High Performance Computing in Julia from the ground up.

Measuring Performance & Optimisation

Aims

- To be able to profile and benchmark Julia code
- To understand the basics of **computational complexity**
- To begin learning some **optimisation** techniques (in Julia)

Measuring Performance



Cache

Cores Clock Cycles

Hemory Bandwidth

Storage

Disk I/O



Wall Time

Which resources can a program consume?



Network I/O

Measuring Wall Time

- The CPU has an internal clock used to synchronise action across the CPU
- Can be used to measure how long a piece of code takes by comparing the clock before and after execution
- Can use the <a>@time macro for simplicity, but this is not very accurate

Example: Sum of cubes

```
julia> f(arr) = sum(x->x^3, arr);
julia> arr = rand(1024);
julia> @time f(arr)
    0.042326 seconds (56.16 k allocations: 3.067 MiB, 99.96% compilation time)
251.11661067703318
julia> @time f(arr)
    0.000005 seconds (1 allocation: 16 bytes)
251.11661067703318
```

• First usage includes compile time - always measure twice

BenchmarkTools.jl



More accurate results

Why Benchmark?

- Time taken to execute code can be **highly** variable
- Some variability can be due to scheduling on the CPU, tasks may be interrupted while processing
- CPU thermals may cause it to lower the clock speed to avoid damage
- **Boost clocks** are common on modern CPUs, which is turned off when multiple cores are used

Profiling

- Statistical profilers sample your program during execution
- Interrupts the program and takes note of where in a stack frame (function) the program is
- Can infer which pieces of code take the longest, as they have the most samples
- Can show results as a flame graph

Profiling: Flame Graph

Ji_apply								
ji_appiy								
JI_appiy_	generic							
Revise.#	80 - kaalu							
Revise.ru	п_раске	ena						
REPL.eva	I_user_I	nput						
JI_appiy_	generic							
Core.eva	l I sus li	_						
ji_topiev	er_evar_r	n Iov						
ji_toplev	eval_i	Rex						
ji_topiev	evar_i	nex	unk					
Junknow		ver_mu	ЛК					
il intern	at tople	vel thu	ink callback					
ji_interpi	et_topie	ver_triu						
eval st	eval	body						
eval st.	eval	value						
eval v.	do ca	all						
do call	il app	olv aene	eric					
il appl	il	Main.#	#main					
il com		#mai	in#32					
il aen		#mai	in#32					
getAd		Main.	#solve all#31	colvot				
				Solve:				
jl_add		ji	il_gc_pool_alloc	solve!				
jl_add jl_add		ji ji	il_gc_pool_alloc il_gc_collect	solve: #prune!	#backtrack!			ConstraintSolve
jl_add jl_add addM		ji ji	il_gc_pool_alloc il_gc_collect jl_gc_collect	solve! solve! #prune! #prune!	#backtrack! #backtrack!			ConstraintSolve jl_apply_gen
jl_add jl_add addM addM		ji ji ji	il_gc_pool_alloc il_gc_collect jl_gc_collect sweep_page	solve: solve! #prune! #prune! ConstraintSolver.#prune	#backtrack! #backtrack! ConstraintSol	. prune!		ConstraintSolve jl_apply_gen ConstraintSo
jl_add jl_add addM addM			I gc_pool_alloc I gc_collect jl_gc_collect sweep_page sweep_page	solve: solve: #prune! #prune! ConstraintSolver.#prune jl_apply_generic	#backtrack! #backtrack! ConstraintSol jl_apply_g	prune!		ConstraintSolve jl_apply_gen ConstraintSo ConstraintSo
jl_add jl_add addM addM opera PassM			I_gc_pool_alloc I_gc_collect I_gc_collect Sweep_page Sweep_page sweep_page	solvei #prune! #prune! ConstraintSolver.#prune jL apply_generic ConstraintSolver.#all	#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain	prune! prune! getindex	ConstraintSolver.#prune!#20	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo
jl_add jl_add addM addM opera PassM FPP			I_gc_pool_alloc I_gc_collect J_gc_collect sweep_page Sweep_page sweep_page sweep_page	solvei #prune! #prune! [] apply_generic ConstraintSolver.#all [] ConstraintSolver.#all	#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 jl_apply_generic	ConstraintSolve jl_apply_gen ConstraintSo ConstraintSo
jl_add jl_add addM addM opera PassM FPP FPP			I_gc_pool_alloc I_gc_collect J_gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune il_apply_generic ConstraintSolver.#all ConstraintSolver.#all	#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 I_apply_generic ConstraintSolver.#all_different	ConstraintSolve jl_apply_gen ConstraintSo ConstraintSo Con
jl_add jl_add addM addM opera PassM FPP FPP			I_gc_pool_alloc I_gc_collect J_gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune jl_aply_generic ConstraintSolver.#all ConstraintSolver.#all	#backtrack! #backtrack! ConstraintSol J. apply_g Constrain Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo
jl_add jl_add addM addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect sweep_page sweep_page sweep_page sweep_page sweep_page sweep_page sweep_page	solve: #prune! #prune! ConstraintSolver.#prune ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all	#backtrack! #backtrack! ConstraintSol JI apply g Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 jl_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different strong_components ConstraintSolver.#all_different	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo
ji_add ji_add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune JI.apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all Stro	#backtrack! #backtrack! ConstraintSol Constrain Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 Image: ConstraintSolver.#all_different Image: ConstraintSolver.#all_different Image: ConstraintSolver.#all_different#11 ConstraintSolver.#all_different Image: ConstraintSolver.#all_different#11 Strong_components ConstraintSolver.#all_different#11 Strong_components MatrixN bipartite Type Type geti	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo ConstraintSo
ji_add ji_add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect I gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune jl.apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all Stro	#backtrack! #backtrack! [ConstraintSol] Lapply.g Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 jl_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different strong_components Sparse bipartite Type Type Type	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo
ji_add ji_add addM opera PassM FPP FPP			I_gc_pool_alloc I_gc_collect I_gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune jl_apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all Stro	<pre>#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain Constrain </pre>	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 J_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different Strong_components MatrixN bipartite Type sparse	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo
ji_add ji_add addM opera PassM FPP FPP			I_gc_pool_alloc I_gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune ji_apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all	<pre>#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain Constrain Constrain Constrain</pre>	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 I apply generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different strong_components MatrixN bipartite Type geti sparse bipartite sparse sparse	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo Con
ji_add ji_add addM addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune J. apply.generic ConstraintSolver.#all ConstraintSolver.#all Stro	 #backtrack! #backtrack! ConstraintSol Constrain Constrain	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 ILapply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different#11 strong_components Sparse sparse </td <td>ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo Con</td>	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo Con
ji_add ji_add addM addM opera PassM FPP FPP			I gc_pool_alloc J gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune jl_apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all I I I I I I I I I I I I I I I I I I I	<pre>#backtrack! #backtrack! ConstraintSol JLapply_g Constrain Constrain I I I I I I I I I I I I I I I I I I</pre>	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 Image: Sparse Spars	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo ConstraintSo ConstraintSo
jl_add jl_add addM addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune jl_apply_generic ConstraintSolver.#all Con	#backtrack! #backtrack! ConstraintSol jl_apply_g Constrain Constrain Output Image: Constrain Image: Constra	prune! grune! getindex getindex	ConstraintSolver.#prune!#20 j.apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components Sparse	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo Con
jI_add jI_add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect I gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune	#backtrack! #backtrack! IConstraintSol JI apply.g Constrain Constrain II apply.g Constrain II apply.g I	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 J_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different Strong_components Sparse	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo ConstraintSo
ji add ji add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect sweep_page sweep_page sweep_page sweep_page sweep_page sweep_page	solve: #prune! #prune! ConstraintSolver.#prune	#backtrack! #backtrack! OnstraintSol I apply_g Constrain Constrain Constrain I apply_g Constrain I apply_g	prune! getindex getindex	ConstraintSolver.#prune!#20 I apply generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different strong_components MatrixN bipartite Type sparse Spar	ConstraintSolve ji_apply_gen ConstraintSo ConstraintSo ConstraintSo
ji_add ji_add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune: ConstraintSolver.#prune ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all I apply generic ConstraintSolver.#all I apply generic ConstraintSolver.#all I apply generic ConstraintSolver.#all I apply generic ConstraintSolver.#all I apply generic I apply generic ConstraintSolver.#all I apply generic I apply	#backtrack! #backtrack! ConstraintSol Ji apply g Constrain Constrain Constrain Backstrack! Backstrak! Backstrak!	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components Sparse apply_generic Strong_components Sparse apply_generic Strong_components Sparse apply_generic Sparse apply_generic Sparse apply_generic Sparse apply_generic apply_generic Sparse apply_generic	ConstraintSolve iJ_apply_gen ConstraintSo Con
jI_add jJ_add addM addM opera PassM FPP FPP IIIIIIIIIIIIIIIIIIIIII			I gc_pool_alloc J gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune: #prune: ConstraintSolver.#prune	#backtrack! #backtrack! ConstraintSol JL apply_g Constrain Constrain Constrain II II II II II II II II III III III III IIII IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII	prune! prune! getindex getindex	ConstraintSolver.#prune!#20 j_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different#11 strong_components MatrixN bipartite Type sparse bipartite Sparse bipa	ConstraintSolve ji_apply_gen ConstraintSo ConstraintS
jl_add jl_add addM addM opera PassM FPP FPP III			I gc_pool_alloc J gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune: #prune: ConstraintSolver.#prune J. apply_generic ConstraintSolver.#all Con	#backtrack! #backtrack! ConstraintSol I Constrain Constrain I Constrain I Constrain I Constrain I I <td>prune! getindex getindex</td> <td>ConstraintSolver.#prune!#20 j.apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components Sparse Sparse</td> <td>ConstraintSolve ji_apply_gen ConstraintSo Con</td>	prune! getindex getindex	ConstraintSolver.#prune!#20 j.apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components Sparse	ConstraintSolve ji_apply_gen ConstraintSo Con
jI_add jJ_add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect I gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver. #prune J] apply_generic ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all ConstraintSolver.#all Stro Stro <td< td=""><td>#backtrack! #backtrack! I ConstraintSol I apply g Constrain Constrain I apply g Constrain I apply g I apply g Constrain I apply g I apply g I</td><td>prune! getindex getindex</td><td>ConstraintSolver.#prune!#20 I apply generic ConstraintSolver.#all different ConstraintSolver.#all different Strong_components ConstraintSolver.#all different strong_components Sparse Sparse</td><td>ConstraintSolve ji_apply_gen ConstraintSo Con</td></td<>	#backtrack! #backtrack! I ConstraintSol I apply g Constrain Constrain I apply g Constrain I apply g I apply g Constrain I apply g I apply g I	prune! getindex getindex	ConstraintSolver.#prune!#20 I apply generic ConstraintSolver.#all different ConstraintSolver.#all different Strong_components ConstraintSolver.#all different strong_components Sparse	ConstraintSolve ji_apply_gen ConstraintSo Con
ji add ji add addM opera PassM FPP FPP			I gc_pool_alloc I gc_collect J gc_collect Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page Sweep_page	solve: #prune! #prune! ConstraintSolver.#prune	#backtrack! #backtrack! ConstraintSol ji apply g Constrain Constrain Constrain Backtrack! Backtrak! Backtrack!	prune! getindex getindex	ConstraintSolver.#prune!#20 J_apply_generic ConstraintSolver.#all_different ConstraintSolver.#all_different Strong_components ConstraintSolver.#all_different Strong_components MatrixN bipartite Sparse	ConstraintSolve ji_apply_gen ConstraintSo Con

Profiling + Optimising Technique

- Use profiling to **identify** the slow functions in your code
- Use macros from *BenchmarkTools.jl* to measure the performance of the slow functions
- Only optimise the part of your code that is **slow**!
- Keep the old code for reference to compare benchmarks
- Keep hardware the **same**!
- Minimise number of **concurrent tasks**

Computational Complexity – Big \mathcal{O} notation







- All computers have different hardware, with varying speed
- How do you compare algorithms on different hardware?
- We compare the **computational complexity** of the algorithms, which measures how execution time will **scale** with varying input sizes
- The **complexity** classifies the number of resources required to run it, mostly focusing on computation time (or memory storage).

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```



Figure 5.4. Time taken when using elementmul for several values of n. The time taken is measured several times for each value of *n* using the **@belapsed** macro from *Benchmark-Tools.jl*.





function elementmul(a, b)
 # Make sure the inputs are the same size
 @assert all(size(a).==size(b))
 # Allocate a new array to store the result
 c = similar(a)
 for i in eachindex(a)
 c[i] = a[i] * b[i]
 end
 return c
end



Figure 5.4. Time taken when using elementmul for several values of n. The time taken is measured several times for each value of n using the @belapsed macro from *Benchmark-Tools.jl*.





function elementmul(a, b)
 # Make sure the inputs are the same size
 @assert all(size(a).==size(b))
 # Allocate a new array to store the result
 c = similar(a)
 for i in eachindex(a)
 c[i] = a[i] * b[i]
 end
 return c
end



Figure 5.4. Time taken when using elementmul for several values of n. The time taken is measured several times for each value of n using the @belapsed macro from *Benchmark-Tools.jl*.



function elementmul(a, b)
 # Make sure the inputs are the same size
 @assert all(size(a).==size(b))
 # Allocate a new array to store the result
 c = similar(a)
 for i in eachindex(a)
 c[i] = a[i] * b[i]
 end
 return c
end



Figure 5.4. Time taken when using elementmul for several values of n. The time taken is measured several times for each value of n using the @belapsed macro from *Benchmark-Tools.jl*.

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```





$$t(n) = e^c n^m + t_0$$

```
function elementmul(a, b)
    # Make sure the inputs are the same size
    @assert all(size(a).==size(b))
    # Allocate a new array to store the result
    c = similar(a)
    for i in eachindex(a)
        c[i] = a[i] * b[i]
    end
    return c
end
```



Figure 5.4. Time taken when using elementmul for several values of n. The time taken is measured several times for each value of *n* using the **@belapsed** macro from *Benchmark-Tools.jl*.

```
t(n) = an + b
```



Tools.jl.

Big \mathcal{O} notation

- Asymptotic Computational Complexity only keep largest growing factors, and throw away constants
- Usually deals with the **worst-case** complexity, but sometimes **average-case** is also interesting
- Is interesting for problems which **scale**, if problem size is small, benchmarking is often preferred
- Helps to **choose** the right algorithm for a given problem size



```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
   for i in 1:N
        point i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min distance squared = typemax(eltype(pointcloud))
        for j in 1:N
            point_j = pointcloud[:, j]
            distance squared = sum((point i .- point j).^2)
            if min_distance_squared < distance_squared</pre>
                min_distance_squared = distance_squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

```
function nearestneighbour(pointcloud)
             # Get the dimension and size of the point cloud
\mathcal{O}(1)
         \checkmark N, D = size(pointcloud)
             neighbours = zeros(Int, N)
             # Allocate a new array to store the result
             for i in 1:N
                 point i = pointcloud[:, i]
                 # Set the current minimum distance to the
                 # largest possible value, given the type.
                 min_distance_squared = typemax(eltype(pointcloud))
                 for j in 1:N
                     point_j = pointcloud[:, j]
                     distance squared = sum((point i .- point j).^2)
                     if min_distance_squared < distance_squared</pre>
                          min_distance_squared = distance_squared
                          neighbours[i] = j
                      end
                 end
             end
             return neighbours
         end
```

```
function nearestneighbour(pointcloud)
                    # Get the dimension and size of the point cloud
   \mathcal{O}(n)
                 \int N, D = size(pointcloud)
                   neighbours = zeros(Int, N)
(assumption)
                     # Allocate a new array to store the result
                    for i in 1:N
                         point i = pointcloud[:, i]
                         # Set the current minimum distance to the
                         # largest possible value, given the type.
                         min_distance_squared = typemax(eltype(pointcloud))
                         for j in 1:N
                             point_j = pointcloud[:, j]
                             distance squared = sum((point i .- point j).^2)
                             if min_distance_squared < distance_squared</pre>
                                 min_distance_squared = distance_squared
                                 neighbours[i] = j
                             end
                         end
                     end
                     return neighbours
                end
```

```
function nearestneighbour(pointcloud)
                 # Get the dimension and size of the point cloud
             N, D = size(pointcloud)
neighbours = zeros(Int, N)
\mathcal{O}(n)
                  # Allocate a new array to store the result
                 for i in 1:N
         O(1) { point_i = pointcloud[:, i]
# Set the current minimum distance to the
# largest possible value, given the type.
min_distance_squared = typemax(eltype(pointcloud))
                       for j in 1:N
                            point j = pointcloud[:, j]
                            distance squared = sum((point i .- point j).^2)
                            if min distance squared < distance squared
                                 min_distance_squared = distance_squared
                                 neighbours[i] = j
                            end
                       end
                  end
                  return neighbours
            end
```













Example – Recursive calls

```
function recursivecalc(n, a)
    if n==1
        return a*a
    end
    s = 0.0
    for i in 1:n
        s += recursivecalc(n-1, i)
    end
    return s
end
```

 $\mathcal{O}(n!)$

Example – Sorted Insert

```
function insertintosorted!(numbers, num)
    n = length(numbers)
    startpoint = 1
    endpoint = n
    while startpoint-endpoint > 1
        midpoint = (startpoint+endpoint)+2
        if num < numbers[midpoint]</pre>
            endpoint = midpoint
        elseif num > numbers[midpoint]
            startpoint = midpoint
        else
             insert!(numbers, midpoint, num)
             return
        end
    end
    insert!(numbers, startpoint, num)
    nothing
end
```

Algorithm 5.6. A simple algorithm which inserts a number into an already sorted list, making sure the list is still sorted after insertion. This algorithm assumes the list is sorted in ascending order.

 $\mathcal{O}(\log n)$

Summary

- Choose the **best algorithm** for your use case
- Constants matter when the problem size is smaller
- Should always benchmark the code to get an idea of the constants
- Can use a combination of complexity and benchmarking to predict how long execution will take for a larger input / work out the maximum input size

election at the end -ad _ob.select= 1 er_ob.select=1 ntext.scene.objects.act Selected" + str(modifie irror_ob.select = 0 bpy.context.selected_ob ta.objects[one.name].selected_ob

mint("please select exaction

OPERATOR CLASSES -----

ypes.Operator):
X mirror to the select
ject.mirror_mirror_x"
ror X"

context): cxt.active_object is not

Optimisation

Why optimise our code?









Get results **faster** – quickly iterate on our experiments Increase of your own productivity

Scale up the experiments to **larger** sizes and **longer** times Use fewer compute resources on the HPC

Why is code **slow** in the first place?

- **Incorrect** algorithms are chosen
- Contains **unnecessary** operations
- Program has to spend time working out what to do, instead of having the instructions ready (not compiled)
- Choices are made to cooperate with language design, but cause poor performance (i.e. must vectorise code with numpy)

Speed of operations

Not all CPU operations are created equal

ithare.com	Operation Cost in CPU Cycles	10º	10 ¹	10 ²		10 ³	IO ^₄ 1	0 ⁵	106
"Simple"	register-register op (ADD,OR,etc.)	<1							+
· · · · ·	Memory write	~1							
	Bypass delay: switch between								
	integer and floating-point units	0-3							
	"Right" branch of "if"	1-2							
	Floating-point/vector addition	1-3							
	Multiplication (integer/float/vector)	1-7							
	Return error and check	1-7							
	L1 read		3-4						
	TLB miss		7-21						
	L2 read		10-12						
"Wrong" b	ranch of "if" (branch misprediction)		10-20						
	Floating-point division		10-40						
	128-bit vector division		10-70						
	Atomics/CAS		15-3	0					
	C function direct call		15-3	0					
	Integer division		15-4	0					
	C function indirect call		20	-50					
	C++ virtual function call			30-60					
	L3 read			30-70					
	Main RAM read			100-1	150				
NU	JMA: different-socket atomics/CAS			100.2	200				
	(guesstimate)			100-3	500				
	NUMA: different-socket L3 read			100-3	300				
Allocatio	n+deallocation pair (small objects)			20	00-500				
NUM	A: different-socket main RAM read				300-500				
	Kernel call					1000-1500			
Т	hread context switch (direct costs)					2000			
	C++ Exception thrown+caught					5000-10000	2		
	Thread context switch (total costs,						10000 - 1 million		
	including cache invalidation)						10000 - Finanton		



Distance which light travels while the operation is performed

Example: Vectorisation

Native Python

import math

def f_native(x_values):

y_values = []

for x in x_values:

y = math.exp(x)*math.sin(x)*x + x**4 + 5 * math.sqrt(3*x)

y_values.append(y)

```
return y_values
```

```
52.5 ms \pm 446 \mus per loop
(mean \pm std. dev. of 7
runs, 10 loops each)
```

Numpy

 $\sim 16x$ speedup

import numpy as np
def f_np(x):
 return np.exp(x)*np.sin(x)*x + x**4 + 5 * np.sqrt(3*x)

```
3.16 ms ± 29.2 μs per loop
(mean ± std. dev. of 7
runs, 10 loops each)
```

Example: Vectorisation

Native Julia

function f_vectorised(x)

y = similar(x)

(a). y = exp(x)*sin(x)*x + x^4 + 5 * sqrt(3*x)

return y

end

function f_native(x_array)

y = similar(x_array)

@inbounds for i in eachindex(x_array)

```
x = x_array[i]
```

```
y[i] = exp(x)*sin(x)*x + x^4 + 5 * sqrt(3*x)
```

```
end
```

```
return y
```

end

f_vectorised -> 942.500 μs (2 allocations: 512.11 KiB)

f_native -> 900.000 μs (2 allocations: 512.11 KiB)

Numpy

 $\sim 3.5x$ speedup

import numpy as np
def f_np(x):
 return np.exp(x)*np.sin(x)*x + x**4 + 5 * np.sqrt(3*x)

3.16 ms ± 29.2 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)

Example: Vectorisation

Julia

- Julia will fuse broadcast operations together
- Broadcasting does not allocate intermediate results
- Vectorised vs for loop is aesthetic
- Compiler can **automatically** simd the code
- Does not need to pass the array through the language barrier – all native Julia

Numpy

- Requires rewriting the code into a vectorised loop
- Moves the for loop into a compiled function (in C)

Hardware SIMD vs Vectorisation

- @simd just gives the compiler more leeway to use hardware level vector instructions (SSE and AVX) on operations that may change results
- Compiler will automatically hardware vectorise code in specific cases
- Read the help for more info: help?> @simd

Cache and Memory Locality

- All arrays are stored in a **contiguous** block of memory
- Operations on an array done in order are much faster due to cache
- Cache lines store an entire line of memory (~128 bits)



Address (Hex)	Data
5F19	01001001
5F1A	00001000
5F1B	1000001
5F1C	00011110

Multidimensional Arrays

- Even multidimensional arrays are stored linearly in memory
- Indexing scheme must be used to calculate linear index from cartesian index
- Row-major:

$$k = jN_y + i$$

• Column-major:

 $k = iN_x + j$

• Julia uses column-major

Row-major order



Column-major order



Multidimensional Arrays

• Order of iteration makes a **big** performance difference

julia> @benchmark row_major_matrix_add!(\$C, \$A, \$B)
BenchmarkTools.Trial: 184 samples with 1 evaluation.
Range (min ... max): 26.970 ms ... 27.820 ms GC (min ... max): 0.00% ... 0.00%
Time (median): 27.200 ms GC (median): 0.00%
Time (mean ± σ): 27.218 ms ± 133.593 μs GC (mean ± σ): 0.00% ± 0.00%
27 ms Histogram: frequency by time 27.8 ms <
Memory estimate: 0 bytes, allocs estimate: 0.
julia> @benchmark column major matrix add!(\$C, \$A, \$B)

 BenchmarkTools.Trial: 4000 samples with 1 evaluation.

 Range (min ... max): 1.187 ms ... 1.447 ms

 GC (min ... max): 0.00% ... 0.00%

 Time (median): 1.215 ms

 Time (mean $\pm \sigma$): 1.230 ms $\pm 36.353 \ \mu s$

 GC (mean $\pm \sigma$): 0.00% $\pm 0.00\%$



Memory estimate: 0 bytes, allocs estimate: 0.

function row_major_matrix_add!(C, A, B)
 @assert size(C)==size(A)==size(B)
 @inbounds for i in axes(A, 2)
 for j in axes(A, 1)
 C[i, j] = A[i, j] + B[i, j]
 end
 end
 nothing
end

function column_major_matrix_add!(C, A, B)
 @assert size(C)==size(A)==size(B)
 @inbounds for j = axes(A, 2)
 for i in axes(A, 1)
 C[i, j] = A[i, j] + B[i, j]
 end
 end
 nothing
end

Multidimensional Arrays

• Julia arrays can be **linearly** indexed – same performance



Memory estimate: 0 bytes, allocs estimate: 0.

function vector_add!(C, A, B)
 @inbounds for i in eachindex(C, A, B)
 C[i] = A[i] + B[i]
 end
 nothing
end

Heap Allocations

- When we say "allocations", we refer to heap allocations
- Allocating memory on the heap is costly
- Memory needs to be cleaned up by the Garbage Collector (GC)
- The GC will interrupt processing to clean up memory performance hit!

Preallocating vs Appending

```
function cumulative_sum_preallocated(numbers)
    results = similar(numbers)
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        results[i] = total_sum
    end
    return results
end
```

```
function cumulative_sum_appending(numbers)
    results = (eltype(numbers))[]
    total_sum = zero(eltype(numbers))
    for i in eachindex(numbers)
        total_sum += numbers[i]
        push!(results, total_sum)
    end
    return results
end
```



Reusing Memory: Caching

- Can pre-allocate storage for the results
- Still need to allocate the memory, but only needs to be done once
- Can make a difference on speed, but also memory

```
function example_equation_no_cache(x)
    numerator = 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    denominator = exp.(-4 .* x) .- x .^ 2
    y = numerator ./ denominator
    return y
end
```

```
function example_equation_cache!(y, x)
    # Set y to the value of the numerator
    y .= 5 .* x .^ 5 .* sin.(x.^2) .+ 20
    # Divide out the denominator
    y ./= exp.(-4 .* x) .- x .^ 2
    return y
end
```

Reusing Memory: Avoid Copying

- Slicing an array allocates new memory for that array
- We can instead create a view into that data to point at the same memory
- Changing values in a view will propagate back to the original array
- *Sometimes* copying may be **faster** due to memory locality

```
function nearestneighbour(pointcloud)
    # Get the dimension and size of the point cloud
    N, D = size(pointcloud)
    neighbours = zeros(Int, N)
    # Allocate a new array to store the result
    for i in 1:N
        point i = pointcloud[:, i]
        # Set the current minimum distance to the
        # largest possible value, given the type.
        min distance squared = typemax(eltype(pointcloud))
        for j in 1:N
            point j = pointcloud[:, j]
            distance squared = sum((point i .- point j).^2)
            if min_distance_squared < distance_squared</pre>
                min distance squared = distance squared
                neighbours[i] = j
            end
        end
    end
    return neighbours
end
```

Avoiding the Heap – Stack Allocating

- Can use packages like StaticArrays.jl to create small arrays on the stack
- Can be used to make the code more readable, and avoiding allocations
- These are **immutable** by default
- Useful for small vectors representing positions/velocities etc

Profiling & Reducing Allocations

Live Demonstration

Workshop – Thursday 26/01/2023

Assignment

Link will be on the website

Tasks:

- Optimise the nearest neighbour algorithm
- Read through the README for assignment details