Experimental Multi-threading Support for the Julia Programming Language

Tobias Knopp
Section for Biomedical Engineering
University Medical Center Hamburg Eppendorf
Hamburg, Germany
tobias@knoppweb.de

ABSTRACT

Julia is a young programming language that is designed for technical computing. Although Julia is dynamically typed it is very fast and usually yields C speed by utilizing a just-in-time compiler. Still, Julia has a simple syntax that is similar to Matlab, which is widely known as an easy-to-use programming environment. While Julia is very versatile and provides asynchronous programming facilities in the form of tasks (coroutines) as well as distributed multi-process parallelism, one missing feature is shared memory multi-threading. In this paper we present our experiment on introducing multi-threading support in the Julia programming environment. While our implementation has some restrictions that have to be taken into account when using threads, the results are promising yielding almost full speedup for perfectly parallelizable tasks.

1. INTRODUCTION

Julia is a promising programming language that has been developed since 2009 by Jeff Bezanson, Stefan Karpinski, Viral Shah [1, 2] and various other contributors [3]. While Julia has a clear focus on technical and numerical computing it is very versatile leading to the development of packages for web applications [4] and graphical user interfaces [5].

Julia’s most remarkable feature is that it is as easy to use as most dynamic languages such as Matlab, Python, and R but still achieves a very high execution speed that is typically within a factor of 1.5 of well written C code. In scientific computing it is typical that one develops two tier applications where the time-uncritical parts are written in a dynamic – easy-to-use – programming language, while the time-critical part is written in a fast language such as C/C++ or Fortran. Popular examples are Matlab with its MEX interface and Python with ctypes or Cython [6]. Such two tier applications require much experience to get right and usually have not to be negligible maintenance burden. In Julia it is usually not necessary to offload time-critical parts in C. Thus, one can get much faster from research to production code as there is rarely a need to reimplement slow code paths.

The speed of Julia is based on some design choices in the type system that enable the generation of highly optimized machine code using just-in-time (JIT) compilation. The Julia reference implementation uses LLVM [7] for the generation of machine code. A core feature of Julia is multiple dispatch which allows to call different function implementations based on the type of the function arguments. This together with a parametric type system makes Julia very expressive without any performance implications. Interestingly, despite its Matlab inspired syntax, Julia has some concepts that are commonly used in the C++ community. Parametric types and the dynamic type system make generic programming the paradigm of choice during algorithm development. Similarly, the C++ standard library (STL) is also largely based on generic programming concepts by using the template feature of C++ [8].

Asynchronous programming can be done using tasks, which is Julia’s term for what is also known as co-routine or green thread [9]. For taking advantage of multiple processors or processor cores in CPU bound algorithms Julia provides distributed multi-process parallelism. However, there are various algorithms that are much easier to parallelize using native threads that run within the same process. Thus, especially as Julia is made for scientific computing, where multi-threading is a common tool to speedup the execution time of programs, it would be important that Julia supports writing multi-threaded applications. To date, utilizing multiple threads requires writing C code (e.g. using OpenMP [10]) that is called using Julia’s foreign function interface (FFI). But this two language approach is what Julia actually promises to make unnecessary.

The purpose of this paper is to exploit the feasibility of adding multi-threading support to Julia. Looking at other dynamic languages such as Matlab, Python, and R, one can see that adding multi-threading in dynamic languages can be a challenging if not unfeasible task as there is usually a tight coupling between the user code and the language runtime. For instance, Python has multi-threading support but the global interpreter lock (GIL) prevents any speedup that would be expected from utilizing multiple threads.

2. OVERVIEW

An overview of Julia’s internal structure is given in figure 1. Basically, Julia’s reference implementation is composed of three parts:

• Julia REPL (i.e. the julia main executable)
After sketching our attempt to make the critical parts of `libjulia` thread-safe, we will then take a look at the actual threading implementation.

As Julia is a cross-platform environment we use the threading facilities and mutex implementation from `libuv` that is already statically linked into `libjulia`. Hence, no additional dependencies are introduced by our multi-threading patch.

### 3.1 Code Generation

In contrast to other popular dynamic programming languages, Julia code is not interpreted but compiled to machine code during runtime. This concept known as just-in-time (JIT) compilation is also used in the popular statically typed languages Java and C#.

In particular, a simple Julia function will not interact with the runtime after it has been compiled. When a function is compiled for the first time the resulting machine code will be hold in a globally available method cache. On the second function call, no code generation will take place and instead the cached version of a function will be used.

Since access to the method cache is not thread-safe, we introduce a mutex to lock code generation. The mutex is named `codegen_mutex`. Using a regular mutex would, however, yield a deadlock when locking the code generation function. This is due to the fact that the code generation function can be recursively called. Consider for instance the following function `f`:

```plaintext
function f()
    g()
end
```

During code generation of the function `f` it is observed that no code for the inner function `g` is available. Thus, the code generation code path will be traversed a second time leading to a deadlock when using a regular mutex.

To solve this issue we use a recursive mutex. When entering the code generation function, it will first be checked if the mutex is already locked. If yes and if the current thread already has the lock, the code execution will be proceeded without locking the mutex again. If not the mutex will be locked. At the end of the code generation function, the mutex will be only released if it has been locked before.

### 3.2 Garbage Collection and Memory Allocation

Julia uses a simple mark and sweep garbage collection algorithm that is invoked at defined places in the Julia runtime. For instance, when acquiring new memory, the `gc` function will be typically invoked. In the following we sketch the places where we had to make the garbage collection and memory allocation code thread-safe.

#### 3.2.1 Memory Allocation

When acquiring memory in Julia this is managed by the garbage collector (GC). Some global values are hold in the GC such as the total number of allocated bytes. In order to make these memory accesses thread-safe we use atomic operations such as `__sync_fetch_and_add` that is available in...
### 3.2.2 Garbage Collection

Making the garbage collector work in a multi-threaded environment and especially making it efficient is a challenging task. Within this feasibility study we have decided to implement a simple solution that works effectively in various common situations but is not feasible in general.

The idea is to disable the garbage collector while multiple threads are running. This of course has to be taken into account when using threads because otherwise the memory consumption can increase drastically up to the point that a program is not usable anymore. However, when considering in-place functions and avoiding operations that create temporaries, disabling the garbage collector is a very effective way to prevent race conditions without major performance degradation.

One further complication considering the garbage collector in a multi-threaded environment is that the rooting of variables is performed on the stack using a global GC stack pointer. This is done using the JL_GC_PUSH and JL_GC_POP macros, which cannot be accessed in a thread-safe way. With the assumption that the garbage collector is disabled we can, however, simply disable variable rooting on user threads without any harm that a variable is cleared. In our multithreading patch [14] we have therefore adapted the JL_GC_PUSH and JL_GC_POP macros to be noops when the current thread is not the Julia main thread.

### 3.3 Threading Implementation

The actual thread support is implemented as a thin wrapper around the libuv threading functions. The C prototypes of the libuv threading functions are

```c
jl_thread_t* jl_create_thread
    (jl_function_t* f, jl_tuple_t* targs);
void jl_run_thread(jl_thread_t* t);
void jl_join_thread(jl_thread_t* t);
void jl_destroy_thread(jl_thread_t* t);
```

The `jl_create_thread` takes as argument a Julia function `f` that will be called within the native thread. Furthermore, `jl_create_thread` takes the actual function arguments that will be passed to `f` as a tuple (a native Julia container type). Within `jl_create_thread`, the function `f` will be precompiled, which partially generates code for `f`. Inner Julia functions being used in `f` will be lazily compiled when `f` is actually called for the first time. The `jl_run_thread` function starts the thread, `jl_join_thread` joins the thread, and `jl_destroy_thread` cleans the memory up that has been allocated in order to hold the `jl_thread_t` struct.

The C Julia threading functions are called from Julia using `ccall`. Within Julia it is important that the garbage collection is disabled using `gc_disable` before the threads are created. Once the threads are finished, garbage collection can be enabled again using `gc_enable`. As a simple interface to the threading functions we have implemented a function `parapply` that applies a function concurrently over a predefined range. The range is split into various subranges and the function is run on multiple threads with distinct subranges. Enabling and disabling the GC is automatically handled by `parapply`. By default, `parapply` uses as many threads as available CPU cores.

### 3.4 Exceptions

Julia has support for exceptions that stop the regular control flow of a program and propagate the error upwards in the call stack. One issue with threads in multi-threaded programs is that the main thread somehow has to be notified when an uncaught exception has been thrown in another thread.

In our multi-threading patch we implemented a thread-local exception handler that puts an unhandled exception in a thread-local `jl_thread_exception_in_transit` variable defaulting to `NULL`. The main thread can always check whether an exception has been thrown in a user thread by actively calling `jl_thread_exception` that takes as input the thread handle and outputs either the thrown exception or `NULL` when no exception has been thrown. When joining
a thread the exception will be automatically rethrown in the main thread. Hence, our high level threading function \texttt{parapply} will automatically rethrow exceptions that have been thrown in one of the created threads.

4. EXAMPLES
In order to test the performance of our multi-threading patch we have implemented a simple example program. The aim is to speed-up the vectorized \texttt{tanh} function that takes as input a vector and calculates the tangens hyperbolicus element-wise. Using the \texttt{parapply} function a multithreaded version of the built-in \texttt{tanh} function can be implemented as follows:

```julia
function tanh_core(x, y)
    for i in r
        @inbounds y[i] = tanh(x[i])
    end
end

function ptanh(x)
    y = similar(x)
    N = length(x)
    parapply(tanh_core, 1:N, x, y)
    return y
end
```

Here, \texttt{tanh\_core} is the kernel function that takes a range as first argument and calculates the tangens hyperbolicus of \texttt{x} element-wise and writes the results to \texttt{y}. The for loop runs over the subrange of array elements that is specified by \texttt{r}. The \texttt{@inbounds} macro is a performance hint for the compiler to disable bounds checking.

To take into account that the garbage collector is disabled while the threads are running, in the wrapper function \texttt{ptanh}, first the output array \texttt{y} is allocated using the \texttt{similar} function. Then \texttt{parapply} is called on the full range of the input and output arrays. The kernel function \texttt{tanh\_core} is passed as the first argument to \texttt{parapply}. Finally, in the last line of \texttt{ptanh}, the array \texttt{y} is returned.

5. RESULTS
In figure 3, results of applying \texttt{ptanh} and the single-threaded \texttt{tanh} to vectors of different length are shown. The code is run on a Macbook featuring an Intel Core 2 Duo P7350 processor with two cores. Hence, the \texttt{ptanh} function runs on two threads. Each call to \texttt{ptanh} and \texttt{tanh} is repeated 20 times and the fastest timing is taken.

As one can see, for small array sizes below 4000 elements, the single-threaded \texttt{tanh} function performs better, which is typical for multi-threaded applications as the thread creation / handling overhead amortizes only for larger problem sizes. For arrays larger than 4000 elements, the \texttt{ptanh} outperforms \texttt{tanh} and reaches a full speedup at an array length of about $5 \times 10^4$. For larger arrays, the speedup decreases again to about 1.75 at an array length of $10^7$. This might be due to non-optimal cache usage for larger arrays.

6. CONCLUSION AND OUTLOOK
We have shown in this paper that multi-threading in Julia is feasible and that full speedups can be achieved.

Our multi-threading patch is certainly not ready for inclusion into the main branch of Julia. One key issue that limits the application is that the garbage collector has to be disabled. It is therefore not feasible to spawn a long running background thread while proceeding work in the main thread. For the parallelization of computation intensive tasks where the main thread can wait for the worker threads to finish, our patch is already useful though. Here, one just has to take into account that memory is preallocated and no temporary arrays are created withing threaded code.

Making the garbage collection work in a parallel environment is the key to making multi-threading an official feature of Julia. As has been discussed in [15] using thread local GC heaps is one way to get to a multi-threading aware garbage collector.

In our implementation the \texttt{parapply} function has used as many threads as available CPU cores by default. In practice it would be better to have a central place where the number of currently running threads is taken into account and where the threads are appropriately scheduled.

From an interface point of view it would be desirable if the threading facility would not be restricted to callback functions as in our \texttt{parapply} implementation but would work on arbitrary code blocks. In C/C++ with OpenMP one can simply annotate that a certain for-loop should be parallelized and the code is internally rewritten in a form that schedules the work on different threads. Fortunately, Julia comes with a very powerful macro system that makes these thing feasible to implement. For instance in the Julia's Base module there is already a \texttt{@simd} macro that rewrites for loops in order to bring the code into a form that makes generation of SIMD instructions by the LLVM vectorizer possible.

7. ACKNOWLEDGMENT
We want to gratefully thank all Julia contributors for discussions on the Julia mailing list and Julia's issue tracker on Github. In particular, very helpful comments on multi-threading in Julia by Jeff Bezanson, Stefan Karpinski, Oscar Blumberg, Steven G. Johnson, and Tim Holy are acknowledged.
8. REFERENCES


