before B. The only guarantee given by seq is that both a and b will be evaluated before seq returns a value. In particular, this means that b may be evaluated before a. If you need to guarantee a specific order of evaluation, you must use the function pseq from the 'parallel' package." This introduces the primary difference between seq and pseq: pseq can enforce an evaluation order due to only

the first parameter being strict, as opposed to both parameters being strict in seq. Enforcing evaluation order is incredibly important for the purposes of deterministic parallelism for obvious reasons. If we want to be able to determine an output given some (non-commutative) inputs, we will need to be able to either predict or control the order of operations. As the type for the previously listed constructs may prove useful, they are included below (source- Haskell.org parallel programming library).

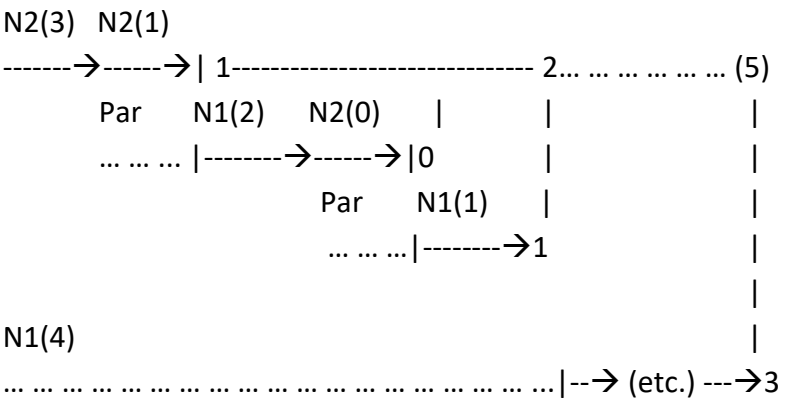
```
par:: a -> b -> b
pseq:: a -> b -> b
seq:: a-> b -> b
```

The par construct "sparks" the evaluation of A to weak head normal form, and then returns B. Sparks are the atomic unit of work designated by the Haskell runtime. They are not executed immediately, but are instead put into a circular buffer in FIFO order, waiting for an idle CPU to be assigned the spark, in which case it will be converted into a real thread. These sparks can be of varying size, but it is optimal to keep them similar in size. This is the way that available parallelism is spread among the CPUs by Haskell. According to the Haskell user guide: "When using par, the general rule of thumb is that the sparked computation should be required at a later time, but not too soon. Also, the sparked computation should not be too small, otherwise the cost of forking it in parallel will be too large relative to the amount of parallelism gained." These two tools, Par and pseq, are mostly used in combination with each other, or as building blocks for other libraries. Below is an example of a parallel Fibonacci that uses both:

```
import Control.Parallel
nfib :: Int -> Int
nfib1 n = (n1 'par' n2) 'pseq' (n1 + n2)
            where n1 = nfib1 (n-1)
                  n2 = nfib1 (n-2)
```

Notice that this version of fib is naive (no use of memoization or bottom-up recursion), however it will still benefit from deterministic parallelization given large enough N. In the above code, pseq n2 forces n2 to be evaluated first by the main thread, and par n1 sparks the evaluation of n1 to the thread queue, giving this branch of the problem to a new thread. This order does not however, practice the instructions provided for proper sparking listed in the user guide. This is because the granularity of work distributed will be too small,

and far too many sparks will be created. By this method, all main branches will evaluate all occurrences of n2 first, with all occurrences of n1 sparking new threads that will wait for their main branch to evaluate. An outline of the execution for fib 5 is provided below:



… (idle)
→(executing)
| pseq fulfilled

The method is not quite efficient enough, and could be improved. As evidenced in the diagram even for such small N, the program creates many sparks using the par command. The overhead of combining the massive number of sparks that will be spawned starts to become quite significant. Additionally, as can be seen in the diagram, a significant portion of time is being spent on synchronization. Indeed, the amount of overhead is so extensive for nfib1 that calling only a top-level calculation of a sequential fib in parallel (where n1 = sequential_fib(n-1), n2 = sequential_fib(n-2)) would be only slightly slower, even though this approach only uses two threads.

The key is to use the appropriate amount of parallelization. One important thing to notice about Fibonacci is that the amount of work is exponential. This means that fib [1..39] is approximately the same amount of work as computing fib[40]. This means that the top-level parallel version of fib has two evenly balanced threads, and very little overhead, which is why it works almost as well as the overly parallel nfib version above.

Thus, it falls to the programmer to engage as many cores as possible, with as little overhead as possible. To achieve this, a cutoff value is useful in order to control granularity. Given small enough n, sequential_fib can be used to avoid thread overhead, and given large enough n, we would like to engage as many of our cores as possible. The optimal cutoff point varies depending on the application, and as such it is useful to experiment with different

values, but for our purposes we will use 10. Below is an example program implementing this idea:

```
Cutoff = 10
pfib n
| n < cutoff = sequential_fib(n-1) + sequential_fib(n-2)
| n >= cutoff = (n1 'par' n2) 'pseq' (n1 + n2)
Where
        n1 = pfib (n – 1)
        n2 = pfib (n – 2)
```

This is the most efficient version of naive fib for Haskell. Because pfib delegates any N values low enough to sequential_fib, it avoids an extreme amount of thread overhead. Additionally, this follows the guidelines for sparking, as the granularity for any sparks is greater than or equal to 10, allowing for enough workload to be assigned to every new thread, with sequential_fib(9) being the smallest spark any one thread will take on. For all N above 9, the approach provided by nfib is used. With the problems from nfib being addressed, pfib works approximately four times as fast as the original nfib. Of course, solving fib with a bottom-up recursive solution would be better, but pfib as provided here could further improve even an optimized sequential fib (though demanding a significantly higher cutoff). It is evident that for deterministic, or "pure" parallelization, one cannot just throw more threads at code and expect it to run better. Very careful thought and a proper knowledge of the way that Haskell implements parallelization is required to adequately make use of the Control.Parallel library.

While deterministic parallelism in Haskell can be very complicated, Concurrent parallelism via the Control.Concurrent library follows a familiar forking syntax. The thread system in Haskell is very lightweight, and therefore has low overhead. Parallelizing a program that has already been organized concurrently is not very difficult, as it is simply a matter of enabling portions of the problem to be threaded. The main difficulty with concurrent parallelism is designing the program initially to be concurrent in nature, and then forking can easily be introduced to increase speed. Concurrent parallelism is a conceptual challenge for Haskell, as by definition functional languages should not have, and are not designed to have side effects. One of the ingenious solutions to this issue was to provide monadic syntax.

To achieve side effects in Haskell, programmers often make use of monads. Monads are a rather complicated subject, but essentially allow the programmer to supercharge function composition so that they have an added effect. There are different types of monads,

and each has a different context associated with it when applied, or “bound”, to a function composition. For example, the Maybe monad can provide the context “this computation could fail”, and the added effect to the composed function is that if any of the functions fail, the entire composition will fail. With respect to I/O, the context would be that “this computation has side effects and is impure”, and when bound to a composition has the added effect of “this result is impure, pass it along to the next function anyway”. Thus, the added effects of monads are explicit in the types. For example, a regular function that takes an int and returns an int will have type 1 below, however a function that takes an int and returns an int but also prints the returned int to stdout will have type 2 below.

1. Int -> Int
2. Int -> IO Int

Monads are composed of three elements: a type constructor, a binding operator, and a return function. Type constructors in Haskell are parameterized type definitions used with polymorphic types. To create a new concrete type, simply supply the constructor with one or more concrete types as in the Maybe monad’s type constructor shown in 1 below. What makes this new type monadic is the observance of supplying a binding operator and a return function. The “bind” operator (>>=) is a generic operator that is defined for any instance of the Monad type class. The bind operator is typically represented in the format (a >>= b), which could be interpreted naturally as meaning: “take wrapped value a, unwrap it, feed a to a function that returns a monad, and return a wrapped value”. This natural description can be seen in the bind operator’s type definition provided below. The type definition holds for all monads, with m representing any monad. The return function constructs a monadic value of type (m a) from a value of type a, effectively wrapping it.

```
return  :: a -> m a
(>>=)   :: m a -> (a -> m b) -> m b
```

```
1. data Maybe a = Nothing | Just a
2. return a = Just a
3. a >>= b = case a of
                Nothing -> Nothing
                Just x -> b x
```

With this definition in mind, using the Maybe monad in practice could look something like this (source computerphile), with safediv being a modified div function that either returns an Int or Nothing data type:

```
eval :: Expr -> Maybe Int
eval (Val a) = return a
eval (Div a b) = eval a >>= (\c->
                 eval b >>= (\d->
                 safediv n m))
```

This notation is abstruse, and could benefit from a different syntax to get to the core of what is happening, and thus the do syntax comes into play. The compiler will translate do syntax into a combination of bind and lambda expressions just like the one above, it is just a method of convenience for the programmer. Using do notation, the above program would instead look like the below block, increasing readability greatly:

```
eval:: Expr -> Maybe Int
eval(Val a) = return a
eval(Div a b) = do a <- eval c
                   b <- eval d
                   safediv n m.
```

With these concepts outlined, we are ready to talk about the specific elements of the concurrent library. It was important to go over the previous concepts, as these are the building blocks for implementing concurrent design in Haskell programs, and almost every concurrent Haskell program makes use of both monads and do blocks. Concurrency is enabled by default, and does not require any flags to operate, however, to make use of Haskell's concurrency extension the programmer must import the Control.Concurrent library. This library introduces functions for: forking and raising exceptions in threads, sleeping, synchronized mutable variables

Forking a thread can be done using the "forkIO" function. Different to the way that sparked threads work, the forked thread will start executing almost instantaneously, not having to wait for any pseq restrictions, only relying upon a processor idling for it to be picked up from the thread pool. There is also a kill thread and sleep thread. One important note is that the GHC runtime treats the main program thread differently than any forked threads.

When the main thread is completed, any forked threads still executing will be killed. It is important to provide measures against this happening.

Along with the forking option, IO threads have some features and metadata. Every thread is created with a ThreadId, which is essentially a pointer to the thread itself. Haskell provides some functions that raise exceptions within the thread. KillThread (tid) raises the ThreadKilled exception in the targeted thread. throwTo (exception tid) raises an exception in the targeted thread. Throwto will not return until the exception has been raised in the targeted thread, so one can be sure of its success upon return. However, because Haskell prevents threads from raising exceptions until they allocate memory (have thunked a memory block), Throwto can in some cases not be timely. For this reason, it is desirable to avoid long loops with no memory allocation as these will result in a loss of pre-emption. There is also a yield() function, which will force a context switch to any other currently runnable threads, and a threadDelay(int) function which will suspend the targeted thread for a given number of microseconds.

One of the most important features provided in the concurrent library is the Mvar construct. The MVar is basically a mutual exclusion box that holds one value at a time. When a thread is using the value held by MVar, it will be empty. If a thread tries to access an empty MVar via takeMvar, it will block until the MVar box is full again. If a thread tries to put a value into a full MVar box via putMvar, it will block until another thread takes the value out of the box. One thing that contrasts the MVar from other typical mutex devices is that the MVar holds the entire piece of data in a box rather than just being a primitive used solely to control concurrency on an arbitrary block. Basically, it is more robust as it can both be used as a lock but also hold some specific data to prevent it from being changed at the same time. The type signatures for all Mvar functions are provided below:

```
newMVar :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
isEmptyMVar :: MVar a -> IO Bool
```

As a matter of fact, these MVars are so important that they are used in place of explicit synchronization functions typically used in other languages. Below is an example program provided from the Haskell wiki that makes use of multiple monads, a do block, forkIO, and an Mvar (continued on the next page):

```
import Data.Digest.Pure.MD5
import qualified Data.ByteString.Lazy as L
import System.Environment
import Control.Concurrent
import Control.Monad (replicateM_)

main = do
  files <- getArgs
  str <- newEmptyMVar
  mapM_ (forkIO . hashAndPrint str) files
  printNrResults (length files) str

printNrResults i var = replicateM_ i (takeMVar var >>= putStrLn)

hashAndPrint str f = do
  bs <- L.readFile f
  let !h = show $ md5 bs
  putMVar str (f ++ ": " ++ h)
```

This is a concurrently designed parallel program for hashing an arbitrary number of files. As it can be seen, the do block not only allows for more succinct monadic expressions, but is also versatile enough to include typical commands. The files are collected into an array called files, and an empty Mvar called str is declared. The map monad (mapM_) is applied to the files array with the side effect of forking a new thread that executes the hashAndPrint function with a reference to the str MVar. This hash function reads the file, and then converts the results of applying the md5 hash function into a string literal h. Then hashAndPrint will attempt to put its resulting string expression into the str Mvar, only succeeding if it is currently empty, and blocking otherwise. Meanwhile the main thread, after mapping threads to hash each of the files, will attempt to print the results by calling takeMvar on the str Mvar length of files times, and then putting the result to stdout. This again, will succeed when some thread has successfully put its value into the str Mvar, and will block while str is empty. Because of the way the program has been structured, there is no risk of the main thread completing before all the children threads have been completed. This is because the main thread blocks until it has a new value that is guaranteed to be already evaluated, and will execute length of file times. It also avoids garbled output due to multiple threads printing to output, because all printing is done in the main thread.

JAVA/MPJexpress

Java is a widely used object-oriented, imperative programming language that is useful for almost any application. Java first appeared 23 years ago, developed by James Gosling of Sun Microsystems, and has often been advertised as a write once run anywhere framework. In other words, code that is compiled should be able to run on any platform that supports Java. Thus, one of the primary design goals of Java was portability, and to achieve this, Java compiles into an intermediary representation called bytecode which allows any instance of a Java virtual machine to run it. This originally quite unique concept is one of the main reasons that Java is so widespread, along with its extensive collection of high-level libraries.

Included in these libraries is MPJexpress, which is a thread-safe implementation of mpiJava that supports both distributed and shared-memory models. According to the MPJexpress windows user guide: “MPJexpress is a reference implementation of the mpiJava 1.2 API, which is an MPI-like API for Java defined by the Java Grande forum.” In plainer words, MPJexpress implements the guiding specifications provided by the mpiJava 1.2 API, and serves as standard which other implementations can compare to. The mpiJava 1.2 API is a Java API that describes the Message Passing Interface model, or MPI. The Message Passing Interface is a standardized model that outlines the capacity for applications to pass messages among different processes to perform tasks. This allows programmers to write parallel applications that are portable to all major parallel architectures. Because MPJExpress implements the mpiJava API, it can do everything that any typical MPI suite can do, and as such, it is more beneficial to speak in terms of MPI.

As MPI is a message passing interface, it has some predictable fundamental concepts. Sending and receiving are two methods outlined in MPI by which processes communicate. Send and receive operate according to the following description provided by mpitutorial: “First, process *A* decides a message needs to be sent to process *B*. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as *envelopes* since the data is being packed into a single message before transmission. After the data is packed into a buffer, the communication device is responsible for routing the message to the proper location which is defined by the process’s rank.” Along with these messages, processes are able to attach ID’s to the messages (called tags) to indicate which message type it is so that process B can select the message it currently needs, with all other transmitted messages being buffered until select by process B. The prototype for the MPI_Recv function (sourced from mpitutorial.com) is included below, as it has a couple more fields than its Send counterpart:

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

A few of the prototype fields require explanation. MPI-defined datatypes specify the structure of a message at a higher level. These datatypes are typically the corresponding language equivalent datatype preceded by MPI_. For example, an int representation in MPI is MPI_INT. A communicator encapsulates all communications among a set of processes. In this group of processes, each process is assigned a unique rank, and processes can communicate with one another by their ranks. This rank is similar to the common tid (threadID) concept. A process may send a message to another process by providing the rank of the destination and a unique tag to identify the message. Communications which involve one sender and receiver are known as *point-to-point* communications. Communications which involve a group of processes are collective communications. The status field is an MPI datatype that contains fields useful for describing the status of the current process.

The MPI framework is quite extensive, and thus the breadth of functionality covered within MPJexpress is quite extensive as well, indeed too broad to be covered adequately in this discussion. However, the learning curve to achieve basic parallel functionality via MPJexpress is not overwhelmingly steep. In fact, without even using any of the functions outlined above a simple parallel program can be written. To demonstrate, a program for threading the output of hello world is outlined below:

```
import mpi.*;
public class HelloWorld {
        public static void main(String args[]) throws Exception{
                MPI.Init(args);
                int size = MPI.COM_WORLD.Size();
                int rank = MPI.COM_WORLD.Rank();
                System.out.println("Hello, I am process " + rank + "out of " + size);
                MPI.Finalize();
        }
}
```

When run with an argument of 5, this program will print out: “Hello, I am process 1 out of 5”, followed by 2 out of 5, then three, etcetera on to 5 out of 5. This is not to be confused for simple iteration. Notice that there is only one call to println with no loops to be seen. This is because 5 different processes are being created, as stated in the output. When the MPI_Init function is called, it will create N (provided in args) new instances of the main process, each assigned to an available processor. This simple example reveals the beautiful nature of MPI. A set of imperative instructions can be prescribed, and each created thread will execute them with their own context. Once the MPI related functionality has been accomplished, MPI.finalize() ends the MPI context, allowing for only non-MPI commands to be run from that point forward. The examples can get much more complex; however, this is where we will leave the discussion.

C/CILK

Cilk is an extension to C that is concise and powerful. It was created in 1990 at MIT, and has since evolved to support more enjoyable syntax, as well as having a version ported for C++. Cilk makes use of the fork-join() idioms to express parallel loops. Any program making use of Cilk should both work on a machine with only one processor, and despite the Cilk keywords being removed.

Cilk introduces two new keywords, and one new method of iteration to the C language. These are cilk_spawn, cilk_sync, and cilk_for respectively. Cilk_spawn is a keyword that can be added in front of a recursive call to a function to parallelize it. It should be noted that according to the cilk_plus tutorial “cilk_spawn permits parallelism. It does not command it. cilk_spawn does not create a thread. It allows the runtime to steal the continuation to execute in another worker thread.” Cilk_sync is added before evaluation to make sure that all spawned children threads of the current thread have returned before returning a value. This means that children of parent threads to the thread calling cilk_sync may continue to run in parallel with the function calling cilk_sync. The cilk_for loop converts a regular for loop into a parallel for loop. This works for loops that have no dependencies between iterations, otherwise some combination of spawn and sync must be used. The cilk_for loop should not, however be confused with a for loop containing a cilk_spawn, as this creates only one source of parallelism, whereas a cilk for loop get translated to the compiler differently.

These keywords make it incredibly easy to parallelize certain problems. A classic example of using cilk for very easy parallelization is again the Fibonacci function. Included below is an example of parallelizing fib, as well as an example of the cilk_for to map a function f to an array:

```
int fib(int n)
{
   if (n < 2)
      return n;
   int x = cilk_spawn fib(n-1);
   int y = fib(n-2);
   cilk_sync;
   return x + y;
}

void loop(int *a, int n)
{
   cilk_for (int i = 0; i < n; i++) {
      a[i] = f(a[i]);
   }
}
```

For deterministic parallelization, Cilk has proven to be very useful. The examples above are indeed so simple conceptually to the point of being self-explanatory.

EVALUATION

Of the three languages provided, Haskell has by far the steepest learning curve. However, as we inspected with concurrent programming especially, this comes with the territory of bypassing the paradigm of the language. The syntax for deterministic parallelism is not overly complicated, however I found it bothersome to have to explicitly prescribe evaluation order due to the laziness of the language. Another problem with deterministic Haskell, as shown in an example, is that getting proper use of sparking takes a significant amount of understanding of the Haskell runtime, and an intuition for proper granularity. Using cutoffs either too small or too large can result in either enormous overheads and seq waiting times, or too large sequential evaluations. That being said, the threads created by Haskell are extraordinarily light, and under the right circumstances can achieve an edge over the threads of other languages due to faster context switching.

While I only covered Concurrent parallelization in Haskell, it is significantly more complicated than in the other two languages. This is because in addition to avoiding common concurrency issues such as starvation, deadlock, and race conditions, one must also first

learn how to express their program concurrently. This is no small task in Haskell, as the programmer must first establish an understanding of monads to efficiently create concurrent functions. That being said, once concurrent design has been established, simple introduction of forking syntax quite similar to cilk's cilk_spawn can be introduced. However, Haskell does not have an explicit synchronizing function such as cilk_sync, and must rely upon the programmer making proper use of the robustness of the Mvar construct. While I quite like the concept of the Mvar, it would take significant practice to use properly for concurrent programs of any sophistication.

The MPJexpress library was quite interesting to learn about. While it is necessary to learn MPI to make use of this library, it is not overwhelmingly difficult to do so. For deterministic parallelism, MPJexpress requires much less effort compared to Haskell. This is largely due to Java being an imperative language with explicit evaluation order. Additionally, debugging processes with MPJexpress would be much easier, as there is a wealth of metadata passed with each message send. The syntax of MPI is quite easy to follow for anyone who comes from an object-oriented background, and I would much sooner use this over Haskell for parallel programming. Additionally, the MPI framework provides an extensive amount of functionality for the programmer to make use of.

Cilk is perhaps one of the most elegant and simple solutions to deterministic parallelization available. The syntax is incredibly easy, only requiring a knowledge of the keywords provided. While the Cilk library is quite minimalistic, many useful functionalities can be created via use of either a cilk_for loop, or a combination of spawn and sync. While it was not overly difficult to learn basic parallelization in MPJexpress, Cilk simply could not be simpler. That being said, MPJexpress is designed for expediting distributed computing and does so quite efficiently. Achieving a similar effect with Cilk would be much more complicated, and with Haskell potentially unapproachable for beginners. In order of convenience, power, and flexibility I would rank Cilk, MPI, Haskell; MPI, Haskell, Cilk; and Haskell, MPI, Cilk.

NEW FEATURE

My new feature is an extension to CILK++ called cilk_filter. Cilk_filter takes a Boolean function F, a vector A, a starting index, an ending index, a granularity size, and a reference to a result vector. It applies the function F to chunks of size N of the array A in parallel, returning a resulting vector B. This code can also be modified slightly to produce cilk_reject by changing if(f(a[i]) to if(!(f(a[i])). The syntax for using cilk_filter would simply be the same as those used

for all other cilk operations in C++. Below is some general code that demonstrates how it should be implemented:

```
#include <iostream>
#include <vector>
#include <cilk/cilk.h>

using namespace std;

void cilk_filter(bool (*f)(int), vector<int> &A, int start, int end, int
chunk, vector<int> &B)
{
    //if we have reached the desirable granularity
    if(end - start <= chunk){
        for(int i = start; i < end; i++){
            if(f(A[i])) B.pushback(A[i]);
        }
    }
    else{
        int middle = start + (end + start)/2;
        cilk_spawn filter(f, A, start, middle, chunk, B);
        filter(f, A, middle, end, chunk, B);
        cilk_sync;
    }
}
boolean f(int x)
{
    if(expression x)
        return true;
    return false;
}
int main()
{
        vector<int> original;
        vector<int> result;

        cilk_filter(f, original, 0, original.size(); 200; result);
        for(int i = 0; i < result.size(); i++)
            cout << result[i] << ', ';

        return 0;
}
```

FEATURE EVALUATION

Like everything else in Cilk, my addition is quite simple. It achieves its purpose with almost no learning curve. All the programmer would have to do is to be aware of its existence, and pass the Boolean function pointer of their choice into the filter function. While the implementation illustrated could be more general with regards to data types, it was implemented with integers for simplicity. The feature provided is perhaps not as fundamental as some of the other concepts, such as an Mvar or Communicator, but it provides some functionality to the user that is very straightforward, abstracting away any concerns of improper parallelization, and is in my mind worthwhile.

CONCLUSION

There is surprising variety to the approaches of achieving parallelization, especially when differences between deterministic parallelization and concurrent parallelization are considered. Even within languages, new libraries are continually being developed to ease the burden of difficulty associated with parallel programming. Many of these libraries are the result of significant effort and forethought by groups of talented individuals, often experts in their fields of study. It is because of these efforts that specialized protocols such as MPI, and clever workarounds such as Monads came to be a part of the programming world. Harnessing the power of multiprogramming is essential for future application development, as the quantity of data to be handled gets larger and larger. Every year, millions more devices with multi-core processors are being manufactured, and unless applications are made to take advantage of this hardware, their potential can be wasted. With the convenience provided by libraries such as Cilk, the power of Frameworks such as MPI, and the flexibility of Monads and Haskell, programmers have an arsenal of tools at their disposal with which to achieve parallelization.

References

Anderson, O. (2014). *A tutorial on Parallel Strategies in Haskell.*

Baker, G. (2018, September 24). *Concurrent Haskell Fibonacci.* Retrieved from CourSys: https://coursys.sfu.ca/2018fa-cmpt-383-d1/pages/ExampleConcurrentHaskell

Bryan O'Sullivan, D. S. (2007). Real World Haskell. In D. S. Bryan O'Sullivan, *Real World Haskell.* O'Reilly Media.

Cilk. (2018, July 23). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Cilk

Computerphile (Director). (2017). *What is a Monad?* [Motion Picture].

Haskell (programming language). (2018, November 28). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Haskell_(programming_language)

Java (programming language). (2018, November). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Java_(programming_language)

Kendall, W. (2018). *MPI Send and Recieve*. Retrieved from MPI Tutorial: http://mpitutorial.com/tutorials/mpi-send-and-receive/

Mattson, P. P. (2014, November 13). *Why Haven't CPU Clock Speeds Increased in the Last Few Years?* Retrieved from Comsol: https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years/

Parallelism. (2014, December 24). Retrieved from HakellWiki: https://wiki.haskell.org/Parallelism

Shafi, A. (2014, July 18). *MPJ Express: An Implementation of MPI in Java.* Retrieved from mpj-express.org: http://mpj-express.org/docs/guides/windowsguide.pdf

Team, T. G. (2011). The GLorious Glasgow Haskell Compilation System User's Guide Version 7.0.3.

Tutorial: Cilk Plus Keywords. (n.d.). Retrieved from Cilk Plus: https://www.cilkplus.org/tutorial-cilk-plus-keywords

What is Parallel Programming? (2018). Retrieved from Best Computer Science Degrees: https://www.bestcomputersciencedegrees.com/faq/what-is-parallel-programming/