Building Your Own Parallel Processing Framework: A Deep Dive into ComputeCore

Parallel processing doesn't have to be rocket science. Sometimes, the most elegant solutions are the simplest ones. Today, I want to walk you through a lightweight parallel processing framework called ComputeCore that proves this point beautifully.

The Goals: Keep It Simple, Stupid

When designing ComputeCore, the objectives were refreshingly straightforward:

- **Run computationally intensive tasks** across multiple threads
- **Allow tasks to spawn other tasks** (crucial for divide-and-conquer algorithms like QuickSort)
- **Provide a simple interface** that doesn't require a PhD in computer science to understand

Notice what's missing from this list? Return values and cancellation support. These are deliberate omissions that keep the implementation lean and focused. Sometimes, knowing what *not* to include is just as important as knowing what to include.

The Big Picture: Singleton Simplicity

At its heart, ComputeCore revolves around a single global instance:

```
var
  GlobalComputeCore: IComputeCore;
```

This singleton is automatically created on first use with `CPUCount-1` worker threads (leaving one core free for the main thread and other system processes). Want more control? No problem—just create it manually:

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

The public interface couldn't be simpler:

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

For those familiar with Delphi's Parallel Programming Library (PPL), ComputeCore provides compatible `ITask` and `TTask` interfaces. The `TTask.Run` method even creates the global compute core on-the-fly if it doesn't exist yet, using optimistic initialization patterns.

Under the Hood: Data Structures That Matter

ComputeCore's internal architecture is elegantly minimal:

- **FThreads**: A preallocated array owning the worker `TThread` objects
- **FInactiveThreads**: Another preallocated array serving as a stack of idle threads
- **FTasks**: A queue of tasks waiting to be executed

The beauty lies in the simplicity. No complex lock-free data structures, no elaborate thread pools—just straightforward arrays and queues doing exactly what they need to do.

Thread Safety: The Monitor Approach

In multithreaded programming, protecting shared data is non-negotiable. ComputeCore takes a conservative but effective approach using Monitor-based synchronization:

```
MonitorEnter(Self);
try
  // Protected operations here
finally
  MonitorExit(Self);
end;
```

The code maintains a clear visual separation between "safe" methods (that acquire exclusive access internally) and "unsafe" methods (that expect exclusive access to already be established). The latter are suffixed with "_U", making it immediately obvious which methods require external synchronization.

This approach minimizes the scope of shared access. The `FThreads` array, for instance, is only accessed from the constructor and destructor, eliminating the need for synchronization entirely.

Task Execution: Wake Up and Work

When you call `Run()`, here's what happens:

1. The task gets queued in `FTasks`
2. If an inactive thread is available, it's removed from `FInactiveThreads` and signaled to start work
3. If no threads are available, the task simply waits in the queue

Each worker thread runs this beautifully simple loop:

```
while (not Terminated) and (FSignal.WaitFor <> wrTimeout) do begin
  while (not Terminated) and FOwner_ref.AllocateTask(Self, task) do
    task.Execute;
end;
```

When there's no work, threads wait for their signal. When work arrives, they process tasks until the queue is empty, then return to waiting.

Race Conditions: The Devil in the Details

Here's where things get interesting. Consider this scenario: Thread A finishes a task and goes idle just as Thread B queues a new task. Without careful synchronization, that new task might sit in the queue indefinitely while all threads think they're idle.

ComputeCore solves this by making `AllocateTask` atomic. From within a single critical section, it:

1. Attempts to get a task with `GetTask_U`
2. Marks the worker thread as active or inactive with `MarkThreadActive_U`

This prevents the race condition that would occur if these operations were split across separate critical sections. It's a perfect example of why understanding concurrency is crucial—the obvious solution often isn't the correct one.

The Deadlock Problem: Work Stealing to the Rescue

Here's a scenario that will make any parallel programming enthusiast break out in a cold sweat: recursive task spawning that exceeds the number of available worker threads. Picture a QuickSort implementation where each partition spawns two sub-tasks, and those spawn more sub-tasks, and so on.

With only `CPUCount-1` worker threads, you'll quickly exhaust your thread pool. All threads end up waiting for child tasks that can never execute because no threads are available to run them. Classic deadlock.

ComputeCore's solution is elegant: work stealing in the `WaitFor` method. While waiting for a task to complete, the calling thread doesn't just sit idle—it actively pulls tasks from the queue and executes them:

```
while task.WorkDone.WaitFor(0) = wrTimeout do begin
  __Acquire;
  GetTask_U(newTask);
  __Release;
  if not assigned(newTask) then
    TThread.Yield
  else begin
    newTask.Execute;
    if assigned(newTask.ExceptObj) then
      raise Exception(newTask.ExceptObj);
  end;
end;
```

This prevents resource exhaustion by ensuring that waiting threads remain productive. It's the kind of solution that seems obvious in hindsight but requires deep understanding of the problem domain to implement correctly.

Lessons Learned: The Four Pillars of Parallel Programming

Building ComputeCore teaches us several fundamental principles for parallel programming:

Keep it simple! Complexity is the enemy of correctness. ComputeCore's power comes from its simplicity, not from trying to solve every conceivable parallel programming problem.

Minimize access to shared data structures! The less data you share between threads, the fewer synchronization points you need. ComputeCore's design carefully limits shared state to only what's absolutely necessary.

Think about race conditions! They're not just theoretical concerns—they're real bugs waiting to happen. Every shared data access point needs careful analysis of what happens when multiple threads interact with it simultaneously.

Do lots of testing! Concurrency bugs are notoriously difficult to reproduce and debug. Comprehensive testing isn't optional—it's survival.

Final Thoughts

ComputeCore proves that you don't need a massive, feature-rich framework to do effective parallel processing. Sometimes, a focused solution that does one thing well is better than a Swiss Army knife that does everything poorly.

The next time you're faced with a computationally intensive problem that could benefit from parallelization, consider whether you really need all the bells and whistles of a complex framework. You might be surprised how far you can get with a simple, well-designed solution.

In our next article, we'll explore how to properly unit test multithreaded code like ComputeCore—because if parallel programming has taught us anything, it's that testing concurrent code requires its own special brand of paranoia.

What are your experiences with parallel processing frameworks? Have you built your own, or do you prefer existing solutions? Share your thoughts in the comments below.