The repository for the code and talk given on this topic is [here](https://github.com/JamieMair/julia-for-research-with-hpc).

## What is Julia?

[Julia](https://julialang.org/) is a modern, general purpose, programming language. It was designed from the ground up for high performance, scientific programming. According to a [blog post](https://julialang.org/blog/2012/02/why-we-created-julia/) written by the creators, Julia was borne out of greed for a programming language which is as fast as C, yet easy to use like Python, MATLAB and R. Personally, I use Julia because it is incredibly easy to prototype new code for my research, while being flexible to execute my code across my home machine, my GPU or an entire cluster. All of this with minimum effort on my part!

What I want to show you today is a brief window into what’s possible in Julia, for very little investment. By the end of this article you will see how to write reusable code and take advantage of every level of parallelism, from the hardware parallelism of SIMD, to executing native Julia code on the GPU and scaling it across a cluster. The ease of use and flexibility of this parallelism is unmatched across any language and provides a very low bar to entry.

## Julia is fast

Julia has many advantages, when compared to other languages, however, the main claim that will interest the HPC community is speed. It claims to be as fast as C, so let’s put that to the test.

As an example through the rest of this blog, I will use the example of simulating a Monte-Carlo process, specifically a [random walk](https://en.wikipedia.org/wiki/Random_walk). While the process we will choose is simple, the basics are used in a wide variety of fields, and are common in HPC workloads.

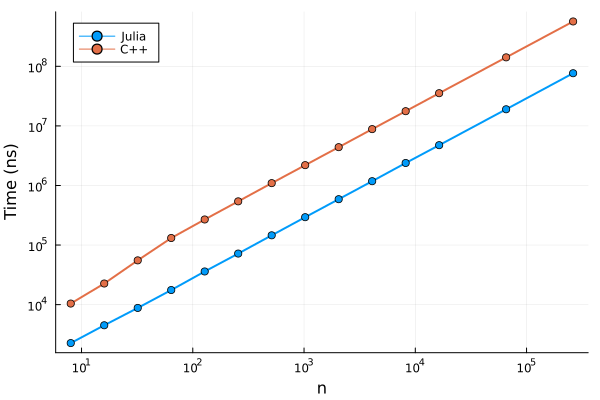
The code for running a random walk is as follows:

function simple\_monte\_carlo(n, T)  
 x = zeros(n)   
 for i in eachindex(x)  
 for t in 1:T   
 x[i] += Random.randn()   
 end   
 end   
 return x  
end

This function performs n random walks, all of length T. We can write some code in C++ that is roughly equivalent to see some empirical differences in performance.

double \*random\_walk(size\_t n, size\_t T, std::normal\_distribution<double> randn, std::mt19937\_64 rng)  
{  
 double \*x = new double[n];  
 for (size\_t i = 0; i < n; i++)  
 {  
 double x\_t = 0.0;  
 for (size\_t t = 0; t < T; t++)  
 {  
 x\_t += randn(rng);  
 }  
 x[i] = x\_t;  
 }  
 return x;  
}

This implementation in C++ is not optimal at all, as I am no C++ programmer, but this is meant to be indicative of what a typical new PhD student may write for their simulations. We find that the Julia version is between 4 and 7 times faster (depending on the size of n).



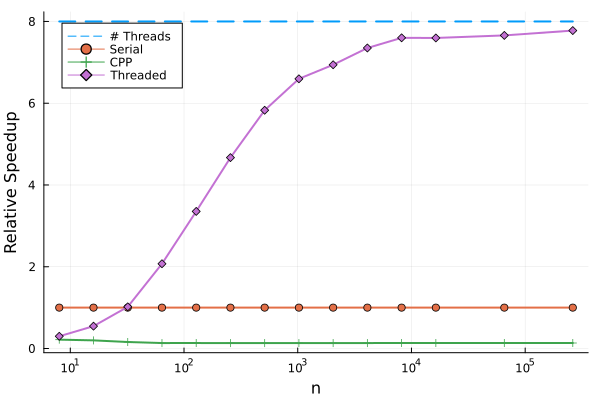
While Julia has the advantage here, I don’t think that Julia is inherently faster than C/C++. A much better C/C++ developer than me could make the C/C++ code perform similar to the Julia code. The main point I want to make is that you can achieve C/C++ level performance for minimal effort, while keeping your code simple and trusting Julia’s “Just in Time” compiler (the standard LLVM compiler) to produce some very fast machine code. Not all code will run quickly, but there is an entire section on [performance tips](https://docs.julialang.org/en/v1/manual/performance-tips/) which will let you achieve these C-like speeds.

## Multithreading in Julia

Almost every modern CPU has multiple cores, giving a huge amount of additional compute for tasks. Julia provides lightweight multithreading support natively in the language. The language provides many ways of parallelising your code, but here we’ll just use a macro to make this process a breeze. A macro is just code that writes other code, which lets you avoid a lot of boilerplate code. For executing a loop in parallel, we just add an @threads macro from the standard Threads library:

function simple\_monte\_carlo(n, T)  
 x = zeros(n)   
 Threads.@threads for i in eachindex(x)  
 for t in 1:T   
 x[i] += Random.randn()   
 end   
 end   
 return x  
end

This is essentially the same source code as above, with only a simple change in front of the for loop. We can add this to the performance graph:



We can see that this is pretty good performance out of the box. Unfortunately, we can see that threading doesn’t reach the theoretical maximum speedup, but there are techniques to [remedy this](https://github.com/JamieMair/julia-for-research-with-hpc/blob/00681690fb4dd4a5b83c2563616658be594a916d/main.jmd" \l "L163).

## Moving to the GPU

The modern GPU really excels at array based operations at scale, such as standard BLAS operations (matrix multiplications etc), which make up the backbone of a lot of numerical and scientific computing. Writing code for a GPU is notoriously difficult, making you resort to writing a custom kernel in C - a very high barrier to entry for beginners.

The good news is that Julia has a very successful package, called CUDA.jl, which allows you to write native Julia code that will compile directly to GPU code, which makes it easier than ever, so let’s take it for a spin!

Up to now, we have focused on using the standard Julia libraries, but there is a rich ecosystem of packages available. Adding a package is as simple as running the following code:

using Pkg  
Pkg.add("CUDA")

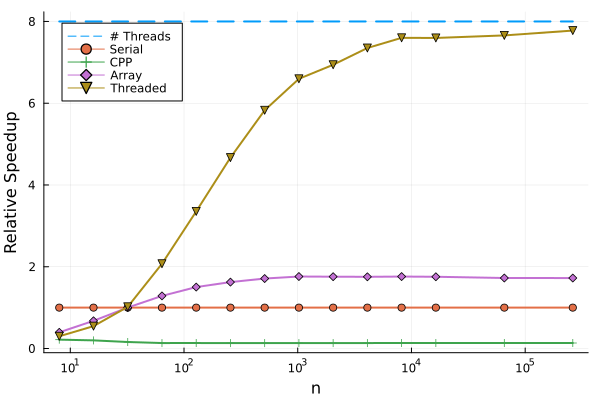
which uses Julia’s built-in package manager to download the CUDA.jl package. What makes this process painless is that when you use CUDA for the first time, it will download the CUDA libraries you need to get things working, instead of having to install CUDA manually (as you need to do in Python libraries like Tensorflow and PyTorch).

GPUs are very different devices to CPUs and have a slightly different programming model. The main difference is that scalar indexing of an array has really poor performance on a GPU, and so the CUDA.jl package errors when you try to do this. Instead, we can use Julia’s excellent “broadcasting” notation, which allows us to write concise “vectorised” code. This notation, also allows us to write *non-allocating* code, which means that we can reuse memory. This can make a huge performance difference. Here’s the same random walk code, rewritten to use array based notation and avoiding allocating memory:

function random\_walk!(x, T)  
 fill!(x, zero(eltype(x))) # Set all elements of the x array to zero  
 cache = similar(x) # Create a cache array of the same size and type as x  
 for t in 1:T  
 Random.randn!(cache) # Populate the cached memory with random numbers  
 x .+= cache # In-place element wise addition into x  
 end  
 return x  
end  
function simple\_monte\_carlo\_array(n, T)  
 x = zeros(n)  
 random\_walk!(x, T)  
 return x  
end

*Note that the use of the ! at the end of the function name is just notation that tells us that the function mutates the input arguments.*

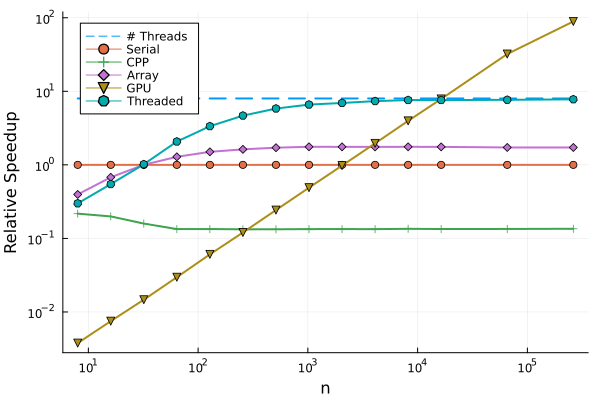
This code runs fine on the CPU with arrays, and is very similar to how someoene would write this in MATLAB or numpy (in Python). We can benchmark this function and add it to the graph:



Notice that this has slightly better performance than the serial implementation, as it can make use of hardware SIMD instructions. What is really powerful about this approach, is that we can directly re-use this code to run the code on the GPU:

using CUDA  
function simple\_monte\_carlo\_gpu(n, T)  
 x = CUDA.zeros(n) # Creates a GPU array of zeros  
 random\_walk!(x, T) # Same fn as before, but different type of `x`  
 return x  
end

There are no changes to our code! The only thing we changed was the type of our array. Since Julia has a Just-in-Time compiler, which specialises on the type of the input arguments, it knows to perform all of the array based operations on x on the GPU. We can see the performance below:



We can see that it is only really worth using the GPU for many elements in our array. If we need millions of calculations, the GPU can be around 100x faster than the CPU! We were able to get this huge increase in performance by only changing the input type to our function, and were able to reuse our original code!

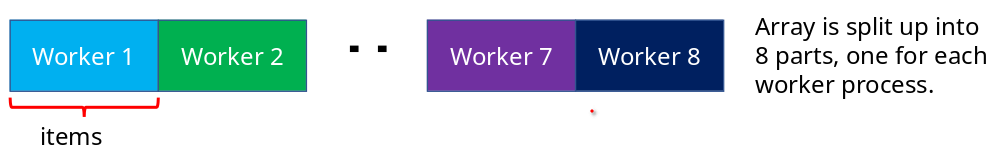
Here, we have really only scratched the surface of GPU programming, as you can always move onto writing your own kernels from scratch. For a more detailed review, check out the CUDA.jl [documentation](https://cuda.juliagpu.org/stable/).

## Scaling up to a Cluster

This wouldn’t be an HPC blog without talking about using a cluster. While it is certainly possible to use a single node with multithreading, Julia also has native support for multiprocessing which scales across multiple nodes. Julia has native support for multiprocessing (with the Distributed.jl standard library that comes with Julia), and support for schedulers like SLURM with ClusterManagers.jl. In order to connect all of the nodes, all we need to do is run the following code:

using Distributed  
using ClusterManagers  
  
# Connect to 8 other processes using SLURM  
addprocs(SlurmManager(8); exeflags=["--project"])

We could use the parallel primitives as described in the [Distributed.jl documentation](https://docs.julialang.org/en/v1/stdlib/Distributed/). But let’s reuse some of our previous code and take the array based programming model. There is a package called DistributedArrays.jl which gives us the DArray type which, as the name suggests, distributes the memory of an array in blocks to each process:

We would hope that we could copy the method of implementing the GPU extension:

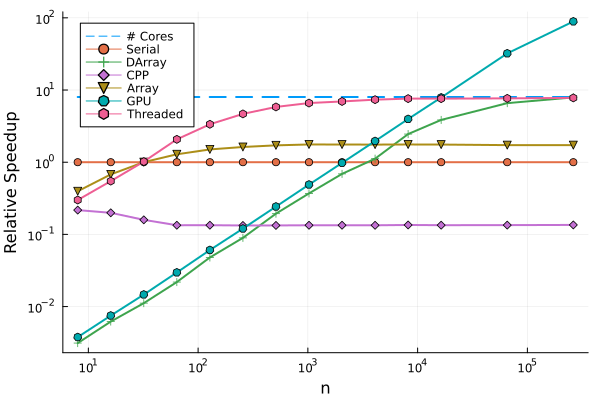
using DistributedArrays  
function simple\_monte\_carlo\_dist(n, T)  
 x = dzeros(n) # Creates a Distributed Array (DArray) of zeros  
 random\_walk!(x, T) # Same fn as before, but different type of `x`  
 return x  
end

Unfortunately, the above code does not work on its own, but needs to implement the Random.randn! function for the DArray type. But since our function is easily mapped to each block in the DArray, we can just write a specific implementation of our random\_walk! for this type, which we can do with a little help from the DistributedArrays.jl [documentation](https://juliaparallel.org/DistributedArrays.jl/stable/" \l "SPMD-Context):

@everywhere function random\_walk!(x::DArray, T)  
 if myid() == 1 # Main process (not a worker)  
 SPMD.spmd(random\_walk!, x, T; pids=workers())  
 else  
 random\_walk!(localpart(x), T)  
 end  
end

We have to use the @everywhere macro to load the function on each of the workers. Usually, you can put all the code that each worker needs in a separate file and just call @everywhere once.

This code just tells Julia how to call our original random\_walk! function on each block of the array, reusing our code from before. The performance (for 8 workers) is added to our graph:



Multiprocessing always has a much higher overhead than multithreading, but we can scale to potentially millions of processes, and use memory and resources from hundreds of different computers.

## What’s next?

Hopefully, in this blog I have been able to show you some of the potential of using Julia to enable a huge range of parallelism, and convince you that there is a low barrier to entry to get started! All of the code in this blog post can be found in the blog folder in [this GitHub repository](https://github.com/JamieMair/julia-for-research-with-hpc).

If you want to learn more about Julia, you can visit the [website](https://julialang.org/) and if you want to start learning the language, start with the [Julia manual](https://docs.julialang.org/en/v1/manual/getting-started/). In this blog, we haven’t even gone through many of the other reasons that people use Julia, such as [multiple dispatch](https://www.youtube.com/watch?v=kc9HwsxE1OY) or state-of-the-art support for [differential equations](https://docs.juliahub.com/DifferentialEquations/UQdwS/6.15.0/) and [scientific machine learning](https://sciml.ai/) as some examples. If there are packages that you need and are only available in Python, there is a package called [PyCall.jl](https://www.juliapackages.com/p/pycall) which will let you let you easily run and write Python code directly from Julia, giving you access to the entire Python ecosystem.

If you want to ask any more questions, there is a very active community on the [Julia discourse](https://discourse.julialang.org/), which is full of people happy to help you with any Julia issues or questions! Happy programming!