



ParallelFlow Guide

High-Performance Blueprint Nodes for Unreal Engine

Version 2.1.1 | Unreal Engine 5.2 - 5.8

Delete one slow node. Drop in one fast one. The stutter goes away.

This is the only guide you need. It starts from zero, walks through a working example, explains how every node behaves, gives you ready-made recipes for the things games do every day, and ends with a complete reference of all 138 nodes. Plain language throughout. No C++ and no threading knowledge required.

Chapters

- 1. What ParallelFlow is, in plain words
- 2. Install it
- 3. Your first node, step by step
- 4. How every node works
- 5. When to use it, and when not to
- 6. Everyday recipes
- 7. The Performance Advisor
- 8. Measure it: benchmark, demo and debugger
- 9. Security: what the file nodes can and cannot touch
- 10. Troubleshooting and questions
- 11. Support and version notes
- 12. Node reference (every node)

(c) Black Scar Studio 2026. All Rights Reserved. Support: <https://discord.gg/a3QdMX9JZj>

Version 2.1.1. Unreal Engine 5.2 to 5.8. Black Scar Studio.

This is the only guide for ParallelFlow. It starts from zero and goes all the way to a full reference of every node. You do not need to know C++. You do not need to know anything about threads. If you can wire a Print String to BeginPlay, you can use everything in here.

How to read it: chapters 1 to 5 take about fifteen minutes and cover everything you need to use the plugin well. Chapter 6 is the recipe book: come back to it when you have a specific job to do. Chapters 7 to 10 cover the Advisor, measuring, security and troubleshooting. The last chapter lists every node with all of its pins.

1. What ParallelFlow is, in plain words

The problem it solves

Your game runs on one main thread, called the Game Thread. Every frame, that thread has to run your Blueprints, move your actors, and then hand the frame to the renderer. If a Blueprint takes 30 milliseconds to do something, the whole frame takes 30 milliseconds longer. That is the stutter you feel.

Most of the time Blueprint is fast enough. But a few jobs are slow, and they are the same few jobs in every game:

- Sorting or searching a big list.
- Measuring the distance to a lot of actors ("which enemy is closest?").
- Checking which of many actors the player can actually see.
- Placing hundreds of trees, rocks or spawn points without overlaps.
- Saving or loading a big file.
- Any loop that runs over thousands of things.

When Blueprint does one of these, the game **stops and waits**.

What the plugin does

ParallelFlow gives you replacement nodes for those jobs. Each node does the same work in fast native code, spread across the spare cores of the CPU, **without stopping the game**. The result comes back to your Blueprint a moment later, on the Game Thread, ready to use.

You delete one slow node. You drop in one ParallelFlow node. The stutter goes away. That is the whole idea.

What it does not do

Being clear about this saves you time:

- It does **not** scan or rewrite your Blueprints. Nothing happens automatically. You choose the slow spots and swap in a node.

- It does **not** run your own Blueprint code on other threads. No plugin can do that; the engine would crash. ParallelFlow speeds up the built-in operations it ships with, such as sorting, searching, distance checks, traces and file work. For your own per-item logic there is a different tool, the frame-spread loops, covered in chapter 6.
- It does **not** make small things faster. Sorting 50 items is already instant in Blueprint. The plugin only helps when the data is big. It even tells you so, honestly, through the Advisor node.

"So it is just C++ nodes?"

Yes. That is the point. It is production-quality, multithreaded C++ for the operations Blueprint is slow at, without you writing, compiling or maintaining any C++. Plus tools that tell you where using it is actually worth it.

2. Install it

From Fab (the normal way)

1. Open the **Epic Games Launcher** and go to **Library**.
2. Scroll to **Fab Library**, find **ParallelFlow** and click **Install to Engine**. Pick your engine version (5.2 to 5.8 are supported).
3. Open your project. Go to **Edit > Plugins**, search for **ParallelFlow**, tick **Enabled**, and restart the editor when it asks.

Plugins installed this way come with prebuilt binaries for Windows, so a Blueprint-only project can use them with no compiler installed.

From a zip (direct download or testing)

1. Unzip. You get a folder called `ParallelFlow` with `ParallelFlow.uplugin` inside it.
2. Copy that folder into your project's `Plugins` folder, so the descriptor ends up at `YourProject/Plugins/ParallelFlow/ParallelFlow.uplugin`. Create the `Plugins` folder if it does not exist.
3. Open the project. The editor will say the `ParallelFlow` modules are missing and ask to rebuild them. Click **Yes**. It takes a minute or two.

The zip contains source code, so this route needs a compiler: on Windows, **Visual Studio 2022** with the **Game development with C++** workload; on Mac, **Xcode**. Use the zip that matches your engine version. Each zip is the same plugin, only the declared engine version differs.

Check that it worked

Open any Blueprint, right-click in the graph and type `ParallelFlow`. You will see the categories:

Category	What is in it
ParallelFlow > Gameplay	Actor queries, sorting your own structs, procedural placement
ParallelFlow > Parallel Arrays	Sort, filter, transform, combine, find, count over big arrays
ParallelFlow > Math Processing	Distances, nearest point, grids, heatmaps, noise, voronoi, bounds
ParallelFlow > File Processing	Save, load, JSON, CSV, copy/move/delete, compression, hashing
ParallelFlow > Async Flow	Frame-spread loops, delays, splits, joins, queues, cancelling
ParallelFlow > Utility	Random data, images, screenshots, text tools, encryption, encoding
ParallelFlow > Debug	Live scheduler statistics and task lists
ParallelFlow > Advisor	The Performance Advisor

Category	What is in it
ParallelFlow > Benchmark	Measure old way versus new way on your machine

Two other places to know about:

- **Tools > Miscellaneous > ParallelFlow Debugger** opens the live task window.
- **Edit > Project Settings > Plugins > ParallelFlow** holds the file security settings (chapter 9).

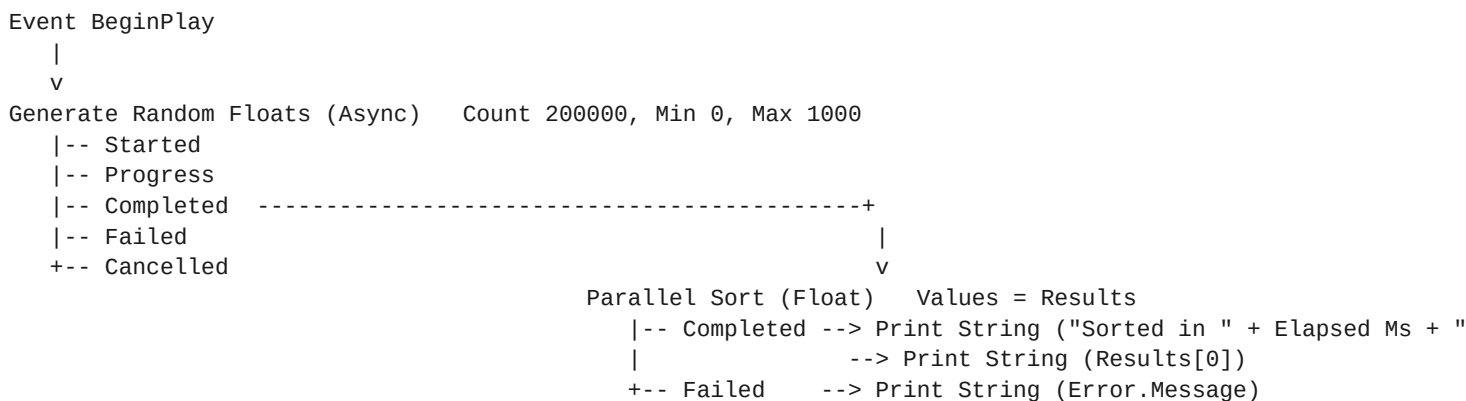
3. Your first node, step by step

Let us sort 200,000 numbers without freezing the game, and print how long it took. Five minutes, start to finish.

Build it

1. Open the **Level Blueprint** (Blueprints button on the toolbar > Open Level Blueprint), or any actor Blueprint you like.
2. Find the **Event BeginPlay** node.
3. Drag off its exec pin, type `Generate Random Floats` and pick **Generate Random Floats (Async)**. Set **Count** to 200000, **Min** to 0 and **Max** to 1000.
4. Drag off the **Completed** pin of that node, type `Parallel Sort` and pick **Parallel Sort (Float)**. Connect the **Results** output of the generator to the **Values** input of the sort.
5. Drag off the **Completed** pin of the sort node and add a **Print String**. For the text, drag from **Elapsed Ms**, add an **Append** node, and make it say something like `Sorted in 12.4 ms`.
6. Add one more **Print String** on the same **Completed** pin, and print element 0 of **Results** (use a **Get** node with index 0). It should be the smallest number, close to 0.
7. Press **Play**.

Here is the shape of the graph:



What you will see

Two lines on screen. The time will be a few milliseconds. The frame rate did not dip while it ran, because the sorting happened on worker threads and your Blueprint only got involved when the answer was ready.

Two things worth noticing

The result is on the Completed pin, not on the node's normal output. The node starts the work and returns immediately. Your "what happens next" logic must hang off **Completed**. If you put a **Print String** on the exec line straight after the node, it runs before the sort has finished and the array will be empty. This is the one habit to build. Everything else follows from it.

Failed is your friend. Wire the **Failed** pin to a Print String showing **Error > Message** on every node while you are building. ParallelFlow never fails silently: if a path is blocked, an array is empty, or a count is wrong, you get a sentence telling you exactly what happened.

Watch it happen

Open **Tools > Miscellaneous > ParallelFlow Debugger** before pressing Play. You will see the two tasks appear, run and finish, with their timings and the thread they ran on. Bump Count to 2,000,000 and watch the game keep rendering while the sort takes longer.

4. How every node works

All ParallelFlow async nodes look the same. Once you understand one, you understand all of them.

The pins

----> [Any ParallelFlow async node]	----> Started	(fires right away)
	----> Progress	(several times while it runs)
	----> Completed	(YOUR RESULT IS READY)
	----> Failed	(something went wrong)
	----> Cancelled	(you cancelled it)
	----> Task	(blue object pin; drag off it to call Ca
	----> Task Handle	(for Cancel Task and Wait For All Tasks)
	----> Progress Value	(0.0 to 1.0; filled by Progress)
	----> Results ...	(the result pins; filled by Completed)
	----> Elapsed Ms	(how long the work took; filled by Compl
	----> Error	(Message and Source; filled by Failed)

The exec pins are at the top of the node, the data pins under them. Each event fills its own data pins: Started fills Task Handle, Progress fills Progress Value, Completed fills the result pins and Elapsed Ms, Failed fills Error. Read a data pin from the event that fills it.

- **Started** fires immediately and hands you a **Task Handle**. Store it in a variable if you might want to cancel later or wait on several tasks.
- **Progress** fires on the Game Thread as the work advances and fills **Progress Value** (0 to 1). Wire Progress Value to a progress bar for long jobs. Ignore it for short ones.
- **Completed** fires once, with the result. This is where the rest of your logic goes.
- **Failed** fires instead of Completed when something is wrong. **Error > Message** says what, in plain words.
- **Cancelled** fires if the task was cancelled before it finished.
- **Task** is the task object itself. Drag off it and call **Cancel** to stop it. Same thing as **Cancel Task** with the handle.

You usually connect only **Completed**, and **Failed** while developing.

The one rule: the result arrives later

The node returns on the same frame, but the result does not. It can arrive a few milliseconds later, or a few frames later for big jobs. So:

- Put the logic that needs the result on the **Completed** pin.
- Do not read the result variable on the exec line after the node.
- If several things must happen after each other (generate, then sort, then filter), chain them: each node hangs off the previous node's Completed.

Where you can place these nodes

- **Event Graphs**: BeginPlay, Tick, input events, Custom Events, timers. This is where they belong.

- **Actors, Components, Widgets, Game Mode, Game Instance, Player Controller, Level Blueprint:** all fine.
- **Editor Utility Widgets and Editor Utility Blueprints:** fine. This is how you use the placement nodes at design time.
- **Not inside a Blueprint Function.** A function must finish in one go; these nodes wait for a result. If you want to reuse a chain, put it behind a **Custom Event** instead of a Function.
- **Not in a Construction Script**, for the same reason.
- **Not in an Animation Blueprint.** The AnimGraph only takes animation nodes, and the Anim Blueprint event graph does not allow waiting nodes. Do the heavy work in your Character or a manager Blueprint, save the result to a variable, and let the Anim Blueprint read that variable.

If the Blueprint that started the task goes away

If the actor that started a task is destroyed before the task finishes, the result is simply dropped. Nothing crashes. Actor queries also handle actors that are destroyed while the query runs: they are left out of the result.

Priority

Most nodes have a **Priority** pin hidden under the small arrow at the bottom of the node (Low, Normal, High, Critical). Leave it on Normal. Use High for something the player is waiting on, such as a save before a level change, and Low for background housekeeping. It changes the order in which queued tasks start; it does not make any single task faster.

Multiplayer

Results are local to the machine that ran the node. Nothing is replicated. Run gameplay decisions (who takes damage, who is targeted) on the server and send the outcome the normal way. Run cosmetic work (sorting a local leaderboard display, placing local foliage) on each client.

Where the work actually runs

Array, math and placement nodes run on the engine's worker threads and use as many cores as the machine has spare. Trace-based nodes (line of sight, snapping to ground) use the engine's own asynchronous physics traces, which the physics system resolves without blocking the frame. Frame-spread loops run your Blueprint code on the Game Thread, a little per frame. In every case the pieces that touch actors, worlds and Blueprints happen on the Game Thread, which is why it is safe.

5. When to use it, and when not to

The golden rule

Small list? Use normal Blueprint. Big list? Use ParallelFlow.

How many items	What to do
Under about 1,000	Normal Blueprint. It is already instant.
1,000 to 10,000	Worth trying ParallelFlow. Measure it.
10,000 and up	Use ParallelFlow.
Anything heavy on Tick	Use ParallelFlow, or do it less often (a timer every 0.2 seconds is usually plenty).

There are two kinds of work where ParallelFlow helps even at small counts, because each item is expensive on its own:

- **Line traces to many targets:** line of sight, snapping to ground. Twenty traces on the Game Thread is already a visible hitch when ten AIs do it.
- **Placing things without overlaps:** the check grows with the square of the point count, so it explodes after a few hundred points.

Ask the Advisor when unsure

You do not have to guess. Right-click, type `slow`, and pick **Advise**. Tell it what you are doing (**Operation**), how many items (**Element Count**) and whether it runs every frame. It answers in a sentence, names the node, and honestly says **Keep Using Blueprint** when your data is small. Chapter 7 has the details.

Things ParallelFlow is not for

- **Your own logic per item.** If every item needs your custom Blueprint code (apply a buff, check a quest flag, spawn a particle), the work has to run on the Game Thread. Use **Async Loop** or **Async For Each (Indices)** so it is spread over several frames and never hitches, but do not expect a speedup.
- **Making the whole game faster.** It speeds up the specific operations you replace. Your rendering, physics and animation are untouched.
- **Replacing PCG, Mass or Niagara.** Those are systems for huge counts with their own pipelines. ParallelFlow is for the Blueprint logic around your game.

6. Everyday recipes

Each recipe shows the slow way, the fast way, and what to watch for. Pin names are exactly as they appear on the nodes.

6.1 The closest enemy (the most common one)

The slow way: Get All Actors Of Class, then a ForEach measuring the distance to every actor and tracking the smallest. Every frame. For every AI.

The fast way:

```
Get All Actors Of Class (BP_Enemy) --> Get Nearest Actors
                                   Actors = Out Actors
                                   Origin = Get Actor Location (self)
                                   Count = 1          (or 5 for "the closest few")
                                   |
                                   +-- Completed --> Actors[0] is your target
                                                Distances[0] is how far away it is
```

Steps:

1. Add **Get All Actors Of Class** with your enemy class.
2. Drag off its exec pin, search nearest and pick **Get Nearest Actors**.
3. Connect **Out Actors** to **Actors**. Connect your actor's location to **Origin**. Set **Count** to how many you want back.
4. On **Completed**, use **Get (a copy)** with index 0 on **Actors** for the nearest one. **Distances** is in the same order, so index 0 is its distance.

Tips:

- You get the distances for free. Do not measure them again.
- Do not run this on Tick. Put it on a timer (Set Timer by Event, every 0.2 to 0.5 seconds) and store the result. Nobody can tell the difference, and the cost drops by 90 percent before you even optimize.
- If the enemy list is short (a handful of enemies), a normal ForEach is fine. The node pays off when there are hundreds.

6.2 Everything within range (AoE, pickups, interaction prompts)

```
Get All Actors Of Class --> Get Actors In Radius
                           Actors = Out Actors
                           Origin = explosion location
                           Radius = 1000
                           Sort By Distance = true
                           |
                           +-- Completed --> ForEach (Actors) --> Apply Damage
```

You get back only the actors inside the radius, nearest first. Your ForEach now runs over 12 actors instead of 3,000. This is a distance test from the actor's location, not a physics overlap, so it does not care about collision shapes. That is exactly what you want for damage falloff, pickup ranges and "press E to

interact" checks.

Damage falloff example: on Completed, ForEach over **Actors** with the matching **Distances**; damage = MaxDamage x (1 - Distance / Radius).

6.3 What is in front of me (field of view, aim assist, melee arcs)

```
Get All Actors Of Class (BP_Energy) --> Get Actors In Cone
Actors      = Out Actors
Origin      = Get Actor Location (self)
Direction   = Get Actor Forward Vector (self)
Cone Half Angle Degrees = 45      (a 90 degree cone in total)
Radius      = 2000
|
+-- Completed --> Actors in front of you, nearest first
```

Cone Half Angle Degrees is measured from the direction line to the edge of the cone. 45 gives a 90 degree field of view. 10 gives a narrow aim-assist cone. 90 gives a full half-space ("anything not behind me").

For aim assist: use the camera's location and forward vector instead of the character's, with a small angle (5 to 10) and the weapon's range as the radius. Actors[0] is the best candidate.

For an enemy's eyes: chain this with recipe 6.4. Cone first (cheap: no traces), then **Get Visible Actors** on what is left, so traces are spent only on the few actors that could possibly be seen.

6.4 Who can this AI actually see (line of sight to many actors)

The slow way: a ForEach over every candidate doing a Line Trace each, on the Game Thread.

The fast way:

```
Get All Actors Of Class (BP_Player) --> Get Visible Actors
Actors      = Out Actors
Eye Location = Get Actor Eyes View Point (self)
Max Distance = 3000      (0 = unlimited)
Trace Channel = Visibility
Ignore Actor = self      (under the small arrow)
|
+-- Completed --> Actors = everything with a clear line of si
                    nearest first, with Distances
```

Actors beyond **Max Distance** are dropped before any trace runs, so traces are spent only on candidates that matter. Every trace is asynchronous: the frame never waits. A trace that hits the target actor itself counts as visible; only something else in the way blocks it.

Two things people get wrong:

- **Set Ignore Actor to the viewer.** Otherwise the trace starts inside the viewer's own collision and is blocked immediately, and everything looks hidden. The pin is under the small arrow at the bottom of the node.
- **Use the Visibility channel** unless you have a reason not to. What matters is that walls and cover **block** the channel you pick. If they do not, the trace passes straight through them and an actor behind

a wall is counted as visible. Whether the target itself blocks the channel does not matter: reaching the target counts as visible either way.

Trace Complex (also under the arrow) traces against the exact mesh triangles instead of simple collision. Slower, more precise. Leave it off unless you need it.

6.5 Sort an inventory or leaderboard (sorting your own structs)

Blueprint cannot sort an array of your own structs quickly. Here is how you do it anyway, in three steps.

Say you have an array of **Item** structs with **Name**, **Value** and **Weight**, and you want them ordered by Value, highest first.

Step 1. Build a plain array of just the values.

```
ForEach (Items) --> Add (Item.Value) to a Float array called Keys
```

Keys now holds one number per item, in the same order as Items.

Step 2. Sort the keys and get the order back.

```
Sort By Key (Float)   Keys = Keys, Ascending = false   (highest first)
|
+-- Completed --> Sorted Order
```

Step 3. Read your original array in that order.

```
ForEach (Sorted Order) --> Get Items[Array Element] --> Add to Sorted Items
```

Done. **Sorted Order** is just a list of positions. For keys [5, 1, 9] sorted ascending, Sorted Order is [1, 0, 2]: the item at index 1 comes first, then index 0, then index 2. It works for any array at all: structs, actors, objects, texts. Use **Sort By Key (Integer)** when the value is an integer (score, tier, rarity).

Why it is fast: the sort itself runs on worker threads. Steps 1 and 3 are simple loops with no comparisons in them, and Blueprint handles those fine even for tens of thousands of items.

6.6 Order actors by distance

Sometimes you want all of them, in order: the closest first for spawning effects, the farthest first for unloading.

```
Get All Actors Of Class --> Sort Actors By Distance
                               Actors = Out Actors, Origin = camera location, Nearest First = true
                               |
                               +-- Completed --> Actors (ordered), Distances (same order)
```

Turn **Nearest First** off to get the farthest first.

6.7 Spawn 500 trees on a hill (procedural placement)

The slow way: pick a random point, compare it with every point placed so far, reject it if too close, repeat. Then a Line Trace per point to find the ground. That is a nested loop plus hundreds of traces on the Game Thread. The editor freezes for seconds.

The fast way, two nodes:

```
Scatter Points (Even Spacing)
  Center      = middle of the area
  Area Size   = (8000, 8000)
  Min Spacing = 300                      (nothing closer than this to anything else)
  Max Points  = 0                        (0 = as many as fit)
  |
  +-- Completed --> Results --> Snap Points To Ground
                                Points      = Results
                                Trace Channel = Visibility
                                Trace Up     = 1000
                                Trace Down   = 10000
                                Max Slope Degrees = 35      (skip cliffs)
                                |
                                +-- Completed --> ForEach (Snapped Points) --> Spawn Actor
```

What each part does:

- **Scatter Points** fills a flat rectangle with random points that are all at least **Min Spacing** apart (blue-noise sampling). The points come out at the height of **Center**. The same **Random Seed** always gives the same layout, so a level looks the same every time it is opened.
- **Snap Points To Ground** fires one asynchronous trace per point, starting **Trace Up** above the point and going **Trace Down** below it. Points that hit nothing, or that land on a slope steeper than **Max Slope Degrees**, are removed. What comes back is ready to spawn on: nothing floats, nothing sits on a cliff face.

Rotate to match the ground: **Surface Normals** is in the same order as **Snapped Points**. Feed the normal into **Make Rot From Z** (or **Make Rot From ZX** with the actor's forward) to tilt each spawned actor to the slope.

Regular layouts: if you want rows and columns instead of natural scatter, use **Grid Processing (Async)** to make the points (with a little **Jitter Amount** for a less mechanical look) and snap those instead.

Where to run it: at runtime on **BeginPlay**, or at design time from an **Editor Utility Widget** or **Editor Utility Blueprint**. Not from a Construction Script (see chapter 4).

6.8 Big arrays: sort, filter, find, count, combine

Every array node comes in four flavours, **Float**, **Integer**, **Vector** and **String**, with the same pins.

Sort:

```
Your float array --> Parallel Sort (Float)   Ascending = true --> Completed --> Results
```

Vectors can be sorted by **Length** or by a single component (**Sort Mode**). Strings are sorted alphabetically, case-insensitive unless you say otherwise.

Filter (keep a range):

```
Parallel Filter (Float)   Values, Min = 0, Max = 50 --> Results (only values inside the range, order kept)
```

Parallel Filter (Vector) keeps points whose distance to **Center** is between **Min Distance** and **Max Distance**, which is a radius query on raw points. **Parallel Filter (String)** keeps strings that **Contains**, **Not**

Contains, Starts With, Ends With a pattern, or whose length is in a range.

Find and count:

```
Parallel Find (Integer)  Values, Value = 42 --> Completed --> Result = index of the first match, or -1
Parallel Count (Float)   Values, Min, Max --> Completed --> Result = how many are in the range
Parallel Search (String Contains) Values, Substring --> Result = index of the first string containing it
```

Combine (sum, average, min, max, product):

```
Combine Array (Float)  Values, Operation = Average --> Completed --> Result
```

Example: the average damage dealt across 100,000 recorded hits, in one node. Floats accumulate in double precision so big sums stay accurate. For vectors, **Average** gives the centroid of a point cloud.

Transform every element: Transform Array applies one built-in operation to every element: for floats Abs, Negate, Sqrt, Square, Scale (multiply by **Param A**), Offset (add **Param A**), Clamp (between **Param A** and **Param B**); for vectors Normalize, Scale, Offset, Abs, Grid Snap; for strings To Upper, To Lower, Trim, Reverse.

Also there: Parallel Unique (remove duplicates), **Parallel Shuffle** (seeded, so repeatable), **Parallel Reverse**, **Parallel Merge** (append two arrays, optionally sorted), and the **Generate Random ...** nodes for test data.

Very large arrays, a piece at a time: Get Chunk Count and **Get Array Chunk** split an array into slices. Combine them with **Async Loop** (next recipe) to process one slice per frame.

6.9 A loop that never hitches (your own logic per item)

When each item needs your own Blueprint code, the work has to run on the Game Thread. The trick is to spread it over several frames instead of doing it all in one.

```
Async Loop  First Index = 0, Last Index = 9999, Iterations Per Frame = 200
  |-- Loop Body (Index) --> your per-item logic, using Index
  +-- Completed (Iterations, Elapsed Ms) --> "all done"
```

With 10,000 items and 200 per frame, the loop takes 50 frames (under a second at 60 fps) and no single frame does more than 200 items' worth of work. Tune **Iterations Per Frame** until the frame time is flat.

The family:

Node	Use it for
Async Loop	A For loop from First Index to Last Index, spread over frames
Async For Each (Indices)	Give it your array's length; Loop Body fires with each Index. Get the element with a Get node
Async Do N	Run the body N times
Async Sequence	One step per frame, in order: spawn wave 0, wave 1, wave 2...
Async Repeat	Fire the body every Interval Seconds, Count times (0 = forever)

Node	Use it for
Async While	Fire the body every Interval Seconds until you stop it
Async Delay	A Delay that reports progress, can be cancelled, and ignores pause and time dilation
Async Delay (Frames)	Wait a number of frames
Run Async	Continue on the next frame. Breaks a heavy chain in two

Stopping Async Repeat or Async While: drag off the node's **Task** pin and call **Break Loop** when your condition is met, or **Cancel** to stop immediately. Check the condition inside the Loop Body.

6.10 Save and load without a freeze

```
Your save text --> Async Save (Text)
                        File Path = "Saved/SaveGames/slot1.txt"
                        Text      = the string
                        Append    = false
                        |
                        +-- Completed --> show "Saved"
                        +-- Failed    --> show Error.Message
```

```
Async Load (Text)  File Path = "Saved/SaveGames/slot1.txt"
  +-- Completed --> Result (the file's text)
```

Relative paths start at your project folder, so Saved/... is YourProject/Saved/.... Missing folders are created. **Async Save (Bytes)** and **Async Load (Bytes)** do the same for binary data.

JSON: Async JSON Write checks that the string is valid JSON before writing, so you never save a broken file. **Async JSON Read** loads, validates, and returns the pretty-printed **Json** plus **Top Level Keys**. Invalid JSON fires Failed with the parser's message.

Encrypted, compressed save (a full pipeline):

```
Save text --> String to UTF-8 Bytes --> Compress (Bytes)  Format = LZ4
                                                +-- Completed --> Data --> Encrypt Bytes  Password = "...
                                                +-- Completed --> Bytes --> Async S
```

Loading is the reverse: **Async Load (Bytes)** > **Decrypt Bytes** > **Decompress (Bytes)** > **UTF-8 Bytes to String**. The compressed container records which codec was used, so Decompress needs no settings. A wrong password or a tampered file fires Failed instead of returning garbage.

File nodes cannot reach outside your project folder unless you change the setting in Project Settings. If a path is refused, Failed tells you which path and why. Chapter 9 explains the whole security model.

6.11 Import a spreadsheet (item tables, spawn tables, dialogue)

```
CSV Import (Async)  File Path = "Content/Data/items.csv", Delimiter = ",", First Row Is Header = true
  +-- Completed --> Rows (array of CSV Row), Headers, Row Count
```

Each **Row** has a **Cells** array of strings, one per column. So for a file with columns Name, Price, Weight:

```
ForEach (Rows) --> Cells[0] is the name, Cells[1] the price (String To Float), Cells[2] the weight
```

Quoted fields, escaped quotes and newlines inside quotes are handled. Hundreds of thousands of rows import without a hitch. **CSV Export (Async)** writes the other way: give it **Headers** and **Rows**, and any cell containing the delimiter, quotes or newlines is quoted for you.

6.12 Distances, heatmaps, noise and other world math

These work on plain arrays of points and are the building blocks behind a lot of procedural and analysis work.

Node	What you give it	What you get back
Distance Calculator (Async)	Points, Origin	Results: the distance of every point, same order
Nearest Point (Async)	Points, Target	Index, Point, Distance of the closest one
Closest Actor (Async)	Actors, Target	Actor, Index, Distance
Bounding Box (Async)	Points	Center, Extents, Box Min, Box Max (the box around a group)
Path Distance (Async)	Path Points	Total Length, plus Cumulative Lengths at every point
Grid Processing (Async)	Grid Size X/Y, Cell Size, Origin, Jitter Amount	Results: the grid's points, row by row
Heatmap Generator (Async)	Points, Bounds Min/Max, Width, Height, Radius	Values (one per cell), Width, Height
Noise Generator (Async)	Width, Height, Scale, Octaves, Persistence, Lacunarity, Seed	Values in 0..1, Width, Height
Voronoi Processing (Async)	Seeds (2D), Width, Height, Cell Size	Cell Owners: which seed owns each cell
Visibility Processing (Async)	Origin, Target Points, Trace Channel	Visible (true/false per point), Num Visible

Reading a grid result: heatmap, noise and voronoi return one flat array. The cell at column x, row y is at $\text{index } y * \text{Width} + x$.

Heatmap example: record player death positions in an array; every minute, feed them to **Heatmap Generator** with the map bounds and a 64 x 64 grid; the hottest cells are where players die most. With **Normalize** on, the hottest cell is exactly 1.0.

Noise example: **Noise Generator** with Width 256, Height 256, Scale 0.05, Octaves 4. Use the value at each cell as a spawn mask: place grass where noise is above 0.6, rocks where it is below 0.2. The same **Seed** always gives the same field.

Path progress example: for a race track spline, sample its points once, run **Path Distance**, and use **Cumulative Lengths** to turn "the car is at point 37" into "the car is 62 percent around the track".

6.13 Running several things at once, then continuing

Split setup work across frames:

```
Parallel Split --> Branch A (frame 1): spawn the environment
                --> Branch B (frame 2): build the UI
                --> Branch C (frame 3): load the save
                --> Branch D (frame 4): start the music
                --> Completed: everything above has run
```

Wait for several tasks: every async node's **Started** pin gives you a **Task Handle**. Collect them into an array, then:

```
Wait For All Tasks   Tasks = the array
  +-- Completed --> Finished Tasks   (all of them are done, failed or cancelled)
```

Example: start three **Async Load (Text)** nodes for three data files, add each Started handle to an array, then Wait For All Tasks before building the menu. **Wait For Any Task** completes as soon as the first one finishes.

Do things one after another, from anywhere: **Task Queue** with a name. Its **Execute** pin fires only when every earlier Task Queue node with the same name has run. Put one in front of every save operation with the name Saves, and saves never overlap, even if two Blueprints trigger them at once.

6.14 Cancelling, progress bars and cleanup

Cancel one task:

```
On Started: store Task Handle in a variable
Later:      Cancel Task (Task Handle)   (or drag off the node's Task pin and call Cancel)
```

The task stops at its next safe point and fires **Cancelled**. Threads are never force-killed, so nothing is left half-written.

Cancel everything: Cancel All Tasks. Good on level change or when the player quits to the menu. Returns how many tasks were told to stop.

Progress bar: on the **Progress** event, wire **Progress Value** to your widget's progress bar (it is already 0 to 1). **Get Task Progress (Task Handle)** reads it from anywhere, and **Is Task Running** tells you whether a handle is still live.

6.15 Images, screenshots, text and encryption

- **Screenshot Save (Async)** takes a screenshot and fires Completed when the PNG is on disk (relative names go to Saved/Screenshots). Then **Image Resize (Async)** makes a thumbnail from it, and **Pixel Processing (Async)** can grayscale, invert, brighten, add contrast or tint it. Useful for save-slot previews.
- **String Processing (Stats)** counts characters, words and lines of a big text. **String Processing (Find & Replace)** replaces every occurrence and tells you how many.
- **Encrypt String / Decrypt String** use AES-256 with a password and give you Base64 text you can put straight into JSON. **Encrypt Bytes / Decrypt Bytes** do the same for binary data. Good for making save files hard to edit. Not a substitute for server-side validation in a competitive game: the password is inside your game.

- **Bytes to Base64 / Hex** and back, **String to UTF-8 Bytes** and back: the glue between the text and binary worlds.

6.16 Checksums and hashes (did this file change?)

Checksum File (Async) File Path --> Completed --> Result (CRC32 as text)
 Hash File (Async) File Path, Algorithm = MD5 / SHA1 / CRC32 --> Result (hex text)

Store the checksum when you write a save, compare when you load it, and you know whether the file was edited or corrupted. **Hash Bytes** and **Hash String** hash data that is already in memory. Files are read in 1 MB pieces, so even huge files never load fully into memory.

Cheat sheet: which node for which job

You want to...	Use	Category
Find the closest enemy, or the closest few	Get Nearest Actors	Gameplay
Everything within a radius	Get Actors In Radius	Gameplay
Everything in front of me, in view	Get Actors In Cone	Gameplay
Which actors can I actually see	Get Visible Actors	Gameplay
Order actors by distance	Sort Actors By Distance	Gameplay
Sort structs, or anything, by a value	Sort By Key (Float / Integer)	Gameplay
Scatter things without overlaps	Scatter Points (Even Spacing)	Gameplay
Drop points onto the terrain	Snap Points To Ground	Gameplay
Sort, filter, find, count big arrays	Parallel Sort / Filter / Find / Count	Parallel Arrays
Sum, average, min, max	Combine Array	Parallel Arrays
Run my own logic per item without hitching	Async Loop, Async For Each (Indices)	Async Flow
Wait for several tasks, or run them in order	Wait For All Tasks, Task Queue	Async Flow
Save, load, JSON, CSV	Async Save / Load, JSON Read / Write, CSV Import / Export	File Processing
Compress, encrypt, hash	Compress, Encrypt, Hash / Checksum	File Processing, Utility
Heatmap, noise, voronoi, grid, bounds	Heatmap / Noise / Voronoi / Grid / Bounding Box	Math Processing
Not sure it is worth it	Advise	Advisor
Prove it helped	Benchmark Sort / Reduce / Nearest Point, ParallelFlow Benchmark actor	Benchmark

Right-click in any Blueprint and type what you want to do: nearest, visible, scatter, snap, sort, save, slow. The search finds it.

7. The Performance Advisor

The Advisor is a node that tells you, in plain language, whether optimizing a piece of Blueprint is worth it, and which node to use if so. It never pushes ParallelFlow where it does not help.

How to ask

```
Advise (Performance Advisor)
  Operation          = Sort an Array          (pick from the list)
  Element Count      = 50000
  Runs Every Frame   = false
  |
  +-- Return Value --> Advice To Text --> Print String
```

The **Operation** list covers: sort an array, filter, find, reduce (sum, avg, min, max), transform every element, distance to many points, nearest actor or point, actors within a radius, sort an array of structs, visibility and line of sight, procedural placement, heatmap, noise field, heavy ForEach loop, file read and write, hashing, compression.

What it answers

The result is an **Advice** struct with:

- **Use Parallel Flow:** true or false. The short answer.
- **Level:** Keep Using Blueprint, Consider ParallelFlow, Use ParallelFlow, or Strongly Recommended.
- **Headline:** one sentence.
- **Recommended Node** and **Replaces:** which node, and which Blueprint pattern it replaces.
- **Reason** and **Expected Gain:** why, and a realistic range.
- **Warning:** an extra caution when relevant, for example when it runs every frame.

Advice To Text turns all of that into a readable paragraph. For 50,000 elements in a sort it prints:

```
[Use ParallelFlow] Use Parallel Sort.
  Why: At 50000 elements a single-threaded Blueprint loop will visibly hitch the frame.
        Parallel Sort does the same work natively across CPU cores.
  Node: Parallel Sort (replaces: Blueprint Sort node / manual sort loop)
  Expected gain: ~2-5x on a typical multi-core CPU
```

And for 300 elements:

```
[Keep Using Blueprint] Keep using normal Blueprint (300 elements is small).
  Why: At 300 elements a Blueprint loop finishes in microseconds. ParallelFlow's setup
        would cost more than it saves. Optimize when your data grows past ~2000.
```

The thresholds it uses

These are deliberately conservative. The Advisor would rather tell you to keep Blueprint than send you chasing a speedup that does not exist.

Operation	Keep Blueprint below	Recommended from	Strongly recommended from
Sort an array	2,000	20,000	200,000
Filter an array	5,000	50,000	500,000
Find in an array	20,000	100,000	1,000,000
Reduce (sum, avg, min, max)	10,000	100,000	1,000,000
Transform every element	10,000	100,000	1,000,000
Distance to many points	2,000	20,000	200,000
Nearest actor or point	200	2,000	20,000
Actors within a radius	200	2,000	20,000
Sort an array of structs	1,000	10,000	100,000
Heavy ForEach loop	500	5,000	50,000

Visibility and line of sight, procedural placement, heatmaps, noise, file reading and writing, hashing and compression are treated as heavy work: the Advisor recommends ParallelFlow for them at any size, because Blueprint has no cheap way to do them at all. "Runs every frame" raises every recommendation by one level.

Log Advice writes the same text to the Output Log instead of the screen.

8. Measure it: benchmark, demo and debugger

Do not take anyone's word for it, including this guide. The plugin ships with tools to measure on your own machine.

The Benchmark actor

Drop a **ParallelFlow Benchmark** actor into any level (search for it in the Place Actors panel) and press Play. It runs three comparisons and prints them on screen:

- **Sort**: a single-threaded sort versus Parallel Sort, over **Sort Count** floats (default 500,000).
- **Reduce**: a single-threaded sum loop versus Combine Array, over **Reduce Count** floats (default 1,000,000).
- **Nearest point**: the classic "closest point" loop versus the parallel version, over **Nearest Count** points (default 200,000).

It also prints a couple of Advisor examples so you can see what the advice looks like. Change the counts in the Details panel to match your own data sizes.

The Benchmark nodes

The same comparisons are available as nodes: **Benchmark Sort**, **Benchmark Reduce (Sum)** and **Benchmark Nearest Point**, each taking a **Count** and a **Seed** and returning a result that **Benchmark To Text** formats for printing. They run synchronously by design, so use them in a test map, not in gameplay.

The Demo actor

The **ParallelFlow Demo** actor generates and sorts **Sort Count** floats on worker threads and prints the timings, while a separate stress task (**Run Stress Test**) keeps a background worker busy for several seconds. Watch the frame rate: it does not move. That is the whole point of the plugin, visible.

Honest numbers

On a 10-core desktop CPU, sorting 500,000 floats takes about **36 ms** the ordinary single-threaded way and about **13 ms** with ParallelFlow, and the game keeps rendering the whole time. That comparison is against already-fast C++, which is the conservative baseline. A plain Blueprint ForEach doing the same job is far slower than the C++ baseline, so the gap you see when replacing a real Blueprint loop is larger. Your numbers will differ with your CPU and your data. Measure yours.

The Debugger window

Tools > Miscellaneous > ParallelFlow Debugger shows every task: queued, running and recently finished, with name, state, progress, queue time, execution time, the thread it ran on, and any error message. Above the list are the scheduler statistics: running and queued counts, average and last execution times, worker thread count, an occupancy estimate, and memory use. Leave it open while you play and you will see exactly what the plugin is doing.

The same data is available in Blueprint under **ParallelFlow > Debug: Get Scheduler Stats, Get Active Tasks, Get Task History, Clear Task History** and **Log Scheduler State**.

Console commands

- `ParallelFlow.DumpTasks` writes the scheduler statistics and every active task to the Output Log.
- `ParallelFlow.CancelAll` cancels every running and queued task.

9. Security: what the file nodes can and cannot touch

If you are evaluating the plugin for a game that loads user-created content or mods, read this chapter. It explains exactly what the file nodes can reach, how to lock them down further, and what the plugin never does.

The short version

- File nodes are **limited to your project folder by default**. A Blueprint cannot read the player's documents, browse system folders, or touch other programs.
- The limit is enforced in one place that every file node goes through, so no single node can bypass it.
- Paths with `..` are collapsed **before** the check, so `../../../../Windows/...` style escapes are refused.
- Widening access is an explicit, visible choice in Project Settings.
- Deleting and moving files can be switched off entirely.

What the file nodes can access

Every node that touches disk (Async Save and Load, JSON, CSV, Copy, Move, Delete, Folder Scan, Compress and Decompress File, Hash and Checksum File, Image Resize, Pixel Processing, Screenshot Save) resolves its path through the same security layer.

Default policy: **Project Directory Only**.

Location	Allowed by default
Your project folder (Content, Saved, Config, ...)	Yes
Anywhere outside the project folder	No
Paths using <code>..</code> to climb out of the project	No (collapsed and refused)
Absolute system paths (C:\Windows, user documents, ...)	No

A refused path fires the node's **Failed** pin with a message naming the blocked path, and writes a warning to the log. It never silently does nothing.

Changing the policy

Edit > Project Settings > Plugins > ParallelFlow > File Security

- **File Access Mode**
- **Project Directory Only (recommended)**: the default, described above.
- **Project Directory + Allow List**: the project folder plus the specific folders you list in **Additional Allowed Directories**. Subfolders of a listed folder are included. Use this when you genuinely need an external data folder.

- **Unrestricted (not recommended)**: no path limit. Only for internal tools or editor utilities where you fully control what runs.
- **Additional Allowed Directories**: the list used by the second mode.
- **Allow Destructive File Operations** (on by default): switch it off and **File Delete** and **File Move** fail immediately, project-wide.

These are project settings saved to your config files, not Blueprint values, so game code cannot raise its own permissions at runtime.

Games that load mods or user-created content

The real threat model here is bigger than this plugin, and you should know it. **Unreal does not sandbox Blueprints**. If your game loads untrusted user Blueprints or pak files, those Blueprints can already call a large amount of engine functionality. ParallelFlow is one surface among many, and removing it does not make Blueprint modding safe. Any game that accepts user content needs a deliberate security design: curated content, signing, review, or a restricted loading path.

Within that, ParallelFlow gives you concrete controls:

1. Keep **File Access Mode** on Project Directory Only. Mods then cannot reach anything outside your game's own folder through these nodes.
2. Switch off **Allow Destructive File Operations** so nothing can delete or move files, even inside the project folder.
3. If your shipping build has no need for file nodes at all, disable the plugin in shipping configurations.

Who the file nodes are for

They are **developer tools**: for save systems, data import and export, caching, tooling and editor utilities. They are not meant as a general file system API to hand to end users or modders, and the default sandbox reflects that.

What ParallelFlow never does

The plugin contains no code that:

- opens network connections or sends any data anywhere,
- runs external programs or shell commands,
- loads or executes code,
- reads credentials, browser data or operating-system user data,
- writes outside the configured sandbox.

It uses only Unreal's own file APIs, the engine's compression and hashing, and the engine task system. There are no third-party libraries and no external dependencies. The gameplay, array and math nodes never touch the file system at all.

Reporting a security issue

If you find a way to escape the sandbox or otherwise misuse the file nodes, please report it privately rather than publicly, on the support Discord: <https://discord.gg/a3QdMX9JZj>

10. Troubleshooting and questions

"My Completed pin never fires." Look at the **Failed** pin. Wire it to a Print String showing **Error > Message**. There will be a sentence explaining what happened: an empty array, a blocked path, a count below 1, no world.

"I cannot find the node inside my function." Async nodes cannot be placed inside Blueprint Functions, only in Event Graphs (and Editor Utility graphs). Use a **Custom Event** instead of a Function.

"The array is empty right after the node." That is expected. The result arrives on **Completed**, a moment later. Move the logic that needs it onto the Completed pin. See chapter 4.

"Get Visible Actors says everything is hidden." Set **Ignore Actor** to the viewer (under the node's small arrow). Otherwise the trace starts inside the viewer's own collision and is blocked at once. Also check that walls block the trace channel you chose; Visibility is right for most games.

"Snap Points To Ground returns fewer points than I gave it." That is it working. Points that hit nothing between **Trace Up** and **Trace Down**, or that landed on a slope steeper than **Max Slope Degrees**, are removed so you never spawn something floating or on a cliff face. Raise Trace Down or Max Slope Degrees to get more of them back.

"Scatter Points gives me fewer points than Max Points." The area is full. With a Min Spacing of 300 in an 8000 x 8000 area there is only room for a few hundred points. Lower Min Spacing or enlarge the area.

"It says my file path was blocked." File nodes are limited to your project folder by default. Use a path inside the project (Saved/... for example), or change **File Access Mode** in Project Settings. This is deliberate, so a Blueprint can never read a player's personal files. Chapter 9.

"It is not any faster." Your array is probably too small. Ask the **Advise** node. If it says keep using Blueprint, believe it. Also make sure you are comparing like with like: measure the frame time, not just the node's Elapsed Ms, because the node's whole point is that the frame no longer waits.

"Can I use it in an Animation Blueprint?" Not directly. Do the heavy work in your Character or a manager Blueprint, store the result in a variable, and have the Anim Blueprint read that variable.

"Does it work in packaged games, and on consoles or mobile?" Yes in packaged games. Platforms: Windows, Mac, Linux, Android and iOS. The editor tools (debugger window) are editor-only, as you would expect.

"Does it use the GPU?" No. Everything runs on CPU worker threads and the engine's async physics traces.

"How many threads does it use?" It uses the engine's task system, which scales with the cores of the machine it runs on. You do not configure anything. **Get Scheduler Stats** reports the worker count.

"Is anything replicated in multiplayer?" No. Results are local. See chapter 4.

"Can I ship a Blueprint-only project with it?" Yes. Installed from Fab to the engine, it comes prebuilt. Installed from a zip into the project's Plugins folder, it needs a compiler on the machine that packages the game.

"I cannot find the nodes at all." Confirm the plugin is enabled in **Edit > Plugins**. Then right-click in a Blueprint and search for what you want to do: sort, nearest, visible, scatter, radius, save, slow. Everything lives under the **ParallelFlow** category.

11. Support and version notes

- **Version:** 2.1.1
- **Engines:** Unreal Engine 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8
- **Platforms:** Win64, Mac, Linux, Android, iOS (editor tools: Win64, Mac, Linux)
- **Distribution:** source; compiles on install. No third-party dependencies.
- **Support:** Discord, <https://discord.gg/a3QdMX9JZj>

Fixed in 2.1.1

- Every ParallelFlow node now shows the data pins for all of its events: Task Handle, Progress Value, the result pins (Results, Actors, Distances, Sorted Order, ...), Elapsed Ms and Error. Earlier builds showed only Task Handle because of how the engine's stock async node builds its pins, so the results of Completed could not be read in Blueprint. If you placed a ParallelFlow node with an earlier build, delete it and place it again.

What is new in 2.1

- The **Gameplay** category: Get Actors In Cone, Get Visible Actors, Scatter Points (Even Spacing) and Snap Points To Ground join Sort Actors By Distance, Get Actors In Radius, Get Nearest Actors and Sort By Key. These take Get All Actors Of Class output directly and cover the queries games run every day.
- The Performance Advisor knows about cone and radius queries, struct sorting, visibility and procedural placement.
- All documentation consolidated into this one guide.
- File nodes sandboxed to the project folder by default, with Project Settings to widen or lock down access.
- Compiles on UE 5.2 through 5.8.

Known limitations

- Async nodes cannot be used in Animation Blueprints, Construction Scripts or Blueprint Functions. Use Event Graphs, Custom Events, or Editor Utility Blueprints.
- Scatter Points runs on one worker thread (its algorithm is sequential by nature). It is off the Game Thread, but not spread across cores.
- UE 4.27 is not supported.

(c) Black Scar Studio 2026. All Rights Reserved.

12. Node reference (every node)

Every Blueprint node in the plugin, generated directly from the plugin source so this list always matches what is installed. Async nodes deliver their result on the Completed pin and also have Started (fills Task Handle, for cancelling), Progress (fills Progress Value, 0 to 1), Failed (fills Error, which has a Message) and Cancelled. The data pins sit under the exec pins; each event fills its own. Functions run immediately and return a value. Right-click in any Blueprint and search the node name, or browse the ParallelFlow category.

12.1 Gameplay

Blueprint category: ParallelFlow|Gameplay

Actor and world queries straight from Get All Actors Of Class, sorting your own structs, and procedural placement. These are the nodes most games use every day.

Sort Actors By Distance

Async node. Result arrives on the Completed pin.

Sorts actors by distance from Origin, nearest first. Plug Get All Actors Of Class straight in. Replaces a ForEach that measures the distance to every actor and then sorts. Returns the actors in order, plus their distances.

Inputs

Actors Array of Actor
Origin Vector
Nearest First Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Actors Array of Actor
Distances Array of Float
Elapsed Ms Float

Get Actors In Radius

Async node. Result arrives on the Completed pin.

Returns every actor within Radius of Origin, optionally sorted nearest first. Ideal for AoE damage, nearby-enemy checks, pickup ranges and interaction prompts across hundreds or thousands of actors. This is a distance test, not a physics overlap.

Inputs

Actors Array of Actor
Origin Vector
Radius Float (default 1000)
Sort By Distance Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Actors Array of Actor
Distances Array of Float
Elapsed Ms Float

Get Nearest Actors

Async node. Result arrives on the Completed pin.

Returns the Count actors closest to Origin, nearest first. The standard AI targeting query: give me the 5 closest enemies. Far cheaper than sorting everything yourself in Blueprint.

Inputs

Actors Array of Actor
Origin Vector
Count Integer (default 5)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Actors Array of Actor
Distances Array of Float
Elapsed Ms Float

Get Actors In Cone

Async node. Result arrives on the Completed pin.

Returns every actor inside a cone: within Radius of Origin AND within Cone Half Angle degrees of Direction. Use the actor's Forward Vector as Direction for what-is-in-front-of-me checks: enemy field of view, aim assist, lock-on targeting, melee arcs. Nearest first when Sort By Distance is on.

Inputs

Actors Array of Actor
Origin Vector
Direction Vector
Cone Half Angle Degrees Float (default 45)
Radius Float (default 2000)
Sort By Distance Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Actors Array of Actor
Distances Array of Float
Elapsed Ms Float

Get Visible Actors

Async node. Result arrives on the Completed pin.

Returns the actors that have a clear line of sight from Eye Location, nearest first. Actors beyond Max Distance are skipped before any trace runs (set 0 for unlimited). Each remaining actor gets one asynchronous physics trace on the chosen channel, so the Game Thread never stalls. A trace that reaches the actor itself counts as visible. Set Ignore Actor to the viewer so its own collision does not block the traces.

Inputs

Actors Array of Actor
Eye Location Vector
Max Distance Float
Trace Channel Enum
Options: Visibility, Camera, and any custom trace channels in your project
Ignore Actor Actor (default None)
Trace Complex Boolean (default false)

Outputs (on Completed)

Actors Array of Actor
Distances Array of Float
Elapsed Ms Float

Snap Points To Ground

Async node. Result arrives on the Completed pin.

Drops every point onto the surface below it using asynchronous physics traces. Each trace starts Trace Up above the point and goes Trace Down below it. Points that hit nothing, or land on a slope steeper than Max Slope Degrees, are left out of the result, so what comes back is ready to spawn on. Also returns the surface normal under each point for aligning rotation to the ground. Pair it with Scatter Points or Grid Processing for procedural placement.

Inputs

Points Array of Vector

Trace Channel Enum

Options: Visibility, Camera, and any custom trace channels in your project

Trace Up Float (default 1000)

Trace Down Float (default 10000)

Max Slope Degrees Float (default 90)

Trace Complex Boolean (default false)

Outputs (on Completed)

Snapped Points Array of Vector

Surface Normals Array of Vector

Elapsed Ms Float

Scatter Points (Even Spacing)

Async node. Result arrives on the Completed pin.

Fills a flat rectangle (Area Size wide and deep, centered on Center) with random points that are all at least Min Spacing apart, so nothing overlaps and coverage looks natural rather than clumpy. Max Points of 0 means as many as fit. The same Random Seed always gives the same layout. Feed the result into Snap Points To Ground to place things on terrain.

Inputs

Center Vector

Area Size Vector 2D

Min Spacing Float (default 200)

Max Points Integer (default 0)

Random Seed Integer (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector

Elapsed Ms Float

Sort By Key (Float)

Async node. Result arrives on the Completed pin.

Sorts a list of float keys and returns the ORDER, so you can sort ANY array - including arrays of your own structs. Build one array holding the value to sort on (score, item value, weight), plug it in, then read your original array using the returned Sorted Order. This is how you sort structs fast.

Inputs

Keys Array of Float

Ascending Boolean (default true)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Sorted Order Array of Integer

Elapsed Ms Float

Sort By Key (Integer)

Async node. Result arrives on the Completed pin.

Sorts a list of integer keys and returns the ORDER, so you can sort ANY array - including arrays of your own structs. Read your original array using the returned Sorted Order.

Inputs

Keys Array of Integer

Ascending Boolean (default true)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Sorted Order Array of Integer

Elapsed Ms Float

12.2 Parallel Arrays

Blueprint category: ParallelFlow|Parallel Arrays

Sort, filter, transform, combine, find, count, shuffle and merge over Float, Integer, Vector and String arrays. Every type has the same set of nodes with the same pins.

Parallel Sort (Float)

Async node. Result arrives on the Completed pin.

Sorts a float array on background worker threads. Handles hundreds of thousands of elements without hitching the game.

Inputs

Values Array of Float
Ascending Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float
Elapsed Ms Float

Parallel Filter (Float)

Async node. Result arrives on the Completed pin.

Keeps elements between Min and Max (inclusive), preserving order. Runs on background worker threads.

Inputs

Values Array of Float
Min Float (default 0)
Max Float (default 1)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float
Elapsed Ms Float

Transform Array (Float)

Async node. Result arrives on the Completed pin.

Applies a built-in transform (abs, negate, sqrt, square, scale, offset, clamp) to every element on background worker threads. Blueprint code cannot run on worker threads, so transforms are built-in and thread-safe.

Inputs

Values Array of Float
Operation Enum
Options: Abs, Negate, Sqrt, Square, Scale, Offset, Clamp
Param A Float (default 1)
Param B Float (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float
Elapsed Ms Float

Parallel Unique (Float)

Async node. Result arrives on the Completed pin.

Removes duplicate values within the given tolerance. Note: the result is sorted ascending.

Inputs

Values Array of Float

Tolerance Float (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float

Elapsed Ms Float

Parallel Shuffle (Float)

Async node. Result arrives on the Completed pin.

Shuffles the array on a background thread using a seeded Fisher-Yates shuffle. The same seed always produces the same order.

Inputs

Values Array of Float

Seed Integer (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float

Elapsed Ms Float

Parallel Reverse (Float)

Async node. Result arrives on the Completed pin.

Reverses the array order on a background thread.

Inputs

Values Array of Float

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float

Elapsed Ms Float

Parallel Merge (Float)

Async node. Result arrives on the Completed pin.

Appends array B after array A on a background thread. Enable Sort Result to sort the merged array ascending.

Inputs

A Array of Float

B Array of Float

Sort Result Boolean (default false)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float

Elapsed Ms Float

Parallel Sort (Integer)

Async node. Result arrives on the Completed pin.

Sorts an integer array on background worker threads.

Inputs

Values Array of Integer

Ascending Boolean (default true)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Filter (Integer)

Async node. Result arrives on the Completed pin.

Keeps elements between Min and Max (inclusive), preserving order.

Inputs

Values Array of Integer

Min Integer (default 0)

Max Integer (default 100)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Transform Array (Integer)

Async node. Result arrives on the Completed pin.

Applies a built-in transform (abs, negate, scale, offset, clamp, modulo) to every element on background worker threads.

Inputs

Values Array of Integer

Operation Enum

Options: Abs, Negate, Scale, Offset, Clamp, Modulo

Param A Integer (default 1)

Param B Integer (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Unique (Integer)

Async node. Result arrives on the Completed pin.

Removes duplicate values, keeping the first occurrence of each and preserving order.

Inputs

Values Array of Integer

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Shuffle (Integer)

Async node. Result arrives on the Completed pin.

Shuffles the array on a background thread. The same seed always produces the same order.

Inputs

Values Array of Integer

Seed Integer (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Reverse (Integer)

Async node. Result arrives on the Completed pin.

Reverses the array order on a background thread.

Inputs

Values Array of Integer

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Merge (Integer)

Async node. Result arrives on the Completed pin.

Appends array B after array A on a background thread. Enable Sort Result to sort the merged array ascending.

Inputs

A Array of Integer

B Array of Integer

Sort Result Boolean (default false)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer

Elapsed Ms Float

Parallel Sort (Vector)

Async node. Result arrives on the Completed pin.

Sorts a vector array on background threads by length or by a single component.

Inputs

Values Array of Vector

Sort Mode Enum (default Length)

Options: Length, X, Y, Z

Ascending Boolean (default true)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector

Elapsed Ms Float

Parallel Filter (Vector)

Async node. Result arrives on the Completed pin.

Keeps points whose distance to Center is between Min Distance and Max Distance, preserving order. Ideal for radius queries on large point sets.

Inputs

Values Array of Vector
Center Vector (default (0, 0, 0))
Min Distance Float (default 0)
Max Distance Float (default 1000)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector
Elapsed Ms Float

Transform Array (Vector)

Async node. Result arrives on the Completed pin.

Applies a built-in transform (normalize, scale, offset, abs, grid snap) to every vector on background worker threads.

Inputs

Values Array of Vector
Operation Enum
Options: Normalize, Scale, Offset, Abs, GridSnap
Scale Vector (default (1, 1, 1))
Offset Vector (default (0, 0, 0))
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector
Elapsed Ms Float

Parallel Unique (Vector)

Async node. Result arrives on the Completed pin.

Removes duplicate points, keeping first occurrences. With a tolerance above 0, points are considered duplicates when they fall into the same tolerance-sized grid cell.

Inputs

Values Array of Vector
Tolerance Float (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector
Elapsed Ms Float

Parallel Shuffle (Vector)

Async node. Result arrives on the Completed pin.

Shuffles the array on a background thread. The same seed always produces the same order.

Inputs

Values Array of Vector

Seed Integer (default 0)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector

Elapsed Ms Float

Parallel Reverse (Vector)

Async node. Result arrives on the Completed pin.

Reverses the array order on a background thread.

Inputs

Values Array of Vector

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector

Elapsed Ms Float

Parallel Merge (Vector)

Async node. Result arrives on the Completed pin.

Appends array B after array A on a background thread.

Inputs

A Array of Vector

B Array of Vector

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector

Elapsed Ms Float

Parallel Sort (String)

Async node. Result arrives on the Completed pin.

Sorts a string array lexicographically on background worker threads.

Inputs

Values Array of String
Ascending Boolean (default true)
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Parallel Filter (String)

Async node. Result arrives on the Completed pin.

Keeps strings that match the chosen mode (contains / not contains / starts with / ends with the pattern, or length between Min Length and Max Length), preserving order.

Inputs

Values Array of String
Mode Enum
Options: Contains, NotContains, StartsWith, EndsWith, LengthBetween
Pattern String
Min Length Integer (default 0)
Max Length Integer (default 2147483647)
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Transform Array (String)

Async node. Result arrives on the Completed pin.

Applies a built-in transform (to upper, to lower, trim, reverse) to every string on background worker threads.

Inputs

Values Array of String
Operation Enum
Options: ToUpper, ToLower, Trim, Reverse
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Parallel Unique (String)

Async node. Result arrives on the Completed pin.

Removes duplicate strings, keeping the first occurrence of each and preserving order.

Inputs

Values Array of String
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Parallel Shuffle (String)

Async node. Result arrives on the Completed pin.

Shuffles the array on a background thread. The same seed always produces the same order.

Inputs

Values Array of String
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Parallel Reverse (String)

Async node. Result arrives on the Completed pin.

Reverses the array order on a background thread.

Inputs

Values Array of String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Parallel Merge (String)

Async node. Result arrives on the Completed pin.

Appends array B after array A on a background thread. Enable Sort Result to sort the merged array ascending (case-insensitive).

Inputs

A Array of String

B Array of String

Sort Result Boolean (default false)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String

Elapsed Ms Float

Combine Array (Float)

Async node. Result arrives on the Completed pin.

Reduces a float array to a single value on background worker threads. Accumulates in double precision, so very large arrays stay accurate.

Inputs

Values Array of Float

Operation Enum

Options: Sum, Average, Min, Max, Product

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Float

Elapsed Ms Float

Combine Array (Integer)

Async node. Result arrives on the Completed pin.

Reduces an integer array to a single 64-bit value on background worker threads. Average truncates toward zero; Product may overflow for very large inputs.

Inputs

Values Array of Integer

Operation Enum

Options: Sum, Average, Min, Max, Product

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer 64

Elapsed Ms Float

Combine Array (Vector)

Async node. Result arrives on the Completed pin.

Reduces a vector array to a single vector on background worker threads. Average yields the centroid of a point cloud.

Inputs

Values Array of Vector

Operation Enum

Options: Sum, Average, MinComponents, MaxComponents

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Vector

Elapsed Ms Float

Parallel Find (Float)

Async node. Result arrives on the Completed pin.

Finds the first index whose value is within Tolerance of Value, scanning on background worker threads. Completed returns -1 when nothing matches.

Inputs

Values Array of Float

Value Float

Tolerance Float (default 0.0001)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer

Elapsed Ms Float

Parallel Find (Integer)

Async node. Result arrives on the Completed pin.

Finds the first index equal to Value, scanning on background worker threads. Completed returns -1 when nothing matches.

Inputs

Values Array of Integer

Value Integer

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer

Elapsed Ms Float

Parallel Find (String)

Async node. Result arrives on the Completed pin.

Finds the first index equal to Value, scanning on background worker threads. Completed returns -1 when nothing matches.

Inputs

Values Array of String
Value String
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Parallel Find (Vector)

Async node. Result arrives on the Completed pin.

Finds the first index whose point lies within Tolerance of Point, scanning on background worker threads. Completed returns -1 when nothing matches.

Inputs

Values Array of Vector
Point Vector
Tolerance Float (default 0.01)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Parallel Search (String Contains)

Async node. Result arrives on the Completed pin.

Finds the first index whose string contains the substring, scanning on background worker threads. Completed returns -1 when nothing matches.

Inputs

Values Array of String
Substring String
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Parallel Count (Float)

Async node. Result arrives on the Completed pin.

Counts elements between Min and Max (inclusive) on background worker threads.

Inputs

Values Array of Float
Min Float (default 0)
Max Float (default 1)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Parallel Count (Integer)

Async node. Result arrives on the Completed pin.

Counts elements between Min and Max (inclusive) on background worker threads.

Inputs

Values Array of Integer
Min Integer (default 0)
Max Integer (default 100)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Parallel Count (String Contains)

Async node. Result arrives on the Completed pin.

Counts strings containing the substring on background worker threads.

Inputs

Values Array of String
Substring String
Case Sensitive Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Integer
Elapsed Ms Float

Get Chunk Count

Pure function (no exec pins).

How many chunks an array of Element Count elements splits into at the given Chunk Size. Pair with Get Float/Integer/Vector/String Array Chunk to page through large arrays.

Inputs

Element Count Integer

Chunk Size Integer

Outputs

Return Value Integer

Get Array Chunk (Float)

Pure function (no exec pins).

Returns the Chunk Index-th slice of Chunk Size elements. Use with Get Chunk Count to process a large array in batches (e.g. one chunk per frame via Async Loop).

Inputs

Values Array of Float

Chunk Size Integer

Chunk Index Integer

Outputs

Return Value Array of Float

Get Array Chunk (Integer)

Pure function (no exec pins).

Returns the Chunk Index-th slice of Chunk Size elements of an integer array.

Inputs

Values Array of Integer

Chunk Size Integer

Chunk Index Integer

Outputs

Return Value Array of Integer

Get Array Chunk (Vector)

Pure function (no exec pins).

Returns the Chunk Index-th slice of Chunk Size elements of a vector array.

Inputs

Values Array of Vector

Chunk Size Integer

Chunk Index Integer

Outputs

Return Value Array of Vector

Get Array Chunk (String)

Pure function (no exec pins).

Returns the Chunk Index-th slice of Chunk Size elements of a string array.

Inputs

Values Array of String

Chunk Size Integer

Chunk Index Integer

Outputs

Return Value Array of String

12.3 Math Processing

Blueprint category: ParallelFlow|Math Processing

Distances, nearest point, grids, heatmaps, noise, voronoi, path length, visibility and bounds over large point sets.

Distance Calculator (Async)

Async node. Result arrives on the Completed pin.

Computes the distance from every point to Origin on background worker threads. The result array is parallel to the input array.

Inputs

Points Array of Vector

Origin Vector

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float

Elapsed Ms Float

Nearest Point (Async)

Async node. Result arrives on the Completed pin.

Finds the point nearest to Target across a large point set on background worker threads.

Inputs

Points Array of Vector

Target Vector

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Index Integer

Point Vector

Distance Float

Elapsed Ms Float

Closest Actor (Async)

Async node. Result arrives on the Completed pin.

Finds the actor nearest to Target among hundreds or thousands of actors. Positions are snapshotted on the Game Thread when the node runs, then searched on background workers - actors are never touched from worker threads.

Inputs

Actors Array of Actor
Target Vector
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Actor Actor
Index Integer
Distance Float
Elapsed Ms Float

Grid Processing (Async)

Async node. Result arrives on the Completed pin.

Generates a world-space point grid (row-major, index = $Y * \text{GridSizeX} + X$) on a background worker. Jitter randomly offsets each point up to the given amount - great for organic spawn patterns.

Inputs

Grid Size X Integer (default 10)
Grid Size Y Integer (default 10)
Cell Size Float (default 100)
Origin Vector (default (0, 0, 0))
Jitter Amount Float (default 0)
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector
Elapsed Ms Float

Heatmap Generator (Async)

Async node. Result arrives on the Completed pin.

Splats points (using their X/Y) into a Width x Height density grid covering Bounds Min..Bounds Max. Each point contributes within Radius with smooth falloff. Normalize scales the hottest cell to 1.0. Great for player-activity or threat maps.

Inputs

Points Array of Vector
Bounds Min Vector 2D
Bounds Max Vector 2D
Width Integer (default 64)
Height Integer (default 64)
Radius Float (default 200)
Normalize Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Values Array of Float
Width Integer
Height Integer
Elapsed Ms Float

Noise Generator (Async)

Async node. Result arrives on the Completed pin.

Generates a Width x Height multi-octave Perlin noise field on background workers, remapped to 0..1. The same seed always produces the same field - ideal for procedural terrain and spawn masks.

Inputs

Width Integer (default 128)
Height Integer (default 128)
Scale Float (default 0.05)
Octaves Integer (default 4)
Persistence Float (default 0.5)
Lacunarity Float (default 2)
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Values Array of Float
Width Integer
Height Integer
Elapsed Ms Float

Voronoi Processing (Async)

Async node. Result arrives on the Completed pin.

Assigns every cell of a Width x Height grid (cells are Cell Size units) to its nearest seed point on background workers. Use for territory maps, biome regions or destruction patterns.

Inputs

Seeds Array of Vector 2D
Width Integer (default 128)
Height Integer (default 128)
Cell Size Float (default 100)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Cell Owners Array of Integer
Width Integer
Height Integer
Elapsed Ms Float

Path Distance (Async)

Async node. Result arrives on the Completed pin.

Measures a polyline path on a background worker: the total length plus the cumulative length at every point (useful for distance-along-path lookups).

Inputs

Path Points Array of Vector
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Total Length Float
Cumulative Lengths Array of Float
Elapsed Ms Float

Visibility Processing (Async)

Async node. Result arrives on the Completed pin.

Checks line of sight from Origin to every target point using the engine's asynchronous physics traces - no Game Thread blocking, even for hundreds of points. A point is visible when nothing blocks the trace on the chosen channel.

Inputs

Origin Vector

Target Points Array of Vector

Trace Channel Enum

Options: Visibility, Camera, and any custom trace channels in your project

Ignore Actor Actor (default None)

Trace Complex Boolean (default false)

Outputs (on Completed)

Visible Array of Boolean

Num Visible Integer

Elapsed Ms Float

Bounding Box (Async)

Async node. Result arrives on the Completed pin.

Computes the axis-aligned bounding box of a large point set on background worker threads.

Inputs

Points Array of Vector

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Center Vector

Extents Vector

Box Min Vector

Box Max Vector

Elapsed Ms Float

12.4 File Processing

Blueprint category: ParallelFlow|File Processing

Save, load, JSON, CSV, copy, move, delete, folder scan, compression and hashing. All off the Game Thread, and sandboxed to your project folder by default (see the Security chapter).

Async Save (Text)

Async node. Result arrives on the Completed pin.

Writes text to a UTF-8 file on a worker thread. Missing directories are created. Relative paths resolve against the project directory.

Inputs

File Path String
Text String
Append Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

Async Save (Bytes)

Async node. Result arrives on the Completed pin.

Writes a byte array to a file on a worker thread. Missing directories are created. Relative paths resolve against the project directory.

Inputs

File Path String
Bytes Array of Byte
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

Async Load (Text)

Async node. Result arrives on the Completed pin.

Reads a text file on a worker thread and returns its content as a string. Relative paths resolve against the project directory.

Inputs

File Path String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String
Elapsed Ms Float

Async Load (Bytes)

Async node. Result arrives on the Completed pin.

Reads a binary file on a worker thread and returns its bytes. Relative paths resolve against the project directory.

Inputs

File Path String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Bytes Array of Byte

Elapsed Ms Float

Async JSON Read

Async node. Result arrives on the Completed pin.

Loads a JSON file on a worker thread, validates it and returns the pretty-printed document plus its top-level keys. Invalid JSON fires Failed with the parser error.

Inputs

File Path String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Json String

Top Level Keys Array of String

Elapsed Ms Float

Async JSON Write

Async node. Result arrives on the Completed pin.

Validates a JSON string on a worker thread and writes it to disk (pretty-printed or compact). Invalid JSON fires Failed instead of writing a broken file.

Inputs

File Path String

Json String String

Pretty Print Boolean (default true)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String

Elapsed Ms Float

CSV Import (Async)

Async node. Result arrives on the Completed pin.

Parses a CSV file on a worker thread, honoring quoted fields, escaped quotes and embedded newlines. Handles hundreds of thousands of rows without hitching.

Inputs

File Path String
Delimiter String (default ",")
First Row Is Header Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Rows Array of CSV Row
Headers Array of String
Row Count Integer
Elapsed Ms Float

CSV Export (Async)

Async node. Result arrives on the Completed pin.

Writes header and rows to a CSV file on a worker thread. Fields containing the delimiter, quotes or newlines are quoted automatically. Leave Headers empty to skip the header row.

Inputs

File Path String
Headers Array of String
Rows Array of CSV Row
Delimiter String (default ",")
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

File Copy (Async)

Async node. Result arrives on the Completed pin.

Copies a file on a worker thread in 1 MB chunks with progress and cancellation. Cancelling removes the partial destination file.

Inputs

Source Path String
Destination Path String
Overwrite Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

File Move (Async)

Async node. Result arrives on the Completed pin.

Moves (renames) a file on a worker thread. Works across drives (the engine falls back to copy + delete).

Inputs

Source Path String
Destination Path String
Overwrite Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

File Delete (Async)

Async node. Result arrives on the Completed pin.

Deletes a file on a worker thread. Fails with a descriptive error if the file does not exist or cannot be removed.

Inputs

File Path String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String
Elapsed Ms Float

Folder Scan (Async)

Async node. Result arrives on the Completed pin.

Lists files and/or subdirectories matching a wildcard pattern (e.g. *.sav) on a worker thread, optionally recursing into subfolders.

Inputs

Directory String
Pattern String (default "**")
Recursive Boolean (default false)
Include Files Boolean (default true)
Include Directories Boolean (default false)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of String
Elapsed Ms Float

Compress (Bytes)

Async node. Result arrives on the Completed pin.

Compresses a byte array on a worker thread into a self-describing container (Decompress reads the codec automatically). Choose LZ4 for speed, Oodle or Zlib for ratio.

Inputs

Data Array of Byte

Format Enum (default Zlib)

Options: Zlib, Gzip, LZ4, Oodle

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Data Array of Byte

Original Size Integer

Compression Ratio Float

Elapsed Ms Float

Decompress (Bytes)

Async node. Result arrives on the Completed pin.

Decompresses a ParallelFlow container produced by the Compress nodes on a worker thread. The codec is read from the container header.

Inputs

Data Array of Byte

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Data Array of Byte

Original Size Integer

Compression Ratio Float

Elapsed Ms Float

Compress File (Async)

Async node. Result arrives on the Completed pin.

Reads a file, compresses it and writes the container to Destination Path, all on a worker thread. Completed returns the compressed bytes' size info.

Inputs

Source Path String
Destination Path String
Format Enum (default Zlib)
Options: Zlib, Gzip, LZ4, Oodle
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Data Array of Byte
Original Size Integer
Compression Ratio Float
Elapsed Ms Float

Decompress File (Async)

Async node. Result arrives on the Completed pin.

Reads a ParallelFlow container file, decompresses it and writes the original data to Destination Path, all on a worker thread.

Inputs

Source Path String
Destination Path String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Data Array of Byte
Original Size Integer
Compression Ratio Float
Elapsed Ms Float

Hash File (Async)

Async node. Result arrives on the Completed pin.

Computes an MD5 / SHA-1 / CRC32 digest of a file on a worker thread, streaming in 1 MB chunks so even huge files never load fully into memory.

Inputs

File Path String

Algorithm Enum (default MD5)

Options: MD5, SHA1, CRC32

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String

Elapsed Ms Float

Hash Bytes (Async)

Async node. Result arrives on the Completed pin.

Computes an MD5 / SHA-1 / CRC32 digest of a byte array on a worker thread.

Inputs

Bytes Array of Byte

Algorithm Enum (default MD5)

Options: MD5, SHA1, CRC32

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String

Elapsed Ms Float

Hash String (Async)

Async node. Result arrives on the Completed pin.

Computes an MD5 / SHA-1 / CRC32 digest of a string (UTF-8 encoded) on a worker thread.

Inputs

Text String

Algorithm Enum (default MD5)

Options: MD5, SHA1, CRC32

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String

Elapsed Ms Float

Checksum File (Async)

Async node. Result arrives on the Completed pin.

Computes the CRC32 checksum of a file on a worker thread. Fast change detection for save files and downloads.

Inputs

File Path String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String

Elapsed Ms Float

12.5 Async Flow

Blueprint category: ParallelFlow|Async Flow

Frame-spread loops, delays, background work, splits, joins, queues and task control, for keeping your own Blueprint logic hitch-free.

Run Async

Async node. Result arrives on the Completed pin.

Continues execution on the NEXT frame, off the current call stack. Use it to break heavy Blueprint chains apart so a single frame never does all the work.

Outputs (on Completed)

Elapsed Ms Float

Async Delay (Frames)

Async node. Result arrives on the Completed pin.

Continues execution after the given number of frames. Cancellable, with per-frame progress.

Inputs

Frame Count Integer (default 1)

Outputs (on Completed)

Elapsed Ms Float

Async Delay

Async node. Result arrives on the Completed pin.

Waits the given number of REAL seconds with progress events and cancellation. Unlike the stock Delay node it reports progress, returns a task handle and is unaffected by pause or time dilation.

Inputs

Duration Float (default 1)

Outputs (on Completed)

Elapsed Ms Float

Run Background Task

Async node. Result arrives on the Completed pin.

Runs a deterministic CPU workload (transcendental math) on a background worker thread. Use it to benchmark, test priorities and cancellation, and to prove the Game Thread stays smooth. Completed returns a checksum that is identical for identical iteration counts.

Inputs

Iterations Integer 64 (default 1000000)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Result Float

Elapsed Ms Float

Break Loop

Function (runs immediately).

Ends the loop gracefully. Completed fires with the number of iterations that ran. Use Cancel instead to end via the Cancelled pin.

Async Loop

Async node. Result arrives on the Completed pin.

A For loop spread across frames: runs Iterations Per Frame loop bodies each frame from First Index to Last Index. The game keeps rendering while the loop runs.

Inputs

First Index Integer (default 0)

Last Index Integer (default 99)

Iterations Per Frame Integer (default 10)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Async Do N

Async node. Result arrives on the Completed pin.

Runs the loop body N times (indices 0..N-1), spread across frames.

Inputs

N Integer (default 10)

Iterations Per Frame Integer (default 10)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Async For Each (Indices)

Async node. Result arrives on the Completed pin.

Iterates indices 0..Array Length-1, spread across frames. Wire Loop Body's Index into your array Get. Blueprint bodies always run on the Game Thread - for true multi-threaded array work use the Parallel Arrays nodes.

Inputs

Array Length Integer (default 0)

Iterations Per Frame Integer (default 10)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Async Repeat

Async node. Result arrives on the Completed pin.

Fires the loop body every Interval Seconds (real time). Count of 0 repeats forever until Break Loop or Cancel.

Inputs

Count Integer (default 0)

Interval Seconds Float (default 1)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Async While

Async node. Result arrives on the Completed pin.

Fires the loop body every Interval Seconds until you call Break Loop (condition became false) or Cancel. Check your condition inside the loop body.

Inputs

Interval Seconds Float (default 0)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Async Sequence

Async node. Result arrives on the Completed pin.

Runs Step Count steps, one per frame, firing Loop Body with the step index. Perfect for staged initialization (spawn wave 0, wave 1, ...).

Inputs

Step Count Integer (default 5)

Extra output pin: Loop Body

Index Integer

Outputs (on Completed)

Iterations Integer

Elapsed Ms Float

Parallel Split

Async node. Result arrives on the Completed pin.

Fires Branch A..D on four consecutive frames, then Completed. Spreads the cost of independent setup work across frames instead of stacking it into one. Leave unused branches unconnected.

Extra output pin: Branch A

(exec only)

Extra output pin: Branch B

(exec only)

Extra output pin: Branch C

(exec only)

Extra output pin: Branch D

(exec only)

Outputs (on Completed)

Elapsed Ms Float

Wait For All Tasks

Async node. Result arrives on the Completed pin.

Waits until every watched task reached a terminal state (Completed, Failed or Cancelled). Collect handles from other nodes' Started events to build task groups.

Inputs

Tasks Array of Task Handle

Outputs (on Completed)

Finished Tasks Array of Task Handle

Elapsed Ms Float

Wait For Any Task

Async node. Result arrives on the Completed pin.

Waits until the FIRST watched task reaches a terminal state, then completes with that handle. The other tasks keep running.

Inputs

Tasks Array of Task Handle

Outputs (on Completed)

Finished Tasks Array of Task Handle

Elapsed Ms Float

Task Queue

Async node. Result arrives on the Completed pin.

Joins the named FIFO queue. Execute fires when every earlier Task Queue node with the same name has run - use it to serialize work (saves, spawn waves, network requests) from anywhere in your project.

Inputs

Queue Name Name (default "Default")

Extra output pin: Execute

(exec only)

Outputs (on Completed)

Elapsed Ms Float

Cancel Task

Function (runs immediately).

Requests cancellation of the task identified by the handle (from a node's Started event). The task stops at its next safe point and fires its Cancelled pin. Returns false if the task already finished.

Inputs

Task Handle Task Handle

Outputs

Return Value Boolean

Cancel All Tasks

Function (runs immediately).

Requests cancellation of every running and queued ParallelFlow task. Each fires its Cancelled pin as it stops. Returns the number of tasks signalled.

Outputs

Return Value Integer

Is Task Running

Pure function (no exec pins).

True while the task identified by the handle is still queued or running.

Inputs

Task Handle Task Handle

Outputs

Return Value Boolean

Get Task Progress

Pure function (no exec pins).

Last reported progress (0..1) of the task, or -1 if the task is no longer active.

Inputs

Task Handle Task Handle

Outputs

Return Value Float

Is Valid Task Handle

Pure function (no exec pins).

True if this handle was produced by a task. Does NOT mean the task is still running - use Is Task Running for that.

Inputs

Task Handle Task Handle

Outputs

Return Value Boolean

12.6 Utility

Blueprint category: ParallelFlow|Utility

Random data generators, images, screenshots, text statistics, find and replace, encryption, encoding helpers, array chunking and memory estimates.

Generate Random Floats (Async)

Async node. Result arrives on the Completed pin.

Generates Count random floats in [Min, Max] on a background thread. The same seed always produces the same sequence. Great for feeding the parallel array nodes with test data.

Inputs

Count Integer (default 1000)
Min Float (default 0)
Max Float (default 1)
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Float
Elapsed Ms Float

Generate Random Integers (Async)

Async node. Result arrives on the Completed pin.

Generates Count random integers in [Min, Max] on a background thread. The same seed always produces the same sequence.

Inputs

Count Integer (default 1000)
Min Integer (default 0)
Max Integer (default 100)
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Integer
Elapsed Ms Float

Generate Random Points In Box (Async)

Async node. Result arrives on the Completed pin.

Generates Count random points inside the box spanned by Box Min and Box Max on a background thread. The same seed always produces the same points.

Inputs

Count Integer
Box Min Vector
Box Max Vector
Seed Integer (default 0)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Results Array of Vector
Elapsed Ms Float

Bytes to Base64

Pure function (no exec pins).

Encodes a byte array as a Base64 string (safe for JSON and text files).

Inputs

Bytes Array of Byte

Outputs

Return Value String

Base64 to Bytes

Function (runs immediately).

Decodes a Base64 string back into bytes. Returns false if the input is not valid Base64.

Inputs

Base64 String

Outputs

Return Value Boolean

Out Bytes Array of Byte

Bytes to Hex

Pure function (no exec pins).

Encodes a byte array as a lowercase hexadecimal string.

Inputs

Bytes Array of Byte

Outputs

Return Value String

Hex to Bytes

Function (runs immediately).

Decodes a hexadecimal string back into bytes. Returns false if the input has odd length or non-hex characters.

Inputs

Hex String

Outputs

Return Value Boolean

Out Bytes Array of Byte

String to UTF-8 Bytes

Pure function (no exec pins).

Converts a string to its UTF-8 byte representation.

Inputs

Text String

Outputs

Return Value Array of Byte

UTF-8 Bytes to String

Pure function (no exec pins).

Converts UTF-8 bytes back into a string.

Inputs

Bytes Array of Byte

Outputs

Return Value String

Make Index Array

Pure function (no exec pins).

Builds the integer sequence 0..Count-1 in one allocation. Handy test/driver input for the parallel array nodes.

Inputs

Count Integer

Outputs

Return Value Array of Integer

Make Float Range

Pure function (no exec pins).

Builds a float sequence starting at Start, advancing by Step, with Count elements - in one allocation.

Inputs

Start Float

Step Float

Count Integer

Outputs

Return Value Array of Float

Get Memory Stats

Pure function (no exec pins).

Returns the process' physical memory usage: current, peak and available system memory, in megabytes.

Outputs

Used MB Float

Peak Used MB Float

Available MB Float

Estimate Array Memory (KB)

Pure function (no exec pins).

Rough memory footprint of an array: Element Count x Element Size Bytes, in kilobytes. Floats/ints are 4 bytes, vectors 24, strings vary.

Inputs

Element Count Integer

Element Size Bytes Integer

Outputs

Return Value Float

Image Resize (Async)

Async node. Result arrives on the Completed pin.

Loads an image file (PNG, JPEG or BMP), resizes it with bilinear filtering and saves it, all on a worker thread. Quality applies to JPEG output (1-100).

Inputs

Source Path String

Destination Path String

New Width Integer

New Height Integer

Output Format Enum (default PNG)

Options: PNG, JPEG

Quality Integer (default 90)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String

Elapsed Ms Float

Pixel Processing (Async)

Async node. Result arrives on the Completed pin.

Loads an image file, applies a pixel operation (grayscale, invert, brightness, contrast, tint) and saves the result, all on a worker thread.

Inputs

Source Path String

Destination Path String

Operation Enum

Options: Grayscale, Invert, Brightness, Contrast, Tint

Amount Float (default 1)

Tint Color Linear Color (default (1, 1, 1, 1))

Output Format Enum (default PNG)

Options: PNG, JPEG

Quality Integer (default 90)

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

File Path String

Elapsed Ms Float

Screenshot Save (Async)

Async node. Result arrives on the Completed pin.

Requests an engine screenshot and fires Completed once the PNG exists on disk. Relative filenames are written to Saved/Screenshots. Show UI includes the HUD and widgets.

Inputs

Filename String (default "ParallelFlowScreenshot")

Show UI Boolean (default true)

Outputs (on Completed)

File Path String

Elapsed Ms Float

String Processing (Stats)

Async node. Result arrives on the Completed pin.

Counts characters, words and lines of a text on a worker thread. Handles multi-megabyte strings without hitching.

Inputs

Text String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Characters Integer

Words Integer

Lines Integer

Elapsed Ms Float

String Processing (Find & Replace)

Async node. Result arrives on the Completed pin.

Replaces every occurrence of Find with Replace in a text on a worker thread and reports how many replacements were made.

Inputs

Text String
Find String
Replace String
Case Sensitive Boolean (default true)
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String
Occurrences Replaced Integer
Elapsed Ms Float

Encrypt String (Async)

Async node. Result arrives on the Completed pin.

Encrypts a string with AES-256 (password-derived key) on a worker thread and returns Base64 ciphertext. Suitable for save-file obfuscation; see the docs for the security model.

Inputs

Text String
Password String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String
Elapsed Ms Float

Decrypt String (Async)

Async node. Result arrives on the Completed pin.

Decrypts Base64 ciphertext produced by Encrypt String on a worker thread. A wrong password or tampered data fires Failed instead of returning garbage.

Inputs

Base64 Ciphertext String
Password String
Priority Enum (default Normal)
Options: Low, Normal, High, Critical

Outputs (on Completed)

Result String
Elapsed Ms Float

Encrypt Bytes (Async)

Async node. Result arrives on the Completed pin.

Encrypts a byte array with AES-256 (password-derived key) on a worker thread. Pair with Async Save (Bytes) for encrypted save files.

Inputs

Bytes Array of Byte

Password String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Bytes Array of Byte

Elapsed Ms Float

Decrypt Bytes (Async)

Async node. Result arrives on the Completed pin.

Decrypts a byte array produced by Encrypt Bytes on a worker thread. A wrong password or tampered data fires Failed instead of returning garbage.

Inputs

Bytes Array of Byte

Password String

Priority Enum (default Normal)

Options: Low, Normal, High, Critical

Outputs (on Completed)

Bytes Array of Byte

Elapsed Ms Float

12.7 Debug

Blueprint category: ParallelFlow|Debug

Read the scheduler's live statistics and task lists from Blueprint (the same data the ParallelFlow Debugger window shows).

Get Scheduler Stats

Pure function (no exec pins).

Returns aggregate ParallelFlow statistics: running/queued/finished counts, average and last execution times, worker thread count, an occupancy estimate and process memory usage.

Outputs

Return Value Scheduler Stats

Get Active Tasks

Pure function (no exec pins).

Returns a snapshot of every queued and running ParallelFlow task: name, state, progress, timings and thread.

Outputs

Return Value Array of Task Debug Info

Get Task History

Pure function (no exec pins).

Returns snapshots of recently finished tasks, newest first, including execution times and error messages.

Outputs

Return Value Array of Task Debug Info

Clear Task History

Function (runs immediately).

Clears the finished-task history shown in the debug window. Counters are unaffected.

Log Scheduler State

Function (runs immediately).

Writes the scheduler statistics and every active task to the output log. Also available as the ParallelFlow.DumpTasks console command.

12.8 Advisor

Blueprint category: ParallelFlow|Advisor

Ask whether an optimization is worth it and get a plain-language answer.

Advise (Performance Advisor)

Pure function (no exec pins).

Describe what your Blueprint is doing and how much data is involved. The Advisor tells you - in plain language - whether ParallelFlow will help, which node to use, and the expected gain. It will honestly say 'keep using Blueprint' when your data is small enough that optimizing is not worth it.

Inputs

Operation Enum

Options: Sort an Array, Filter an Array, Find in an Array, Reduce an Array (sum/avg/min/max), Transform every Element (Map), Distance to many Points, Nearest Actor / Point, Actors within a Radius, Sort an Array of Structs, Visibility / Line-of-Sight (many), Procedural Placement (scatter + snap to ground), Heatmap / Influence Map, Noise Field, Heavy ForEach Loop, File Read / Write, Hashing / Checksum, Compression

Element Count Integer

Runs Every Frame Boolean (default false)

Outputs

Return Value Advice

Advice To Text

Pure function (no exec pins).

Formats an advice result as a readable multi-line string for printing on screen or to the log.

Inputs

Advice Advice

Outputs

Return Value String

Log Advice

Function (runs immediately).

Prints an advice result to the output log.

Inputs

Advice Advice

12.9 Benchmark

Blueprint category: ParallelFlow|Benchmark

Measure the old way against the ParallelFlow way on your own hardware.

Benchmark Sort

Function (runs immediately).

Generates Count random floats, then times a single-threaded sort against ParallelFlow's parallel sort. Returns the measured comparison. Runs synchronously - use it in a benchmark map, not in gameplay.

Inputs

Count Integer (default 500000)

Seed Integer (default 12345)

Outputs

Return Value Benchmark Result

Benchmark Reduce (Sum)

Function (runs immediately).

Times a single-threaded sum loop against ParallelFlow's parallel reduce over Count floats. Returns the measured comparison.

Inputs

Count Integer (default 1000000)

Seed Integer (default 777)

Outputs

Return Value Benchmark Result

Benchmark Nearest Point

Function (runs immediately).

Times the classic 'find the closest point' loop against ParallelFlow's parallel version over Count points. This is the 'closest enemy' pattern every game has. Returns the measured comparison.

Inputs

Count Integer (default 200000)

Seed Integer (default 4242)

Outputs

Return Value Benchmark Result

Benchmark To Text

Pure function (no exec pins).

Formats a benchmark result as a readable multi-line string.

Inputs

Result Benchmark Result

Outputs

Return Value String

12.10 Tasks

Blueprint category: ParallelFlow|Tasks

Cancel

Function (runs immediately).

Request cancellation. The task stops at its next safe point and the Cancelled pin fires. Threads are never force-terminated.

(c) Black Scar Studio 2026. All Rights Reserved.