Great! I'll write a ~1000-word technical blog post in the tone of The Delphi Geek, exploring the `ComputeCore` unit from its design goals to implementation intricacies. I'll include example code and assume the reader is familiar with the Parallel Programming Library (PPL), without using any visual elements. I'll let you know as soon as it's ready for review.

ComputeCore: A Simple Parallel Task Framework

ComputeCore is a lightweight parallel processing framework for Delphi that lets you easily run CPU-intensive tasks across multiple threads. Its **goals** are straightforward: to run (possibly compute-heavy) tasks concurrently, allow tasks to spawn child tasks, and expose a very simple interface for the caller. For example, you can submit an anonymous method (`TProc`) and get back an `ITask` to wait on. Behind the scenes ComputeCore manages a pool of worker threads to process these tasks. By default it creates `CPUCount-1` worker threads (leaving one core free for the main thread), but you can also construct your own instance with a custom thread count. Importantly, tasks have **no return value and no cancellation** support – they simply run to completion and you wait on them if needed. This design choice keeps the framework minimal and focused.

Goals

- **Parallel task execution.** Run many tasks concurrently to utilize multiple CPU cores, speeding up computational workloads.
- **Nested task support.** Any task may itself call `Run(...)` to schedule subtasks, allowing parallel algorithms to spawn subtasks (e.g. a parallel quicksort).
- **Simple interface.** Only two methods: one to run a task and one to wait for it. There's no complex configuration or result-passing – just fire-and-forget tasks.

These goals influence the design heavily. For example, because tasks have no return value, there's no need to store results or futures inside the framework. And because there is no built-in cancellation, the implementation can avoid the complexity of aborting tasks, focusing instead on just execution and completion.

Implementation Choices

ComputeCore keeps the implementation minimal:

- **No return value:** Tasks are procedures (`TProc`) that return nothing. If you need a result, your task can write to a shared variable or object that you manage externally.
- **No cancellation:** Once started, a task runs to completion. This avoids dealing with cancellation tokens or cooperative aborting. (If you need to stop long-running tasks, you'd have to handle that inside the task code itself.)

These choices make the core simpler. You just queue a `TProc` and let it run. There's no need to manage futures or handle cancellation flags in the library itself.

Top-Level Design

At the top level, ComputeCore uses a **singleton** pattern. There is one global compute core, accessible via the interface `IComputeCore`. By default, the first time you call `Run(...)`, ComputeCore automatically creates a global instance using `(CPUCount - 1)` worker threads. If you want a different setup (for example, you only have 2 cores but still want 2 worker threads), you can override this by explicitly assigning:

```
GlobalComputeCore := TComputeCore.Create(MyThreadCount);
```

Once set, all `Run` and `WaitFor` calls go through that global instance. The interface `IComputeCore` is fully thread-safe, meaning you can call `Run` from any thread (even from within a task) without additional locking.

ComputeCore's public API is extremely simple:

```
function Run(const taskProc: TProc): ITask; overload;
procedure WaitFor(const task: ITask);
```

Underneath, these calls just dispatch to `GlobalComputeCore`. For compatibility with Delphi's Parallel Programming Library (PPL), ComputeCore also provides `ITask` and a class `TTask` with static methods. For example, `TTask.Run(procedure begin ... end)` internally ensures the global compute core is created (an "optimistic initialization" technique) and then calls `IComputeCore.Run`. This mirrors the Delphi PPL, which similarly offers `TTask.Run` for asynchronous procedures.

Because the framework relies on this global instance, you *could* create multiple compute cores (by using `TComputeCore.Create` multiple times with different thread counts), but doing so would mean the convenience of `TTask.Run` and `TTask.WaitFor` (which assume a single global core) would no longer apply. Typically you just let the singleton instance serve the whole application.

In creating the global core, ComputeCore uses **optimistic lazy initialization**. That is, it doesn't lock before creating the instance; it simply creates it and tries to store it in the global variable atomically. This is a common concurrency pattern described in the OmniThreadLibrary documentation and TheDelphiGeek blog. Essentially: if two threads try to initialize simultaneously, they may both create an instance but only one will "win" and the other will be discarded. This avoids a locking bottleneck on startup while still ensuring exactly one instance ends up in use.

Data Structures in the Compute Core

Internally, `TComputeCore` manages its threads and tasks using a few basic data structures:

- **Worker thread list (`FThreads`).** An array (fixed at creation time) of thread objects. Each one runs the worker loop (see below). This array is only accessed during construction and destruction of the core, so it doesn't require additional locking.
- **Inactive thread stack (`FInactiveThreads`).** An array (used as a stack) of indices of threads that are currently idle and waiting. When a thread finishes a task and has no more work immediately, it pushes itself onto this list. When a new task arrives, if there's an idle thread index available here, the core pops it and signals that thread to wake up.

- **Task queue (`FTasks`).** A first-in-first-out queue of pending tasks (each task is encapsulated in an object). When you call `Run`, the `TProc` is wrapped in a task object and enqueued here.

Because multiple threads (the main thread and any of the worker threads) can be calling `Run`, `WaitFor`, or finishing tasks concurrently, **access to `FTasks` and `FInactiveThreads` must be synchronized**. ComputeCore uses Delphi's `TMonitor` (via `MonitorEnter`/`MonitorExit`) on the compute core object (`Self`) to protect these shared structures. In other words, most methods grab the core's monitor, do a quick operation on the queue or array, and release it. Delphi's monitor can be thought of as a ready-to-use critical section tied to any `TObject`.

In practice, methods that acquire the lock end in `_U` (unsafe) if they assume the lock is already held, or without `_U` if they do the locking themselves. For example, `GetTask_U` assumes the monitor is held and just pops a task from the queue. `AllocateTask` is a higher-level method that holds the lock while calling `GetTask_U` and also updates the thread status (active/inactive) inside the same lock. This "big lock" strategy ensures there are no race conditions between pulling a task and updating the idle list. (If these were done in two separate locks, you could schedule a task right after a thread checks the queue but before it marks itself active, leading to a task that no thread picks up.)

In short: whenever the queue or the inactive-thread list is accessed, it's protected by `MonitorEnter(Self)` / `MonitorExit(Self)`. This keeps the code correct at the cost of briefly blocking other threads, but because each operation is quick, contention stays low.

Running a Task

When you call `GlobalComputeCore.Run(MyProc)`, ComputeCore does something like:

1. **Enqueue the task.** The `TProc` is wrapped in an `ITask` object and added to `FTasks`.
2. **Wake a worker if idle.** If `FInactiveThreads` is not empty, pop an idle thread index and signal that thread's event (using something like `TEvent.SetEvent`) so it wakes up immediately to grab tasks.
3. **Return the ITask.** You get back an `ITask` interface you can later pass to `WaitFor`.

If no thread was idle (because all workers were busy), the task just sits in the queue. As soon as any worker thread finishes its current task, it will loop and notice there's a new task, so it will start working on it.

In code form, the core of the worker loop looks like this (simplified):

```
while (not Terminated) and (FSignal.WaitFor <> wrTimeout) do
begin
  // A signal arrived (new task or termination)
  while (not Terminated) and FOwner_ref.AllocateTask(Self, task) do
    task.Execute;
end;
```

Here, `FSignal` is a synchronization event that a main thread pulses whenever a new task is enqueued or a thread should check for work. `AllocateTask` (called inside the inner loop) does something like: take the core's lock, get a task from `FTasks`, mark this thread active (or

put it on the idle list if no task was found), then release the lock. If a task was returned, the thread exits the lock and calls `task.Execute` (which runs your `TProc`). It then loops to try `AllocateTask` again, draining as many queued tasks as it can before going back to wait on `FSignal`.

Worker Threads

Each worker is a simple Delphi `TThread` running code similar to above. In more detail, a worker thread does roughly:

```
repeat
  // Wait until signaled or timeout
  if (not Terminated) and (FSignal.WaitFor <> wrTimeout) then
    // Process tasks until no more are available
    while (not Terminated) and FOwner_ref.AllocateTask(Self, task) do
      task.Execute;
until Terminated;
```

- **Waiting:** When there are no tasks, the thread waits on an event (`FSignal.WaitFor`). This puts it to sleep efficiently.
- **Signaling:** When `Run` enqueues a task, it calls `FSignal.SetEvent`, waking exactly one waiting thread. That thread grabs tasks and goes to work.
- **Draining:** Once awake, the thread keeps calling `AllocateTask` in a loop, processing all available tasks in the queue one by one. Only when the queue is empty does it go back to waiting.

This simple loop means threads stay active as long as work is available, and otherwise don't spin or consume CPU.

Race Conditions and Thread Safety

In any multithreaded queue, careful attention to **race conditions** is crucial. For example, consider what happens if one task is enqueued at the exact same time another thread finishes a task. The core must ensure that the newly enqueued task isn't "lost" (i.e. enqueued but no thread ever wakes up to handle it).

ComputeCore avoids such races by doing multiple related actions under one lock. For instance, the `AllocateTask` method (inside the lock) both takes a task from the queue (`GetTask_U`) and updates the thread's state (marks it active or pushes it on `FInactiveThreads`) before releasing the lock. This atomic update prevents the scenario where a thread might check the queue, see nothing, and go inactive, while another thread enqueues a task in the tiny gap. By bundling those steps, either the task is seen by an already-active thread or an idle thread is immediately woken.

Likewise, signaling new tasks to workers is done only after enqueuing under the same lock that protects `FTasks`. This coordination ensures that any new task always has a chance to trigger exactly one thread. Because all shared state changes (queue and idle list) happen within protected critical sections, there's no chance for a "blind spot" where a task is sitting in the queue but all threads are sleeping.

Preventing Deadlocks with Nested Tasks

A classical problem in thread pools is **deadlock by saturation**: suppose every worker thread is busy running a task, and those tasks do something like `TTask.Run` and then immediately `WaitFor` the child tasks. If tasks spawn a new task and then wait, it's possible for *all* worker threads to be tied up waiting on subtasks, with no thread left to actually execute those subtasks – a deadlock. This is exactly what can happen with a naive parallel QuickSort implementation if not handled carefully.

ComputeCore solves this by letting waiting threads "help out." Instead of truly blocking when you call `WaitFor(task)`, the waiting thread will loop and execute other tasks from the queue while it's waiting for `task` to finish. In pseudo-code, `WaitFor` looks like:

```
while task.WorkDone.WaitFor(0) = wrTimeout do
begin
  Acquire;              // lock
  GetTask_U(newTask);   // try to get another task from queue
  Release;              // unlock
  if not Assigned(newTask) then
    TThread.Yield       // no work, yield CPU momentarily
  else
  begin
    newTask.Execute;    // run the task ourselves
    if Assigned(newTask.ExceptObj) then
      raise Exception.Create(newTask.ExceptObj);
  end;
end;
```

In words, while the `WorkDone` event of our awaited task is not yet signaled, the thread grabs the core lock, takes another pending task if available, and executes it. If there's no other task, it just yields briefly (allowing other threads to run). This way, threads waiting on child tasks will process unrelated tasks. If the queued tasks happen to be exactly the subtasks it's waiting for, this effectively keeps the pipeline full and avoids deadlock. In practice this means a depth-`N` nested chain of tasks can execute without deadlock as long as each waiting thread helps execute work. (This is sometimes called a *helping* or *work-stealing* strategy, though here it's a simple single-queue version.)

With this approach, even parallel QuickSort won't hang as long as there is at least one "helping" check in the wait loop. In fact, this "wait by executing pending tasks" trick is a common pattern in custom thread pools to avoid starvation in nested parallel workloads.

Conclusion

ComputeCore shows that you can implement a custom parallel task pool in Delphi with relatively little code if you follow a few key principles:

- **Keep it simple.** A minimal interface (`Run`/`WaitFor`) and simple data structures make the code understandable and maintainable.
- **Minimize sharing.** Only the task queue and idle-thread list are shared, and only for short critical sections. Most of the work (executing tasks) happens outside the lock.

- **Think about races.** Every insertion into the queue or change to the idle list is done under a lock to prevent missed signals. Combining actions (e.g. pulling a task and marking thread state) into one locked block prevents subtle bugs.
- **Prevent deadlock.** In nested-task scenarios, have idle threads help finish work. The "help while waiting" loop in `WaitFor` is crucial to avoid exhausting the pool.
- **Test thoroughly.** Concurrent code is tricky. Even with careful design, always write unit tests and stress tests to exercise scheduling, nested waits, and edge cases.

By following these ideas – and studying patterns like optimistic initialization and object monitors – you can build your own lightweight parallel framework for Delphi. It's not that hard, and the result (like ComputeCore) can give you fine-grained control over task scheduling without the overhead of a heavier library.