Implementation of cohort parallelisation in the SEAPODYM code

Alexander Pletzer, Chris Scott (NeSI/REANNZ),
Inna Senina, Lucas Bonnin and Romain Forestier (SPC)

September 16, 2025

Abstract

This report describes NeSI/REANNZ's consultancy project entitled "SEAPODYM cohort parallelisation" whose work was performed from May to September 2025. The outcome of this work is a C++ library, `seapodym-parallel`, which can be leveraged by the SEAPODYM code to parallelize the spatio-temporal evolution of tuna densities in the Pacific. The `seapodym-parallel` library implements a variation of the manager/worker setup whereby workers create new fish age cohorts and advances these in time. The manager distributes tasks to the workers and stores results. Our manager-worker design differs from traditional ones by enforcing that tasks only start after certain internal steps of other tasks have completed.

1 Introduction

SEAPODYM is a quantitative spatio-temporal model of population dynamics that solves partial differential equations with initial and boundary conditions in C++. The model is parameterized using a maximum likelihood estimation approach that integrates georeferenced datasets obtained from industrial fishing and scientific campaigns.

The objective of this work is to parallelise the SEAPODYM code to achieve higher performance. This was achieved by designing and implementing the sepodym-parallel library (`github.com/PacificCommunity/seapodym-parallel.git`), a collection of C++ classes that abstract parallelization. The library comes with a `CMake` build system, unit tests and continuous integration via GitHub actions. The documentation of the API can be found at `https://pacificcommunity.github.io/seapodym-parallel/`.

2 What is a cohort?

A cohort is a group of fish that are born at the same time. Each cohort can be integrated forward in time independently of other cohorts, until the cohort reaches a certain age and dies. At the next time step, a new cohort is born and integrated forward in time until it, too, dies. Except at the first time step, the initial condition of each cohort depends on the density of the other cohorts at the previous time step. Thus, the integration of each cohort must take into account the state of the other cohorts and this requires careful consideration when parallelising the code.

An example of cohort time integration is shown below:

$$\begin{array}{ccc} 0 & 1 & 2 \\ 3 & 1 & 2 \\ 3 & 4 & 2 \\ 3 & 4 & 5 \\ 6 & 4 & 5 \\ 6 & 7 & 5 \end{array} \tag{1}$$

Here, the vertical axis represents time increasing from top to bottom ($N_t = 6$). The horizontal axis represents the tasks that can be performed concurrently. Each row represents different age groups (here $N_a = 3$). Each cohort is identified by an integer $(0, 1, \cdots 7)$. Cohort 1 is one unit time older than cohort 2. Therefore cohort 1 dies one time step before cohort 2. The number of time steps a cohort is integrated is $i = i_{beg} \cdots i_{end} - 1$ where $i_{beg} \geq 0$ and $i_{end} \leq N_a$. When a cohort dies, it is replaced by a new cohort at the next time step.

Table 1: Start and end integration indices for each cohort of example (1)

cohort Id	i_{beg}	i_{end}
0	2	3
1	1	3
2	0	3
3	0	3
4	0	3
5	0	3
6	0	2
7	0	1

3 Cohort task dependencies

To instantiate the new cohort, data will need to be communicated from the cohorts at the previous time step. Figure 1 shows the dependency of the new cohorts on the older cohorts at different steps of example (1).

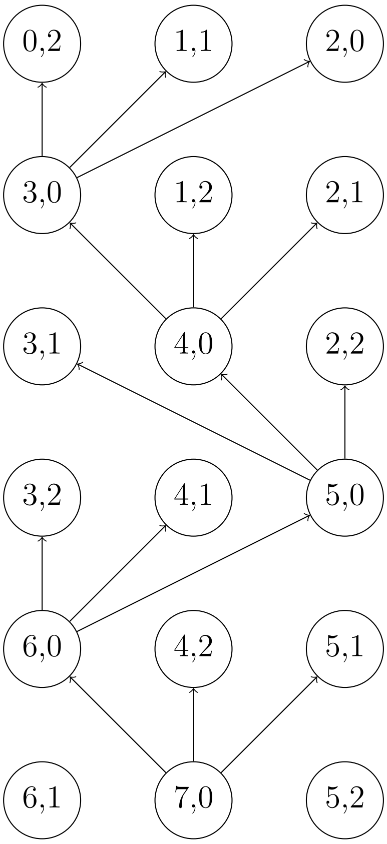

Figure 1: Dependency of cohort tasks on other cohort, step tuples. The first digit is the cohort Id and the second the step.

4 Parallel task farming with dependencies

We opted for a task farming approach, which involves creating a pool of tasks that can be executed concurrently, and a manager that assigns the tasks to the workers. Each task consists of advancing a cohort in time.

In classical task farming, the manager starts by assigning tasks to the workers. The manager sends a message to each worker, along with some input parameters. Each worker then executes the task and reports the results back to the manager, who then assigns a new task to the worker. When no more tasks are available, the manager sends a message to the workers to shut down. The advantage of task farming over other approaches is that resources (workers) and tasks are dynamically matched, often allowing for better load balancing. A possible drawback of task farming is the additional complexity of coordinating the assignment of tasks, additional communication and the need to reserve an MPI rank for the manager.

Clearly, the dependencies between tasks and sub-tasks shown in Fig. 1 prevent us from using a naive task farming approach, since tasks can only be started when other tasks' particular steps have completed (as indicated by the arrows). This requires the manager to keep track of task-step dependencies.

We start by describing a worker's task. Each task has an identifier `taskid` hat fully describes what the task will do. A worker waits for a task to be assigned by the manager. The worker reports back the status of the execution of each task to the manager. The corresponding pseudo-code is shown in Algorithm 1.

Algorithm 1 A worker's pseudo-code.

```
while true do
    taskid = getTaskIdFromManager()
    if taskid < 0 then Break
    end if
                        ▷ Perform the task, stepping from stepBeg to stepEnd - 1
    taskFunc(taskid, stepBeg, stepEnd)
                        ▷ Notify the manager that this worker is available again
    sendSignalToManager(”done”)
end while
```

The task function defined in Algorithm 1 is responsible for advancing the cohort associated with `taskid` from `stepBeg` to `stepEnd - 1`. Note the call

Algorithm 2 The task function executed by the worker.

```
function MyFunction(taskid, stepBeg, stepEnd)
    data = getDataFromManager(taskid)
    cohort = createCohort(taskid)                ▷ Advance a cohort
    for step = stepBeg to stepEnd - 1 do
        stepForward(cohort)
        sendStepCompleteMessageToManager(taskid, step)
    end for
end function
```

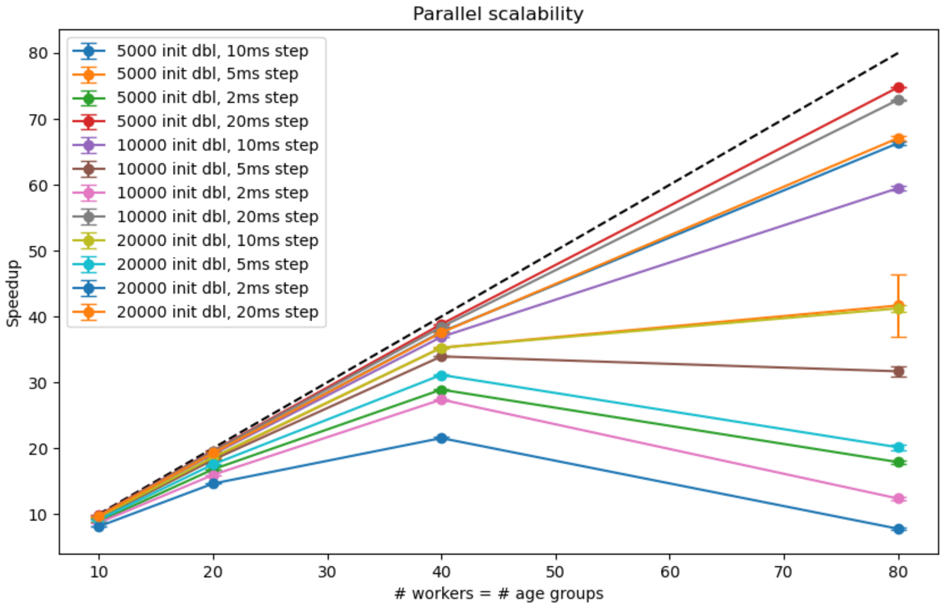

Figure 2: Parallel scaling as a function of the number of workers, for different step execution times and number of doubles initially fetched from the manager. Here the number of age groups equals the number of workers, and these results assume no initialisation/setup overhead.

to `sendStepCompleteMessageToManager` which informs the manager that a specific step for a task has been completed.

The manager code orchestrates the work. It maintains a list of all active workers, a task queue, a list of completed tasks and stores the results. Since workers will be sending various types of messages ("worker is available" or "result of task-step X"), the manager needs to be able to distinguish between these. This is done by using message tags. The manager's pseudo-code is shown in Algorithm 3.

5 Scalability

Figure 2 shows the parallel scalability of the cohort parallelisation approach described above as the number of workers is increased (see code `testTaskStepFarmingCohort`). The speedup numbers were obtained for an idealized scenario, assuming a fixed cost of communication to initialize a new cohort and a constant cost

Algorithm 3 The manager's pseudo-code.

```
while not taskQueue.empty() or not assigned.empty() do
                                        ▷ Look for messages "task-step complete"
    while true do
                                        ▷ Probe for messages from workers
        msg = getMessageFromAnyWorker
        if msg.type != "task-step complete" then
            break
        end if
                                        ▷ Store the result R(e)sults.insertmsg.output
        taskid = output[0]
        step = output[1]
        completed.insert(taskid, step)
        if step == stepEnd - 1 then
            assigned.erase(taskid)
        end if
    end while
                                        ▷ Assign ready tasks to any available worker
    for taskid in taskQueue do
        taskDependencies = getTaskStepDependencies(taskid)
        bool ready = true
        for dep in taskDependencies do
            if  completed.find(dep) == completed.end() then
                ready = false
                break
            end if
        end for
        if ready and not activeWorkers.empty() then
            worker = activeWorkers.begin()
            activeWorkers.erase(worker)
            sendTaskToWorker(taskid, worker)
            assigned.insert(taskid)
            taskQueue.erase(taskid)
        end if
    end for
                                        ▷ Drain all worker-available messages
    for worker in workers do
        msg = getMessageFromAnyWorker
        if msg.type == "worker is available" then
            activeWorkers.insert(worker)
        end if
    end for
end while
                                        ▷ Send stop signal to workers
stop = -1
for worker in workers do
    sendStopSignalToWorker(worker)
end for
```

of executing a step on the NeSI/REANNZ AMD Milan node cluster. In all cases the number of workers was chosen to match the number of age groups N_a.

Perfect parallel scaling is shown as the black dashed line. Good scalability depends on the time taken to execute each step and the amount of data fetched from the manager at the start of each task. Parallel efficiency above 50% is achieved for time step times $\geq$ 10ms and initial data sizes $\leq$ 10000 doubles. For instance, a time step taking 10ms and an initial data size of 10000 doubles yields a speedup of 60 when using 80 workers, i.e. 75% parallel efficiency.

At every step, the worker sends data to the manager, as well as a message to inform the manager that the step has been completed. This introduces a communication overhead that limits parallel scalability. Taking the example of $20,000$ doubles sent by 80 workers to the manager at a cadence of 10ms, we get a data transfer rate of 1.3GB/s = 100Gb/s, which is close to the bandwidth of the Infiniband communication fabric. To further improve parallel scalability, one would need to either increase the computational load per step, reduce the amount of data to be transferred or apply some form of data compression. For instance, one could send floats instead of double and this would improve the speedup from 40 to 60.

6 The cost of initialization

Scalability can be affected by load imbalance. Some concurrently executing tasks may take longer than others to complete. When synchronization occurs, which is the case at every time step in SEAPODYM, then workers whose task has finished need to wait for the slowest worker to finish their task. While the solution of the reaction-diffusion equations takes the same time for all cohorts, the initialization of a new cohort can lead to severe load imbalance and can thus affect the parallel speedup. Fig. 3 shows the ratio of the compute time over the initialization time, the higher the better. Initialization involves reading data from file and other one-off operations.

Let us estimate the effect of a high initialization time on parallel scalability. Let I denote the time it takes to initialize a cohort, C the compute time required to advance the density one time step and N the number of workers. The parallel speedup, defined as the ratio of single worker execution over the parallel execution, is

$$S = \frac{I + NC}{I + C} = N \frac{I/(NC) + 1}{I/C + 1} \tag{2}$$

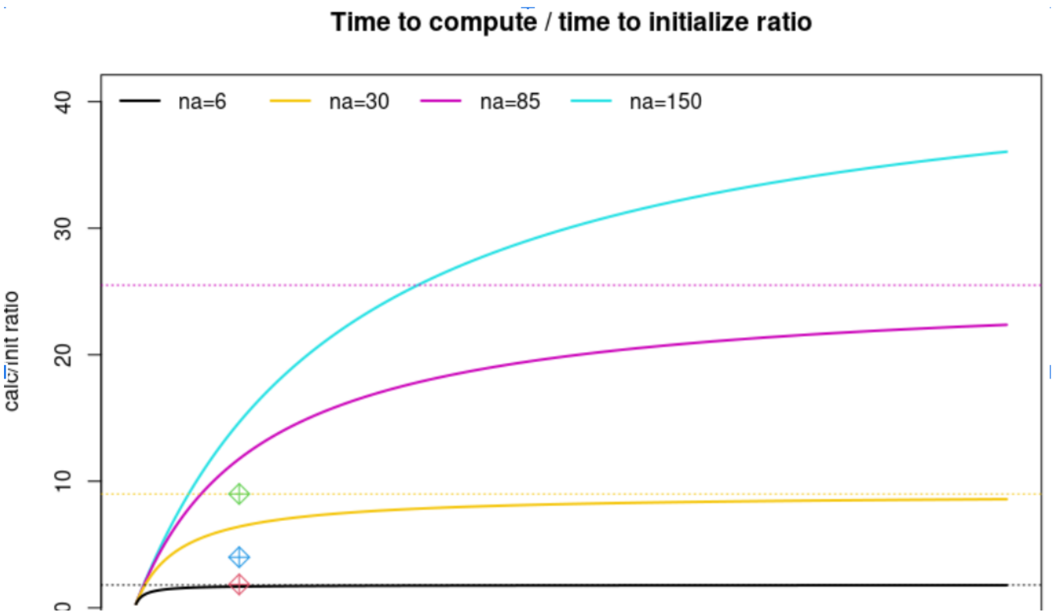

Figure 3: Ratio of compute over initialization time for SEAPODYM.

Note that for large numbers of workers $S \approx N\frac{1}{I/C+1}$ and thus the impact of initialization is to reduce the speedup slope ($I/C > 0$). Figure 3 shows the ratio $(I/C)^{-1}$ for our initial implementation. This ratio, though increasing for large Ns, was close to 1 for moderate N values.

To mitigate the initialization effect on parallel scalability, the cohort initialization step was split into cohort independent and dependent steps. The cohort independent initialization can be performed for all cohorts at the start of the simulation. Since there are $\sim N$ cohorts, this has the effect of amortizing the initialization cost. Only the final stage of cohort construction, which involves reading forcing and spawning data, is executed when launching a new cohort.

7 Comparing blocking and non-blocking get operations

A significant part of initializing a cohort is spent *fetching* data from the manager. Assuming the number of age groups to match the number of workers N, the speedup becomes

$$S = N \frac{I/(NC) + 1}{I/C + DN/C + 1} \tag{3}$$

where the term D represents the time it takes to gather spawning data from a previous time cohort. The term DN/C is the relative cost of communicating the spawning data of the N previous cohorts relative to the compute time of a step. Clearly, communication flattens the speedup curve as N increases. In contrast to the initialization effect, the flattening gets more pronounced with a large number of workers.

One way to lessen the impact of communication is to allow communication and computation to overlap. Our library supports both synchronous and asynchronous `get` operations. The default mode to `get` data is blocking and in this case the retrieved data is ready when the call completes. Alternatively, the programmer may use `getAsync` calls, indicating that the data will be used in the future. After calling `getAsync`, the data is not immediately available. `startEpoch` and `endEpoch` calls before, respectively after `getAsync`, determine the earliest moment the data can be requested and the latest time data must be available. The retrieved data are guaranteed to be available only when `endEpoch` is called. (An additional "flush" method can be invoked to synchronize operations within a period.)

In Fig. 4 we compare the execution time of MPI `get` operations using the blocking and non-blocking variants. Each test (implemented in `testAsyncPutGet`) involves reading N chunks (= number of workers) of nd values from rank 0 and summing the result. In the blocking test, the chunks are read one at a time, followed by a sleep operation lasting nm milliseconds and a sum operation over the chunk. In the nonblocking test, a big array is filled by reading the chunks asynchronously, followed by the sleep operation – these two operations are within a single epoch. The final step then involves summing up the values of the big array. The benefit of using non-blocking `get` calls is particulary apparent for large chunks nd and small amount of work (low nm).

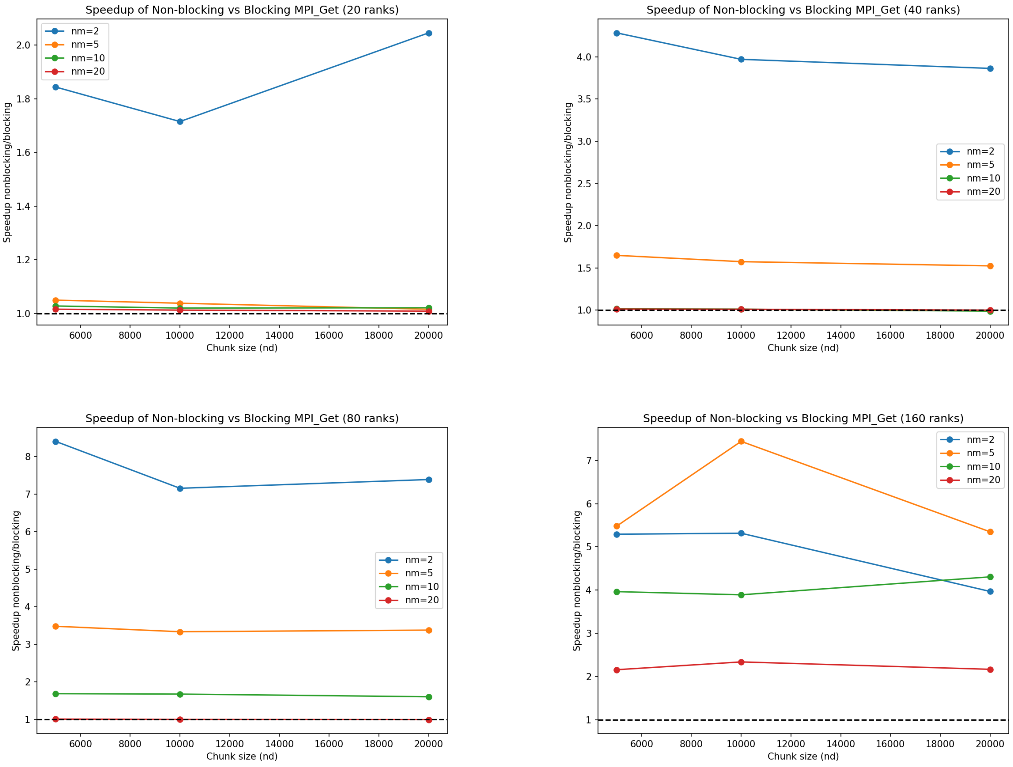

Figure 4: Comparing non-blocking with blocking communication for different execution time steps (nm in milliseconds) and chunk/data sizes (nd is the number of doubles). Values > 1 indicate a benefit of using non-blocking "get" operations.

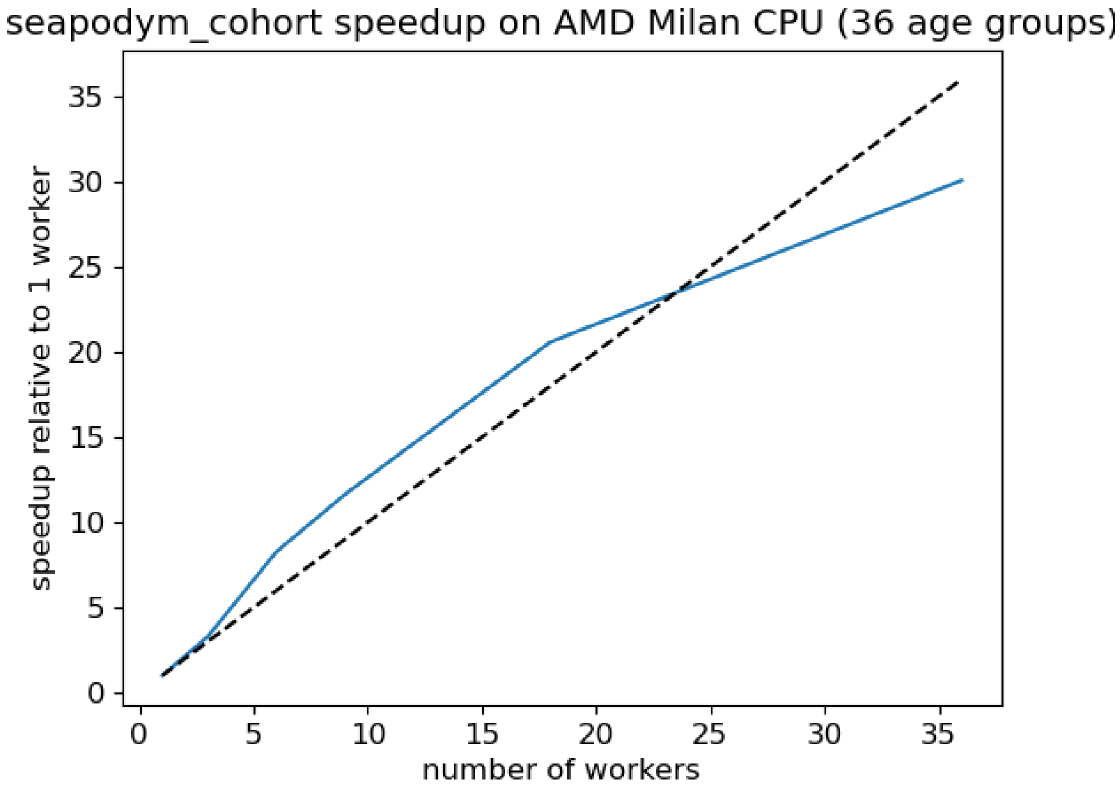

Figure 5: Parallel speedup relative to a single worker run for a SEAPODYM test.

8 Parallel Scaling for SEAPODYM

The parallelization strategies presented in the previous sections have been implemented into the SEAPODYM code. We report a scalability test involving 36 age groups (96 cohorts), run over the period 1979-1983 (60 time steps). All the runs (Fig. 5) were performed on REANNZ AMD Milan platform using a single node.

Note the better than linear scalability between 3-18 workers. Possible reasons for this behaviour are: (1) with more workers processes handle smaller amounts of data which may fit better in cache, (2) with more workers the scheduling overhead distributes work more evenly (reduction of load imbalance), (3) MPI may pipeline messages more efficiently with additional ranks and (4) disk and I/O contention may be decreased.

9 Summary and future work

A parallel library (`seapodym-parallel`) has been developed to parallelize the SEAPODYM code. While the library has been tested independently and a version of SEAPODYM now runs with this library, more checks are warranted. The SEAPODYM code comes in different flavours, parallelization has been added at present only to the version that evolves the fish densities in time. Extending parallelization to other SEAPODYM versions, including one that maximizes the likelihood function remains to be done.

The parallelization is based on a task farming approach with additional features: (1) tasks have sub-tasks (steps), (2) tasks can be started only when other tasks' specific steps have completed and (3) data is sent to the manager at the end of each step. The manager stores and distributes the data to workers on request using one-sided MPI communication with passive targets. As such, the manager does not actively contribute to the data exchange, it is the workers who decide when to fetch and send data.

In our design, all data exchanges are channeled through the manager. Hence, the (spawning) data required by workers at the start of a new cohort cannot be directly received from other cohorts (or workers) but always pass through the manager. The rationale for this choice is to facilitate the implementation of parallelism in other versions of SEAPODYM.

Nevertheless, we demonstrated that parallel efficiency of ≈ 84 % can be achieved for 36 age groups and one worker per age group using blocking communication. The manager's execution time dropped from 60 to 2 seconds, approximately a 30x speedup. Additional gains can potentially be attained by using non-blocking communication, a feature supported by `seapodym-parallel`. However, care should be taken to ensure correct synchronization. A common mistake is to consuming fetched data before the "get" operation has completed; another is requesting data from the manager that has not yet been populated. Therefore, we recommend careful profiling, and replacing blocking by non-blocking calls only in the regions that are known to be performance intensive.