ParallelFlow

The Blueprint Performance Toolkit

Stay in Blueprint. Stop the lag.

The missing performance layer for Blueprint developers.
No threads. No async. No C++.

User Manual - Version 2.0
For Unreal Engine 5.2 | (c) Black Scar Studio 2026

Contents

Introduction

Welcome to **ParallelFlow** - the missing performance layer for Blueprint.

When your Blueprint starts to lag, you are usually told *"rewrite it in C++."* ParallelFlow is the alternative. Replace a few slow Blueprint nodes with optimized native equivalents and your game runs smooth again - no threads, no async, no C++.

The mission

If a Blueprint developer asks *"how do I optimize this?"*, the answer should almost always be *"ParallelFlow has a node for that."*

You never think about threads, workers, or schedulers. You think:

My Blueprint is lagging. -> I replace a few nodes. -> My game is smooth.

How to read this manual

– New here? Read **The Blueprint Optimization Guide** first - it teaches you how

to think about Blueprint performance, then **Quick Start** to make your first optimization in five minutes.

– Have a specific game system to speed up? Jump to **The Blueprint Performance

Cookbook** and find your recipe.

– Want the details of a node? See the **API Reference**.

Three things to remember

1. **The Advisor is your friend.** Not sure whether to optimize? Drop in the

Advise node with your element count - it gives an honest answer, including "keep using normal Blueprint" when your data is small.

1. **Measure, don't guess.** The Benchmark nodes and Benchmark Map prove the

gain on your own hardware.

1. **Small data stays simple.** ParallelFlow is for when things get big.

ParallelFlow - The Blueprint Optimization Guide

This chapter teaches you *how to think* about Blueprint performance. Read it once and you'll know when to reach for ParallelFlow - and when not to.

Why Blueprint slows down

Blueprint is not slow because it's "interpreted." It's slow in specific, predictable situations:

1. **Big loops.** A `ForEach` over 100,000 items runs every node in the loop body 100,000 times, one after another, on the **Game Thread**. Nothing else - rendering, input, the rest of your game - happens until that loop finishes.
2. **Nested loops.** A loop inside a loop is multiplication: 1,000 x 1,000 = 1,000,000 iterations. This is where "why did my game freeze for a second?" comes from.
3. **Per-frame heavy work.** A moderately expensive loop is fine once. On **Tick**, every frame, it becomes a permanent frame-rate tax.
4. **Blocking operations.** Saving a big file, parsing a large data table, hashing - these stop the world while they run.

None of these are Blueprint's fault, exactly. They'd be slow in single-threaded C++ too. The difference is that a C++ programmer knows to move them off the Game Thread or across CPU cores. **ParallelFlow does that for you.**

Understanding the Game Thread

Your game has one **Game Thread** that runs gameplay, Blueprint, and ticks - and it has to finish all of that in ~16 ms to hit 60 FPS. Meanwhile your CPU has 8, 12, 16+ cores mostly sitting idle.

– A Blueprint loop uses **one** core (the Game Thread) and blocks everything else on it.

– A ParallelFlow node uses the **idle** cores, and hands the result back to the Game Thread when done - so your frame never stalls.

That's the whole trick. You don't need to understand threads to benefit from them; you just need to stop doing heavy work on the one thread that can't afford it.

What ParallelFlow actually does

– Runs the heavy part (the sort, the search, the distance math) as **native C++ across multiple cores**.

– Keeps your **Blueprint logic on the Game Thread**, where it's safe - you get the results back in a normal Blueprint event.

– Chooses worker count, chunk size, and small-vs-large strategy **automatically**. There is nothing to configure.

What ParallelFlow does NOT do

– It does **not** run your Blueprint graph on other threads. That would crash any engine - the Blueprint VM isn't thread-safe. ParallelFlow runs *native operations* in parallel and returns to Blueprint for the logic.

– It does **not** claim to beat well-written custom C++. It gives you production-quality native code so you don't have to write that C++ yourself.

– It does **not** help small data. Sorting 50 items is already instant; parallelizing it is pure overhead. The Advisor will tell you so.

The optimization workflow

1. **Measure, don't guess.** Use the profiler (`stat unit`, `stat game`) or just notice the hitch. Find the loop that's actually slow.
2. **Ask the Advisor.** Drop in **Advise (Performance Advisor)** with the operation and element count. It gives you an honest call.
3. **Swap the node.** Replace the Blueprint pattern with the ParallelFlow node from the Blueprint Replacement Guide.
4. **Prove it.** Run the matching **Benchmark** node, or open the Benchmark Map, to see the real gain on your hardware.
5. **Move on.** Don't optimize what isn't slow. ParallelFlow is a scalpel, not a coat of paint.

Rules of thumb

– **Under a few thousand elements:** keep normal Blueprint.

– **Tens of thousands and up:** ParallelFlow starts to clearly win.

– **Hundreds of thousands, or anything on Tick:** you should almost certainly be using ParallelFlow.

– **Inherently heavy work** (heatmaps, noise, file I/O, many line traces): use ParallelFlow regardless of count - Blueprint has no good answer for these.

The mindset shift

Before ParallelFlow, the answer to "my Blueprint is too slow" was "rewrite it in C++." That's a wall a lot of Blueprint developers hit and never get past.

ParallelFlow moves that wall much further out. You stay in Blueprint - designing, iterating, shipping - and reach for a ParallelFlow node the moment something gets heavy. You'll write custom C++ later, and less of it, and only where it truly matters.

ParallelFlow - Installation Guide

Requirements

– Unreal Engine **5.2**

– Windows, Mac, Linux, Android or iOS target platforms (runtime module)

– No third-party dependencies - ParallelFlow uses only engine Runtime modules (`Core`, `CoreUObject`, `Engine`, `Json`, `ImageWrapper`)

Installing from Fab

1. Purchase / claim ParallelFlow on Fab.
2. In the **Epic Games Launcher**, open *Unreal Engine Library*, find ParallelFlow under *Fab Library* and click **Install to Engine** for 5.2.
3. Open your project, go to **Edit Plugins**, search for *ParallelFlow*, tick **Enabled** and restart the editor.

Installing into a single project (source)

1. Close the editor.
2. Copy the `ParallelFlow` folder into your project's `Plugins` directory (create it if needed):

```
YourProject/
Plugins/
ParallelFlow/
ParallelFlow.uplugin
Source/...
```

1. Right-click your `.uproject` **Generate Visual Studio project files** (C++ projects), or just open the project - the editor offers to compile the plugin.
2. Confirm the plugin is enabled under **Edit Plugins Performance ParallelFlow**.

Verifying the installation

– In any Blueprint, right-click and search for **"Parallel Sort"** - you should see the ParallelFlow nodes.

– Open **Window Developer Tools ParallelFlow Debugger** - the task monitor should appear.

– In the console (\`), run ParallelFlow.DumpTasks` - the scheduler logs its state.

Packaging

ParallelFlow's runtime module is packaged automatically with your game. The editor module (debug window) is editor-only and never ships. No additional packaging settings are required.

Uninstalling

Disable the plugin in **Edit Plugins** or delete the `Plugins/ParallelFlow` folder. Blueprints using ParallelFlow nodes will report missing nodes afterwards, so remove those nodes first if you plan to keep the Blueprints.

ParallelFlow - Quick Start Guide

Five minutes from install to your first background task.

1. Your first background task

In any Blueprint (e.g. the Level Blueprint):

1. **Event BeginPlay** -> **Run Background Task** (Iterations = `50,000,000`).
2. Drag off **Started** -> **Print String** ("Task started...").
3. Drag off **Progress** -> **Print String** (append the `Progress` float).
4. Drag off **Completed** -> **Print String** (append `Elapsed Ms`).

Press Play. The workload takes real CPU time, the progress prints stream in, and your frame rate never dips - the work runs on a background worker thread.

2. Sorting half a million floats

1. **Generate Random Floats (Async)** - Count = `500000`, Min = `0`, Max = `1000`.
2. From its **Completed** (Results) -> **Parallel Sort (Float)**.
3. From the sort's **Completed** -> print `Elapsed Ms` and `Results` length.

Compare with a Blueprint for-loop sort of the same data - ParallelFlow finishes in milliseconds on worker threads while a Blueprint loop would freeze the game for minutes.

3. Progress bars

Every node's **Progress** pin delivers 0.0-1.0 on the Game Thread, throttled to ~30 updates/second:

- Create a UMG widget with a **Progress Bar**.
- Bind the bar's *Percent* to a float variable.
- In the graph, set that variable from the node's **Progress** event.

4. Cancellation

Two equivalent options:

- Store the **Task Handle** from **Started**, then call **Cancel Task** (ParallelFlow Async Flow) whenever you want - e.g. from a UI button.
- Or drag from the node's **Task** output pin and call **Cancel** on it directly.

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

5. Task groups (Wait For All)

1. Start three tasks (e.g. three **Run Background Task** nodes) and collect their **Task Handle**s into an array.
2. Feed the array into **Wait For All Tasks**.
3. Its **Completed** fires once every task reached a terminal state - perfect for "loading" phases.

Wait For Any Task completes as soon as the first one finishes instead.

6. Spreading Blueprint work across frames

Blueprint bodies can only run on the Game Thread, so for per-element *Blueprint* logic use the frame-spreading loops:

- **Async Loop** - First Index / Last Index / Iterations Per Frame.
- **Async For Each (Indices)** - wire *Loop Body Index* into your array **Get**.
- **Async Sequence** - one step per frame for staged init.
- **Async Repeat / Async While** - interval-driven loops; end them with **Break Loop** (fires Completed) or **Cancel** (fires Cancelled).

Rule of thumb: *built-in heavy math -> Parallel Arrays nodes (worker threads). Custom Blueprint logic per element -> Async Loop (frame-spread).*

7. Async files

- **Async Save (Text)** / **Async Load (Text)** - save-game text, logs, configs.
- **Async JSON Read / Write** - validated JSON with pretty printing.
- **CSV Import** - parse huge spreadsheets into rows without a hitch.
- **Compress File** + **Hash File** - archive and fingerprint files in the background.

Relative paths resolve against your project directory.

8. Watch everything live

Open **Window Developer Tools ParallelFlow Debugger**:

- **Active Tasks**: state, priority, live progress, execution time, worker thread.
- **Finished Tasks**: newest first, with timings and error messages.
- Stat tiles: running/queued counts, average/last execution time, worker count, CPU estimate, process memory.
- **Cancel All Tasks** and **Clear History** buttons.

At runtime (including packaged builds) use the Blueprint nodes **Get Scheduler Stats**, **Get Active Tasks**, **Get Task History** - or the console commands `ParallelFlow.DumpTasks` and `ParallelFlow.CancelAll`.

Where next?

- `BlueprintExamples.md` - copy-ready recipes (heatmaps, voronoi, encrypted saves, image thumbnails...)
- `API.md` - every node, pin and struct in detail.
- `FAQ.md` / `KnownLimitations.md` - the fine print.

ParallelFlow - Blueprint Replacement Guide

> **The mission:** if a Blueprint developer asks *"how do I optimize this?"*, the answer should almost always be *"ParallelFlow has a node for that."*

This is the lookup table. Find the slow Blueprint pattern you have on the left; use the ParallelFlow node on the right. The Performance Advisor node will confirm whether it's worth it for *your* element count.

Your slow Blueprint pattern	Replace with	Typical gain*
Sort node / manual sort on a big array	**Parallel Sort** (Float/Int/Vector/String)	~2-5x
ForEach + Branch + Add to build a filtered list	**Parallel Filter**	~3-6x
ForEach + Branch scanning until you find an item	**Parallel Find / Parallel Search**	~3-8x
ForEach accumulating a running total / min / max	**Parallel Reduce**	~4-8x
ForEach that rewrites/scales every element	**Parallel Map**	~3-6x
ForEach + Distance for every point	**Distance Calculator**	~3-6x
ForEach tracking the smallest distance ("closest enemy")	**Closest Actor / Nearest Point**	~3-6x
ForEach + Line Trace for many targets	**Visibility Processing**	huge (async traces)
Nested loops splatting points into a grid	**Heatmap Generator**	impractical in BP otherwise
Nested loops sampling noise per cell	**Noise Generator**	impractical in BP otherwise
A **ForEach** that hitches the frame because it does too much at once	**Async Loop** (spreads it across frames)	no hitch
Save/Load Game that freezes on big saves	**Async Save / Load**	removes the freeze
Parsing a big data table / spreadsheet	**CSV Import** / **Async JSON Read**	off the Game Thread
Building a save checksum or verifying a download	**Hash File / Checksum**	streamed, off-thread
Shrinking screenshots / textures on disk	**Image Resize**	off the Game Thread

\Gains are rough and scale with CPU cores and data size. ParallelFlow never claims to beat well-written custom C++ - it gives you production-quality native code without writing any.*

How to use this table

1. Notice a Blueprint that lags, hitches, or shows up hot in the profiler.
2. Find its pattern above.
3. Drop in the **Advise (Performance Advisor)** node with your operation + element count - it tells you if the swap is worth it at your scale.
4. If yes, replace the pattern with the ParallelFlow node. Wire **Completed** to what used to come after your loop.

5. Optionally run the matching **Benchmark** node to see the exact gain on your hardware.

The honest part

ParallelFlow will tell you **not** to bother when your data is small. Sorting 20 items? The Advisor says "keep using Blueprint" - the setup cost of going parallel would outweigh the microseconds you'd save. The plugin optimizes you *and* teaches you where the real bottlenecks are.

The Blueprint Performance Cookbook

Real game systems, real bottlenecks, real fixes. Each recipe follows the same shape so you can scan for your problem:

> **Problem** · **The slow Blueprint** · **The ParallelFlow fix** · **Why it's faster** · **Expected gain** · **Common mistakes** · **When NOT to use it**

A rule that runs through every recipe: **small data -> keep normal Blueprint.** When you're unsure, drop in the **Advise** node with your element count and it will tell you honestly. ParallelFlow is a scalpel, not a coat of paint.

> Screenshots referenced as *[shot: ...]* are captured in-editor from the example maps; see `MarketplacePresentation.md` for the shot list.

---

1. Leaderboard / Scoreboard

Problem. End of match: sort 50,000 player scores for a ranked board. The Blueprint Sort (or a manual loop) runs on the Game Thread and the screen freezes for a beat right when players want their result.

The slow Blueprint. `Get All Scores -> Sort (descending)` on the Game Thread, or worse a manual bubble/insertion sort with nested loops.

The ParallelFlow fix. `Parallel Sort (Float)` (or Int), *Ascending* = *false* -> **Completed** -> build the board UI. Wire **Progress** to a spinner.

Why it's faster. A parallel merge sort spreads the work across CPU cores instead of one, and it runs off the Game Thread so the frame never stalls.

Expected gain. ~2-5x on the sort itself, and - more importantly - **zero hitch** on the frame that shows results.

Common mistakes. Sorting the full player *struct* array by copying it repeatedly. Instead sort a parallel array of scores (or score+index) and reorder once at the end.

When NOT to use it. A 12-player lobby. 12 items sort in microseconds; use normal Blueprint.

---

2. Inventory & Loot

Problem. A survival/ARPG inventory with thousands of items needs sorting by value/weight/rarity, and filtering ("show only weapons in range of my level").

The slow Blueprint. `ForEach` over the inventory with a `Branch` per item building a new filtered array, then a `Sort`. Two passes on the Game Thread.

The ParallelFlow fix. Sort with `Parallel Sort`; filter with `Parallel Filter (Int/Float)` on the numeric key (value, weight, item level). For "items within a stat range", `Parallel Filter` with Min/Max.

Why it's faster. Filtering and sorting are the two things Blueprint loops do slowest at scale, and both are natively parallelized here.

Expected gain. ~3-6x on filter, ~2-5x on sort; no frame hitch when opening the bag.

Common mistakes. Re-filtering every frame while the inventory UI is open. Filter once on open / on change, cache the result.

When NOT to use it. A 30-slot inventory. Normal Blueprint is instant.

---

3. Crafting - "can I craft this?" across a big recipe book

Problem. Thousands of recipes; on every inventory change you re-check which recipes are now craftable.

The slow Blueprint. Nested loops: for each recipe, for each ingredient, search the inventory. That's recipes x ingredients x inventory - a classic frame-killer.

The ParallelFlow fix. Precompute an ingredient-count map once. Then use `Parallel Count (Int)` / `Parallel Filter` to evaluate the recipe list in one native pass. For "do I have N of item X", `Parallel Count` over the inventory ids.

Why it's faster. You collapse nested Blueprint loops into native parallel passes and kill the multiplication.

Expected gain. From "noticeable hitch on pickup" to imperceptible; often 5x+ because you also removed the nesting.

Common mistakes. Recomputing the whole craftable set on *every* item pickup. Debounce: recompute at end of frame, not per item.

When NOT to use it. A dozen recipes. Keep it in Blueprint.

---

4. Quest System - objective checks at scale

Problem. Open-world game with hundreds of active/inactive quests, each with conditions evaluated against world state.

The slow Blueprint. A `ForEach` over all quests on Tick evaluating conditions.

The ParallelFlow fix. Don't evaluate on Tick. Use `Async Loop` (spread across frames) to walk the quest list a few per frame, or `Parallel Filter` to find "quests whose numeric trigger is met" in one pass, then evaluate only those in Blueprint.

Why it's faster. You stop doing all the work in one frame; the heavy scan is native and the Blueprint logic only touches the handful that matter.

Expected gain. Eliminates the per-frame quest tax entirely.

Common mistakes. Evaluating string-heavy conditions in the parallel pass. Reduce to numeric triggers for the scan; do the rich logic on the survivors.

When NOT to use it. A linear game with 5 quests.

---

5. Save Game - big saves without the freeze

Problem. Saving a large world/inventory/stats blob freezes the game for a fraction of a second (or more on console/HDD).

The slow Blueprint. `Save Game to Slot` / building a huge string and writing it - all blocking the Game Thread.

The ParallelFlow fix. Serialize to a string/bytes, then `Async Save (Text/Bytes)`. Want it small and safe? `Compress (Bytes)` -> `Async Save`. Want it protected? `Encrypt String` -> `Async Save`. Verify with `Hash File`.

Why it's faster. Disk writes and compression happen on a worker thread; the frame keeps rendering. It's not "faster I/O", it's "no freeze".

Expected gain. Removes the save-hitch entirely; compression also shrinks files 40-70%.

Common mistakes. Kicking off a save every few seconds and overlapping writes. Use one **Task Queue** named "SaveQueue" so saves serialize safely.

When NOT to use it. Tiny option files (a few KB). The stock nodes are fine.

---

6. Minimap & Fog of War

Problem. Splatting hundreds/thousands of unit or POI positions into a minimap texture / influence grid every frame.

The slow Blueprint. Nested loops writing into a 2D array on the Game Thread. Unshippable past small sizes.

The ParallelFlow fix. `Heatmap Generator` for influence/threat maps; `Grid Processing` for sampling positions; feed the result into a dynamic texture.

Why it's faster. Per-cell work is exactly what Blueprint is worst at and native parallel code is best at.

Expected gain. From "impossible in Blueprint" to "runs every frame if you want".

Common mistakes. Regenerating a 512x512 map every frame when the data barely changed. Regenerate on a timer or on change.

When NOT to use it. A 16x16 static minimap computed once.

---

7. AI Manager - "closest enemy / target"

Problem. Each of many AI agents finds its nearest target among hundreds/thousands of actors. This is the single most common Blueprint performance killer.

The slow Blueprint. For each agent, `Get All Actors` + `ForEach` tracking the smallest `Distance`. If every agent does it every frame, it's agents x targets, every frame.

The ParallelFlow fix. `Closest Actor` (snapshots positions on the Game Thread, searches on workers). For many-vs-many, build the position array once and run `Nearest Point` / `Distance Calculator` per query.

Why it's faster. The distance scan is native and parallel, and it's off the Game Thread.

Expected gain. ~3-6x per query; and crucially it doesn't stack onto the frame.

Common mistakes. Calling `Get All Actors Of Class` every frame (that itself is slow). Cache the target list; refresh it periodically, not per frame.

When NOT to use it. 5 enemies. A simple loop is fine and has no setup cost.

---

8. Crowd AI / Boids / Simulation

Problem. Thousands of agents each reading neighbours for flocking, avoidance, or influence.

The slow Blueprint. O(n²) neighbour loops in Blueprint. Dies at a few hundred agents.

The ParallelFlow fix. Use `Parallel Filter (Vector)` (radius ring around each query point) and `Distance Calculator` to get neighbour sets natively; do the steering math in Blueprint on the reduced set. For density, `Heatmap Generator`.

Why it's faster. The n² part becomes native parallel work; Blueprint only handles the small per-agent decision.

Expected gain. Lets you scale from hundreds to thousands of agents.

Common mistakes. Trying to run the *steering* in parallel - that's Blueprint logic and stays on the Game Thread. Parallelize the *queries*, not the decisions.

When NOT to use it. A handful of agents.

---

9. RTS Unit Selection & Commands

Problem. Box-select across a large army; issue formation moves; sort units by distance to target.

The slow Blueprint. `ForEach` over all units testing bounds, then another pass to sort by distance.

The ParallelFlow fix. `Parallel Filter (Vector)` for "units inside the selection volume", `Parallel Sort (Vector)` (by distance/axis) for formation ordering, `Bounding Box` for the group's extents.

Why it's faster. Selection and ordering are native parallel passes instead of Game-Thread loops.

Expected gain. ~3-6x; smooth selection of large armies.

Common mistakes. Re-filtering during the drag every frame at full army size. Filter on mouse-up, or throttle during drag.

When NOT to use it. Selecting 10 units.

---

10. Tower Defense - targeting many towers

Problem. Dozens/hundreds of towers each pick a target (closest / strongest / furthest-along-path) among many creeps every frame.

The slow Blueprint. Per tower, per creep distance loops. Multiplies fast.

The ParallelFlow fix. Build the creep position array once per frame. Each tower query uses `Nearest Point`; "furthest along path" uses `Path Distance` precomputed per creep, then a `Parallel Reduce (Max)`.

Why it's faster. One shared native structure feeds all towers instead of each tower re-scanning in Blueprint.

Expected gain. Scales tower + creep counts far past the Blueprint ceiling.

Common mistakes. Rebuilding the creep array separately for every tower. Build it once, share it.

When NOT to use it. A few towers and a dozen creeps.

---

11. Projectile Manager - many bullets

Problem. Hundreds/thousands of projectiles doing distance/overlap/visibility checks.

The slow Blueprint. Per-projectile loops and line traces on the Game Thread.

The ParallelFlow fix. `Distance Calculator` for proximity culling; `Visibility Processing` for batched line-of-sight (async traces, no frame block).

Why it's faster. Batched native queries and async traces replace serial Blueprint traces.

Expected gain. Large; async visibility especially removes trace-stall.

Common mistakes. Running full collision through this - ParallelFlow does the *queries*, not physics resolution.

When NOT to use it. A few projectiles.

---

12. Damage / AoE System

Problem. An explosion needs "all actors within radius, sorted by distance, with falloff".

The slow Blueprint. `Get All Actors` + distance `ForEach` + sort.

The ParallelFlow fix. `Parallel Filter (Vector)` (radius) -> `Distance Calculator` (for falloff) -> apply damage in Blueprint on the filtered set.

Why it's faster. The find + distance pass is native and parallel; Blueprint only touches actors actually hit.

Expected gain. ~3-6x on big AoE; no hitch on the explosion frame.

Common mistakes. Using this for a 3-target cleave. Overkill.

When NOT to use it. Small radius, few actors.

---

13. Large Data Tables / CSV-driven content

Problem. Importing or scanning a huge external data table (items, dialogue, balance data).

The slow Blueprint. Parsing strings in a `ForEach`, or a blocking file read that freezes on load.

The ParallelFlow fix. `CSV Import (Async)` for external spreadsheets; `Async JSON Read` for JSON; `Parallel Search (String)` to find rows.

Why it's faster. Parsing and disk I/O move off the Game Thread; search is native.

Expected gain. Removes load-screen freezes; searches large tables without hitch.

Common mistakes. Re-importing every time you need one row. Import once, keep the rows.

When NOT to use it. A 20-row DataTable already in the project - use the built-in DataTable.

---

14. NPC Manager / Open World streaming logic

Problem. Deciding which of thousands of NPCs to activate/sleep based on distance to the player each frame.

The slow Blueprint. Distance `ForEach` over every NPC on Tick.

The ParallelFlow fix. `Distance Calculator` (all NPC positions vs player) -> `Parallel Filter` (within activation radius) -> activate only those in Blueprint. Spread activation with `Async Loop`.

Why it's faster. The distance sweep is native/parallel; activation is amortized across frames.

Expected gain. Turns a per-frame tax into a background sweep; big open-world win.

Common mistakes. Toggling NPC state every frame causing thrash. Add hysteresis (activate at R, deactivate at R+margin).

When NOT to use it. A small level with a handful of NPCs.

---

15. Procedural Generation & Voxel Terrain

Problem. Generating heightfields, biome masks, or voxel density fields - inherently per-cell, huge counts.

The slow Blueprint. Nested loops sampling noise per cell. Freezes for seconds, or is simply infeasible.

The ParallelFlow fix. `Noise Generator` (multi-octave Perlin, seeded), `Voronoi Processing` (regions/biomes), `Grid Processing` (sample points). Feed results to your mesh/voxel builder.

Why it's faster. Per-cell math is the textbook parallel workload; native code across cores instead of one Blueprint loop.

Expected gain. From "impossible in Blueprint" to "generate a chunk without a visible stall".

Common mistakes. Generating on the Game Thread at load and blocking. Generate ahead / in background and stream in.

When NOT to use it. A tiny fixed grid computed once at design time.

---

16. Pathfinding Helpers

Problem. Not full navmesh - the helper math around it: path length, distance-along-path, nearest waypoint among many.

The slow Blueprint. Loops summing segment lengths; nearest-waypoint distance loops.

The ParallelFlow fix. `Path Distance` (total + cumulative length per point - perfect for "how far along am I"), `Nearest Point` (closest waypoint), `Distance Calculator` (waypoint costs).

Why it's faster. Native batch math replaces Blueprint segment loops.

Expected gain. ~3-6x on large path/waypoint sets.

Common mistakes. Expecting actual A* navigation - this is the surrounding math, not the search.

When NOT to use it. Short paths, few waypoints.

---

17. Chat / Log / Text Processing

Problem. Filtering, searching, or transforming large chat logs or text buffers (profanity filter, search, formatting).

The slow Blueprint. Per-line `ForEach` with string ops.

The ParallelFlow fix. `Parallel Filter (String)` (contains/startswith/length), `Parallel Search (String)`, `Find & Replace`, `String Processing (Stats)`.

Why it's faster. String scanning across many lines is native and parallel.

Expected gain. ~3-8x on large logs.

Common mistakes. Running per keystroke on a huge buffer. Debounce input.

When NOT to use it. A short chat with a few lines.

---

18. Analytics / Stats Aggregation

Problem. Summing/averaging large arrays of gameplay stats (damage dealt, distances travelled, economy totals).

The slow Blueprint. `ForEach` accumulating a running total.

The ParallelFlow fix. `Combine Array` (Parallel Reduce) - Sum/Average/Min/Max, in double precision so big totals stay accurate.

Why it's faster. Chunked parallel reduction across cores; double precision avoids float drift on large sums.

Expected gain. ~4-8x, plus better numerical accuracy than a naive float loop.

Common mistakes. Summing thousands of floats into a `float` accumulator in Blueprint and getting drift. The native reduce uses double internally.

When NOT to use it. Summing 50 numbers.

Using this cookbook

1. Find the system closest to yours.
2. Confirm scale with the **Advise** node (it will say "keep Blueprint" if you're small).
3. Swap the pattern for the ParallelFlow node; wire **Completed** to what came after your loop.
4. Prove it with the matching **Benchmark** node or the Benchmark Map.

Every recipe shares one honest truth: **ParallelFlow removes the bottleneck, not the need to think.** Measure first, optimize the thing that's actually slow, and let small data stay simple.

ParallelFlow - Blueprint Examples

Copy-ready graph recipes. Node names are shown in **bold**; pins in *italics*.

---

1. Non-blocking leaderboard sort

Sort 200k scores while the game keeps running:

```
Event (scores ready)
-> **Parallel Sort (Float)** (Values = Scores, Ascending = false)
*Started* -> save Task Handle (for a cancel button)
*Progress* -> update UMG progress bar
*Completed* -> *Results* = sorted scores -> rebuild leaderboard UI
*Failed* -> **Print String** (Error Message)
```

2. Radius query on a huge point cloud

Find every pickup within 30 m of the player, preserving order:

```
**Parallel Filter (Vector)**
Values = PickupLocations (TArray<Vector>)
Center = Player location
Min Distance = 0
Max Distance = 3000
*Completed* -> spawn markers for *Results*
```

3. Closest enemy of 5,000 actors

```
**Get All Actors Of Class** (Enemy)
-> **Closest Actor (Async)** (Actors = enemies, Target = player location)
*Completed* -> *Actor* is the nearest enemy (null only if it died mid-task; *Index* stays
valid)
```

Positions are snapshotted on the Game Thread when the node runs; the search happens on workers.

4. Procedural heatmap of player deaths

```
**Heatmap Generator (Async)**
Points = DeathLocations
Bounds Min = (-10000, -10000) Bounds Max = (10000, 10000)
Width = 128 Height = 128 Radius = 500 Normalize = true
*Completed* -> *Values* (row-major, index = Y * Width + X)
-> drive a dynamic material / debug tiles
```

5. Seeded terrain mask

```
**Noise Generator (Async)** (Width = 256, Height = 256, Scale = 0.03, Octaves = 5, Seed =
1337)
*Completed* -> *Values* in 0..1 -> threshold at 0.6 -> spawn foliage on **Async Loop**
(spread across frames)
```

6. Encrypted save file (background, zero hitches)

Save:

```
SaveData (String)
-> **Encrypt String (Async)** (Password = "...")
*Completed* -> *Result* (Base64)
-> **Async Save (Text)** (FilePath = "Saved/SaveGames/slot1.pfsave")
*Completed* -> show "Saved" toast
```

Load:

```
**Async Load (Text)** ("Saved/SaveGames/slot1.pfsave")
*Completed* -> **Decrypt String (Async)** (same password)
*Completed* -> parse SaveData
*Failed* -> wrong password / corrupt file - show an error, never garbage
```

7. CSV-driven content

```
**CSV Import (Async)** (FilePath = "Content/Data/items.csv", First Row Is Header = true)
*Completed* -> *Headers* = column names
-> *Rows* = array of Cells
-> **Async For Each (Indices)** (Array Length = Row Count, Iterations Per Frame = 50)
*Loop Body* -> Get row by *Index* -> spawn/register item
*Completed* -> done, UI unlocked the whole time
```

8. Task group with a loading screen

```
Start:
**Run Background Task** (A) - *Started* -> add handle to TaskHandles
**CSV Import (Async)** (B) - *Started* -> add handle to TaskHandles
**Async JSON Read** (C) - *Started* -> add handle to TaskHandles
-> **Wait For All Tasks** (TaskHandles)
*Progress* -> loading bar (fraction of tasks finished)
*Completed* -> remove loading screen
```

9. Serialized saves with Task Queue

Multiple systems can request saves at any time; they must never overlap:

```
Anywhere in the project:
**Task Queue** (Queue Name = "SaveQueue")
*Execute* -> perform this system's save logic
*Completed* -> queue slot released; the next request runs
```

10. Thumbnail generation

```
**Folder Scan (Async)** (Directory = "Saved/Photos", Pattern = "*.png")
*Completed* -> **Async For Each (Indices)** over *Results*
*Loop Body* -> **Image Resize (Async)**
Source = Results[Index], Destination = Results[Index] + "_thumb.png"
New Width = 256, New Height = 144, Format = JPEG, Quality = 85
```

11. Performance comparison rig (for your own benchmarks)

```
**Generate Random Floats (Async)** (Count = 1,000,000)
*Completed* ->
1) **Get Game Time In Seconds** -> run a Blueprint ForEach sum -> time it (freezes!)
2) **Parallel Reduce (Float)** (Sum) -> *Elapsed Ms* (game keeps running)
-> print both timings side by side
```

12. Line-of-sight batch check

```
**Visibility Processing (Async)**
Origin = sniper tower location
Target Points = 300 patrol points
Trace Channel = Visibility
Ignore Actor = tower actor
*Completed* -> *Visible* (bool per point) + *Num Visible*
```

Uses the engine's asynchronous physics traces - no Game Thread blocking.

Recipe conventions

- Always wire **Failed** at least to a **Print String** - ParallelFlow errors are descriptive and tell you exactly what to fix.
- Store **Task Handle**s if the player can cancel (level exit, back button).
- For arrays above ~100k elements prefer the **Parallel Arrays** nodes over any Blueprint loop.

ParallelFlow - API Documentation

All types live in the **ParallelFlow** runtime module. Blueprint categories mirror this document.

Core types (`ParallelFlowTypes.h`)

`EParallelFlowTaskPriority`

`Low`, `Normal`, `High`, `Critical`. Maps to Unreal task priorities at or below normal - background work never starves the Game Thread. `Low` runs when workers are idle; use `Critical` sparingly for latency-critical results.

`EParallelFlowTaskState`

`Queued -> Running -> Completed | Failed | Cancelled`. Exactly one terminal state per task.

`FParallelFlowTaskHandle`

Copyable id delivered by every node's **Started** event. Used by: `Cancel Task`, `Is Task Running`, `Get Task Progress`, `Wait For All/Any Tasks`.

`FParallelFlowError`

`Message` (human-readable cause) + `Source` (node name). Delivered on every **Failed** pin.

`FParallelFlowTaskDebugInfo`

Snapshot for debugging: handle, name, state, priority, progress, queue time (ms), execution time (ms), thread name, error message.

`FParallelFlowSchedulerStats`

Running/queued/completed/failed/cancelled counts, total launched, average & last execution ms, worker thread count, estimated CPU %, process memory used/peak MB.

`FParallelFlowCsvRow`

`Cells : Array<String>` - one CSV row (Blueprint arrays cannot nest).

The async node contract

Every async node inherits `UParallelFlowAsyncAction` and exposes:

Pin	Payload	Notes
Started	Task Handle	Fired synchronously when the node activates
Progress	float 0-1	Throttled to ~30/s; final 1.0 guaranteed before Completed
Completed	typed results + Elapsed Ms	Exactly once on success
Failed	`FParallelFlowError`	Exactly once on error; never silent
Cancelled	-	Exactly once if cancelled

Pin	Payload	Notes
Task (object pin)	the action	`Cancel()`, `Get Task Handle()`, `Is Task Running()`

All events fire on the Game Thread.

Async Flow (`ParallelFlowFlowActions.h`)

Node	Inputs	Completed payload
Run Async	-	Elapsed Ms (continues next frame)
Async Delay (Frames)	Frame Count	Elapsed Ms
Async Delay	Duration (real seconds)	Elapsed Ms
Run Background Task	Iterations (int64), Priority	Checksum, Elapsed Ms - deterministic CPU workload
Async Loop	First/Last Index, Iterations Per Frame	Iterations, Elapsed Ms; body pin `Loop Body(Index)`
Async Do N	N, Iterations Per Frame	as Async Loop
Async For Each (Indices)	Array Length, Iterations Per Frame	as Async Loop
Async Repeat	Count (0 = forever), Interval Seconds	as Async Loop
Async While	Interval Seconds	runs until `Break Loop` / `Cancel`
Async Sequence	Step Count	one step per frame
Parallel Split	-	Branch A-D fire on consecutive frames, then Completed
Wait For All Tasks	Task Handles	Finished Tasks, Elapsed Ms
Wait For Any Task	Task Handles	first finished handle
Task Queue	Queue Name	`Execute` fires in FIFO turn; Completed after release

`Break Loop` (on loop nodes) ends gracefully via Completed; `Cancel` ends via Cancelled.

Parallel Arrays (`ParallelFlowArrayActions.h`)

Element types: **Float, Integer, Vector, String** (nodes suffixed accordingly).

Operation	Notes
Parallel Sort	Parallel merge sort. Float/Int: ascending/descending. Vector: by Length/X/Y/Z. String: lexicographic, optional case sensitivity
Parallel Filter	Float/Int: `[Min, Max]` range. Vector: distance ring around Center. String: Contains / NotContains / StartsWith / EndsWith / LengthBetween. Order-preserving

Operation	Notes
Parallel Map	Built-in transforms - Float: Abs/Negate/Sqrt/Square/Scale/Offset/Clamp; Int: adds Modulo; Vector: Normalize/Scale/Offset/Abs/GridSnap; String: Upper/Lower/Trim/Reverse
Parallel Reduce	Float -> float (double-precision accumulation), Int -> int64, Vector -> vector (Sum/Average/Min/Max; vector Average = centroid). Min/Max of an empty array -> Failed
Parallel Find / Search	First matching index or 1. Float/Vector accept a tolerance; Search (String) = first *containing* substring
Parallel Count	Range count (Float/Int) or substring count (String)
Parallel Unique	Float: tolerance-based, result sorted. Int/String/Vector: order-preserving (Vector supports tolerance grid)
Parallel Shuffle	Seeded Fisher-Yates - deterministic
Parallel Reverse / Merge	Merge optionally sorts the result
Generate Random Floats / Integers / Points In Box	Seeded, background generation

Chunking (library, synchronous): `Get Chunk Count`, `Get Array Chunk (Float/Integer/Vector/String)`.

Math Processing (`ParallelFlowMathActions.h`)

Node	Completed payload
Distance Calculator	float[] distances (parallel to input points)
Nearest Point	Index, Point, Distance
Closest Actor	Actor (resolved on GT; null if destroyed mid-task), Index, Distance
Grid Processing	Vector[] row-major grid (optional seeded jitter)
Heatmap Generator	float[] WxH density field (smoothstep falloff, optional normalize)
Noise Generator	float[] WxH multi-octave Perlin, 0-1, seeded
Voronoi Processing	int[] WxH cell -> nearest-seed index
Path Distance	Total Length + cumulative length per point
Visibility Processing	bool[] per target + visible count (async physics traces)
Bounding Box	Center, Extents, Min, Max

File Processing (`ParallelFlowFileActions.h`)

Node	Notes
Async Save (Text/Bytes)	UTF-8 text (optional append) or raw bytes; creates directories
Async Load (Text/Bytes)	Missing file -> Failed with path

Node	Notes
Async JSON Read	Validates; returns pretty JSON + top-level keys; parser errors -> Failed
Async JSON Write	Validates before writing; pretty or compact
CSV Import / Export	RFC 4180 (quotes, escapes, embedded newlines); custom delimiter; optional header row
File Copy	1 MB chunks, progress, cancel deletes the partial file
File Move / Delete	Descriptive failures (missing source, locked file...)
Folder Scan	Wildcard pattern, optional recursion, files and/or directories
Compress / Decompress (Bytes/File)	Zlib / Gzip / LZ4 / Oodle; self-describing container (decompress auto-detects the codec); reports original size + ratio
Hash File / Bytes / String	MD5 / SHA-1 / CRC32; files stream in 1 MB chunks; lowercase hex digest
Checksum File	CRC32 shorthand

Relative paths resolve against the project directory.

Utility (`ParallelFlowUtilityActions.h`)

Node	Notes
Image Resize	PNG/JPEG/BMP in -> PNG/JPEG out; bilinear; JPEG quality 1-100
Pixel Processing	Grayscale / Invert / Brightness / Contrast / Tint
Screenshot Save	Requests an engine screenshot, completes when the PNG exists (10 s timeout)
String Processing (Stats)	Characters / words / lines of multi-MB text
String Processing (Find & Replace)	Returns result + replacement count
Encrypt / Decrypt (String/Bytes)	AES-256, password-derived key (stretched SHA-1 x10,000), PKCS#7, integrity prefix - wrong password -> Failed, never garbage. String output is Base64

Function library (`ParallelFlowLibrary.h`)

Task control: `Cancel Task`, `Cancel All Tasks`, `Is Task Running`, `Get Task Progress`, `Is Valid Task Handle`. Debug: `Get Scheduler Stats`, `Get Active Tasks`, `Get Task History`, `Clear Task History`, `Log Scheduler State`. Conversion: `Bytes Base64`, `Bytes Hex`, `String UTF-8 Bytes`. Arrays: `Get Chunk Count`, `Get Array Chunk (x4)`, `Make Index Array`, `Make Float Range`. Memory: `Get Memory Stats`, `Estimate Array Memory (KB)`.

Console commands

- `ParallelFlow.DumpTasks` - log stats + every active task.

– `ParallelFlow.CancelAll` - request cancellation of everything.

Extending in C++

Subclass `UParallelFlowAsyncAction` (or `UParallelFlowTickedAction` for Game-Thread-driven nodes):

1. Declare your five delegates (`Started`, `Progress`, your typed `Completed`, `Failed`, `Cancelled`) and override the four `Broadcast*` one-liners.
2. Add a static factory with `meta = (BlueprintInternalUseOnly = "true", WorldContext = "WorldContextObject")` that calls `Setup(...)`.
3. Override `ExecuteAction()` and hand a lambda to `LaunchBackgroundWork(...)`. Inside it, use `WorkerHandleCancellation`, `WorkerReportProgress`, `WorkerFail` and finish with `WorkerComplete<YourClass>(...)`. Capture plain data + `WeakThis` only - never UObjects.

The internal parallel algorithms (`ParallelFlowParallelAlgo.h`) are reusable for custom workloads.

ParallelFlow - FAQ

Q: Does ParallelFlow really run my Blueprint code on other threads? No - and nothing safely can. The Blueprint VM is not thread-safe, so ParallelFlow never executes Blueprint graphs off the Game Thread. Instead it gives you two tools: *built-in* heavy operations (sorting, filtering, hashing, image processing, spatial math...) that run as thread-safe C++ on worker threads, and *frame-spreading* loops (Async Loop and friends) for your own per-element Blueprint logic. That combination covers the vast majority of "my Blueprint freezes the game" cases without ever risking a crash.

Q: Which thread do the Completed / Failed / Cancelled events fire on? Always the Game Thread. You can safely spawn actors, update widgets and touch any Blueprint object in those events.

Q: What happens if I stop PIE / open a new level while tasks are running? Tasks are asked to cancel, worker threads finish their current step and exit, and pending callbacks detect that their node object is gone and do nothing. No crashes, no dangling references. The scheduler statistics still record the tasks as cancelled.

Q: Is there a task limit? No hard limit. Tasks queue through Unreal's task system, which schedules them across the available worker threads. Launching thousands of tiny tasks works but is wasteful - prefer one Parallel Arrays node over a thousand single-element tasks.

Q: Why is my task's Progress pin not firing every frame? Progress dispatch is throttled to ~30 updates per second per task so massive workloads can't flood the Game Thread. The final 1.0 update is always delivered before Completed.

Q: Do the priorities starve my game? No. Low/Normal/High map to Unreal's *background* task priorities; even Critical maps to normal task priority, never above. Rendering and gameplay always win.

Q: Can I cancel a task from anywhere? Yes. Keep the Task Handle from the Started event and call **Cancel Task** from any Blueprint, or **Cancel All Tasks** for everything (also available as the `ParallelFlow.CancelAll` console command and a button in the debug window).

Q: Cancelling doesn't stop my task instantly. Why? Cancellation is cooperative - the worker polls the cancel flag at safe points (every few thousand elements, between file chunks, between sort phases). This is what makes it crash-proof. Worst case latency is a few milliseconds.

Q: Does the debug window work in packaged games? The window itself is editor-only. In packaged builds use the Blueprint debug nodes (**Get Scheduler Stats**, **Get Active Tasks**, **Get Task History**) to build your own overlay, or the console commands.

Q: Are file paths relative to something? Yes - relative paths resolve against your **project directory** (`FPaths::ProjectDir()`). Absolute paths are used as-is. Screenshot filenames without a path go to `Saved/Screenshots`.

Q: How secure is the encryption? Encrypt/Decrypt uses AES-256 with a key derived from your password (two stretched SHA-1 chains, 10,000 iterations each) in ECB block mode with PKCS#7 padding, plus an integrity prefix so wrong passwords fail cleanly. That is strong protection for save files and local data against casual tampering. It is **not** a substitute for a vetted cryptography suite in adversarial scenarios (no IV/nonce, no HMAC) - see Known Limitations.

Q: Parallel Unique (Float) changed my array order! Documented behavior: tolerance-based float dedupe sorts the array first (that is what makes it fast at scale). Integer/String/Vector Unique preserve first-occurrence order.

Q: Can two Task Queue nodes with different queue names run in the same frame? Yes. Queues are independent; ordering is only guaranteed *within* a queue name.

Q: Does it work in shipping builds / all platforms? Yes. The runtime module uses only engine Runtime modules and is allowed on Win64, Mac, Linux, Android and iOS. The editor module (debug window) never ships.

Q: UE 5.3+? This release targets UE 5.2 exactly. Newer engine versions get their own releases.

ParallelFlow - Known Limitations

Honest fine print. Every limitation below is a deliberate trade-off in favor of engine stability.

Blueprint execution model

- **Blueprint bodies never run on worker threads.** The Blueprint VM is not thread-safe, so nodes with Blueprint body pins (Async Loop, Async For Each, Task Queue, Parallel Split, sequences) spread execution across *frames* on the Game Thread. True multi-threading is available only through the built-in C++ operations (Parallel Arrays, Math, Files, Utility).
- **Parallel Map / Filter / Reduce use built-in operations**, not arbitrary Blueprint lambdas, for the same reason. If you need a custom per-element transform on workers, that transform must be C++ - the plugin's action framework (`UParallelFlowAsyncAction`) is designed to be subclassed for exactly this.
- **Parallel Split** fires its branches on consecutive frames, not simultaneously - Blueprint execution is inherently sequential on the Game Thread.

Data model

- Array inputs are **copied once** when a node runs (Blueprint cannot safely share arrays with a worker thread). One copy in, moved through the pipeline, one result array out. For a 1M-float array that is 4 MB per direction.
- Blueprint arrays are limited to `int32` element counts and single dimensions; CSV rows wrap their cells in a struct because Blueprint cannot nest arrays.
- **Parallel Unique (Float)** returns a *sorted* array (tolerance-based dedupe requires sorting). Integer/String/Vector variants preserve order.
- **Parallel Reduce (Integer)** `Average` truncates toward zero; `Product` can overflow int64 for large inputs.
- The compression container and encrypted payloads are little-endian. All platforms UE 5.2 ships games on are little-endian, so this only matters if you parse the files with external big-endian tooling.

Timing / scheduling

- Cancellation is **cooperative**: workers stop at their next check point (typically within milliseconds), never instantly. `std::inplace_merge` phases of a running sort finish their current merge before checking.
- Ticked nodes (Async Delay, loops) run on the core ticker: they measure **real time** and keep running while the game is paused. This is intentional (loading screens); use gameplay timers if you need pause-aware delays.
- Task execution order across *different* priorities is up to Unreal's task scheduler; only Task Queue nodes guarantee strict ordering (within one queue name).

Files & platform

- File nodes operate on **loose files**, not assets in .pak files. Reading packaged game content requires the asset registry / pak APIs, which are outside this plugin's scope. Writing to `Saved/` works on every supported platform.
- Folder Scan progress is coarse (directory sizes are unknown up front).

- **Screenshot Save** depends on an active game viewport; it fails with a descriptive error after 10 seconds if the renderer never produces the file (e.g. headless/dedicated server).
- **Image nodes** support PNG, JPEG and BMP input; output is PNG or JPEG.

Encryption

- AES-256 in **ECB block mode** (Unreal's built-in FAES), PKCS#7 padding, stretched-SHA1 key derivation, magic-prefix integrity check. Strong enough for save-file protection and casual tampering; **not** designed for adversarial network security: there is no IV/nonce (identical plaintexts produce identical ciphertexts) and no authenticated encryption (HMAC). If you need bank-grade crypto, use a dedicated library.

Visibility Processing

- Uses the engine's async physics traces, so results arrive after the physics system processes them (typically the next frame). Requires a valid world; it is not available in pure editor-utility contexts.

Debug window

- Editor-only (by design - it keeps shipping builds lean). Runtime introspection is available through the Blueprint debug nodes and console commands.
- Finished-task history is capped at 512 entries; older entries drop off.

Engine version

- Built and tested against **Unreal Engine 5.2 only**.

ParallelFlow - Example Project Walkthrough

This guide builds the demo level that ships with the example project (and doubles as a construction manual if you are wiring it yourself). Each station is one Blueprint actor demonstrating a subsystem. All graphs use only ParallelFlow nodes plus stock UMG - build time is roughly 30-45 minutes for the full map.

> Blueprint assets (.uasset) must be authored in the editor. Every graph below is specified pin-by-pin; the recipes in `BlueprintExamples.md` cover the same ground in shorter form.

Level layout

One map, `PFDemo_Main`, with seven demo stations on podiums plus a wall-mounted stats screen. A shared UMG HUD (`WBP_PFDemo`) shows: an FPS counter (top-left, proving the game never hitches), a progress bar, a status text block and Cancel buttons.

Station 1 - Async array processing

`BP_Station_Arrays`:

1. On interact: **Generate Random Floats (Async)** - Count `500000`, Seed `42`.
2. *Completed* -> **Parallel Sort (Float)** -> *Completed* -> status text:

`Sorted 500,000 floats in {Elapsed Ms} ms.`

1. Chain **Parallel Filter (Float)** (Min `250`, Max `750`) and

Parallel Reduce (Float) (Average) off the sorted result and append their timings.

1. Wire every node's *Progress* into the shared HUD progress bar and every

Failed into the status text.

Station 2 - Parallel sorting race (performance comparison)

`BP_Station_Race` demonstrates why ParallelFlow exists:

1. **Generate Random Floats (Async)** - Count `200000`.
2. Path A (the wrong way): a Blueprint ForEach min-search over the array -

watch the FPS counter die while it runs (cap the count if needed).

1. Path B: **Parallel Sort (Float)** - the FPS counter never moves.
2. Print both durations side by side on the podium's TextRender.

Station 3 - Background calculations (math)

`BP_Station_Math`:

1. **Grid Processing (Async)** - 100x100 grid, Cell Size `120`, Jitter `40` ->

Completed -> spawn instanced static mesh markers via **Async For Each (Indices)** (Iterations Per Frame `200`) so even the spawning is spread out.

1. **Noise Generator (Async)** - 128x128, Seed exposed on the actor -> color the

markers by noise value.

1. **Closest Actor (Async)** every second (via **Async Repeat**, Interval `1.0`)

against all markers -> highlight the marker nearest the player.

Station 4 - Task groups

`BP_Station_Groups`:

1. One button starts three **Run Background Task** nodes with different

iteration counts and priorities (Low / Normal / High).

1. Collect the three handles from *Started* into an array ->

Wait For All Tasks -> *Progress* drives a segmented "3 of 3" display, *Completed* fires a celebration particle.

1. A second button runs the same trio through **Wait For Any Task** to show

race semantics.

Station 5 - Progress bars & cancellation

`BP_Station_Cancel`:

1. **Run Background Task** - Iterations `2000000000` (long-running on purpose).
2. *Progress* -> HUD bar. A big red **CANCEL** button calls **Cancel Task** with

the stored handle.

1. *Cancelled* -> status text `Cancelled at {bar percent}%` - demonstrating that

the Cancelled pin (not Completed) fires and the worker stopped cooperatively.

Station 6 - Files & encryption

`BP_Station_Files`:

1. Serialize a small struct of player stats to a JSON string ->

Async JSON Write to `Saved/Demo/stats.json`.

1. **Encrypt String (Async)** the same JSON (password field on the podium) ->

Async Save (Text) to `Saved/Demo/stats.pfenc` -> **Hash File (Async)** (MD5) -> print the digest.

1. Load path: **Async Load (Text)** -> **Decrypt String (Async)** -> parse ->

populate the podium display. Enter a wrong password to see the descriptive *Failed* path (never garbage output).

1. **CSV Import (Async)** on the bundled `DemoData/items.csv` (5,000 rows) ->

spawn one pickup per row via **Async For Each**.

Station 7 - Large data & the debugger

`BP_Station_Debug`:

1. A "STRESS" button launches: two **Parallel Sort** nodes (1M elements each,

Low priority), a **Heatmap Generator** (512x512), a **Voronoi Processing** (256x256, 64 seeds) and a **Compress (Bytes)** of 50 MB of generated data.

1. Open **Window Developer Tools ParallelFlow Debugger** and watch the

scheduler juggle them: states, worker threads, progress, average times, CPU estimate, memory.

1. The podium screen mirrors a subset at runtime using **Get Scheduler Stats**

and **Get Active Tasks** on an **Async Repeat** (Interval `0.25`).

HUD (`WBP_PFDemo`)

– FPS counter: `1 / Get World Delta Seconds`, updated on Tick - the whole demo's

point is that this number never dips while stations run.

– Progress bar + status text bound to Blueprint-exposed variables that every

station writes through a Blueprint Interface (`BPI_PFDemoHud`).

Packaging the demo

The demo uses only runtime nodes, so it packages as-is. The Debugger window is editor-only; Station 7's runtime panel demonstrates the shipping-safe alternative.