Implementation of cohort parallelisation in the Seapodym code

Alexander Pletzer, Chris Scott,
Inna Senina, Lucas Bonnin and Romain Forestier

September 11, 2025

This report describes NeSI/REANNZ's consultancy project entitled "SEAPODYM cohort parallelisation" whose work was performed from May to September 2025.

1 Statement of the problem

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 over cohorts. 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 dies. Except at the first time step, the initial conditions of each cohort, however, 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:

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

Here, the vertical axis represents time increasing from top to bottom ($N_t = 6$). The horizontal axis represents the tasks than can be performed concurrently. Each row represents different age groups ($N_a = 3$). Moreover, 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. Likewise, cohort 0 is one time step older than cohort 1 and thus dies after the first time step. The number of time steps a cohort is integrated is $i = i_{beg} \cdots i_{end} - 1$ where $i_{beg} \geq 0$ and $i_{end} <= N_a$. Going back to our example, we have When a cohort dies, it is replaced by a

Table 1: Start and end integration indices for each cohort

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

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

2 Task farming with dependencies on other tasks' steps

We opted for a task farming approach. This 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 for multiple time steps.

In a classical task farming implementation, the manager starts by assigning tasks to the workers. This involves sending 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. Task farming is particularly suitable when tasks

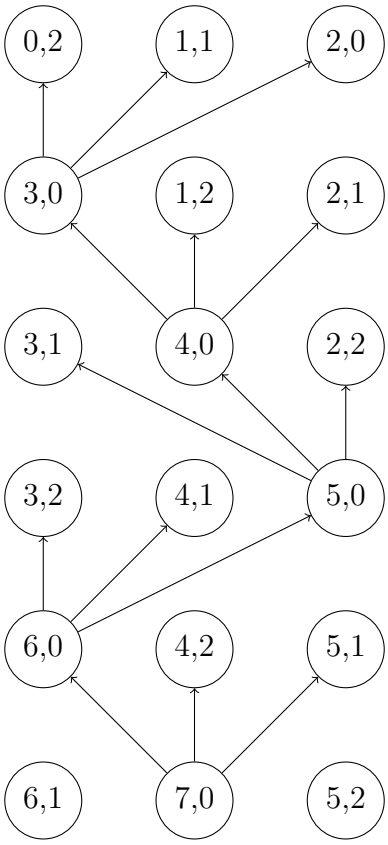


Figure 1: Dependency of cohort tasks on other cohort, step tuples. The first digit is the cohort Id and the second the step.

take a different amount of time to execute, either because of the nature of the task or because of of varying computational loads or because of task heterogeneity.

Clearly, the dependencies between tasks and sub-tasks shown in Fig. 1 prevents us from using a naive task farming approach. In our case, tasks can only be started when other task steps have been completed. In other words, a task corresponding to advancing a cohort at a given step can only be started once all its prerequisite tasks (as indicated by the arrows in Fig. 1) have finished. This requires the manager to keep track of task dependencies and only dispatch tasks to workers when their dependencies are satisfied.

We start by describing a worker's task. A worker waits for a task to be assigned by the manager. A task has a identification number `workerId` that fully describes the task to accomplish. The worker reports back the status of the execution of each task to the manager. The corresponding pseudo-code is shown in Algorithm 2.

Algorithm 1 A worker's pseudo-code.

```
1: while true do
       taskid = getTaskIdFromManager;
2:     if taskid < 0 then Break
3:     end if
                    ▷ Perform the task, stepping from stepBeg to stepEnd - 1
4:     taskFunc(taskid, stepBeg, stepEnd)
                    ▷ Notify the manager that this worker is available again
5:     sendSignalToManager(DONE)
6: end while
```

The task function defined in Algorithm 2 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.

```
1: function MyFunction(taskid, stepBeg, stepEnd)
2:     data = getDataFromManager(taskid)
3:     cohort = createCohort(taskid)                ▷ Advance a cohort
4:     for step = stepBeg to stepEnd - 1 do
5:         stepForward(cohort)
6:         sendStepCompleteMessageToManager(taskid, step)
7:     end for
8: end function
```

to `sendStepCompleteMessageToManager` which informs the manager that a

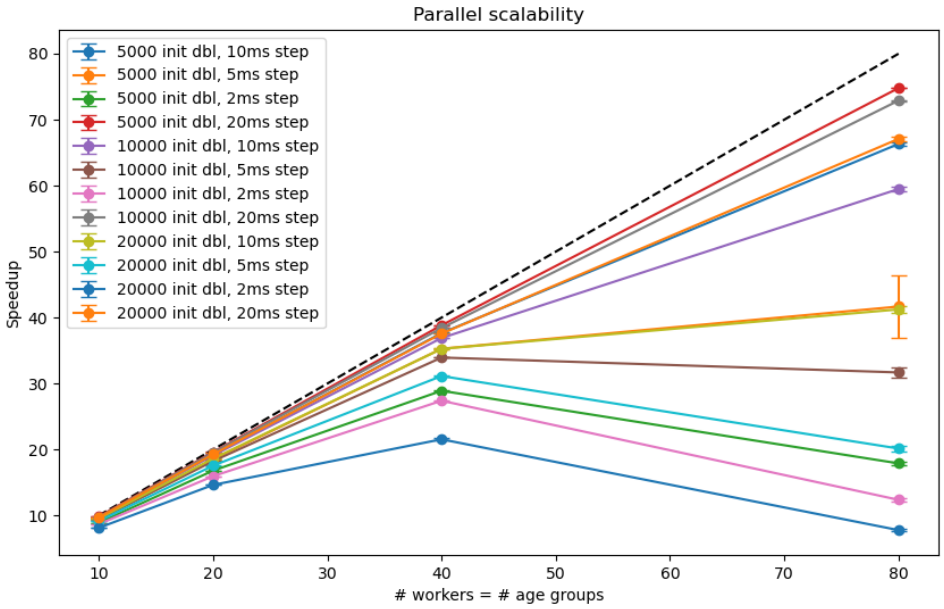


Figure 2: Parallel scalability for different step execution times and number of doubles initially fetched from the manager. The number of age groups matches the number of workers. These results assume no initialisation setup time.

specific step for a task has been completed. This is crucial for managing task dependencies, as the manager needs to know when a task's prerequisites have been satisfied.

The manager code orchestrates the work. It maintains a list of all active workers, 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 step of task X"), the manager needs to be able to distinguish between them. This is done using message tags. The manager's pseudo-code is shown in Algorithm 3.

3 Scalability

Figure 2 shows the parallel scalability of the cohort parallelisation approach described above as the number of workers is increased. The results were obtained on the NeSI/REANNZ cluster by running on the AMD Milan nodes.

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
        results.insert(msg.output)
        taskid = output[0]
        step = output[1] completed.inserttaskid, 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 = getTaskStepDepenenciestaskid 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 !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 s(e)ndStopSignalToWorkerworker
end for
```

In all cases the number of age groups matches the number of workers.

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 $> 50\%$ 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 gives a speedup of 60, 75% parallel efficiency, when using 80 workers.

Note that 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 flots instead of double and this would improve the speedup from 40 to 60.

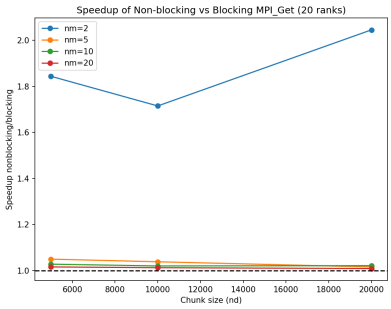
4 The cost of intitalization

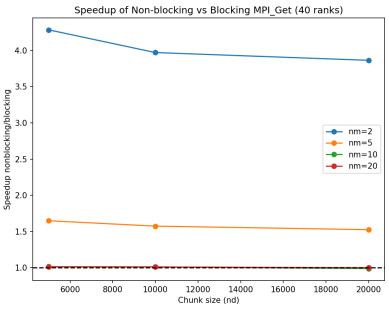
5 Comparing blocking and non-blocking get operations

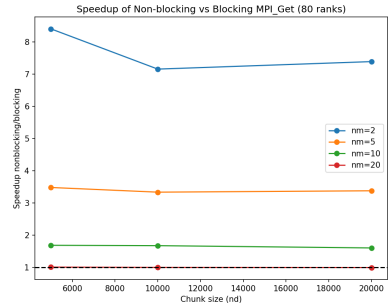
A significant part for the initialization time involves fetching data from the manager. This takes place at each time step. Our library support asynchronous get operations, which allow one to overlap communication with computation.

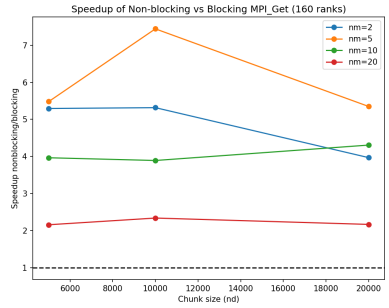
The default mode to get data is blocking, that is that retrieved data are ready when the "get" call completes. Alternatively, the programmer may rely on "getAsync" calls. In this case the programmer indicates his/her desire to fetch the data but the data need not be immediately available. A "startEpoch" and "endEpoch" calls determine the the time when remote memory access operations take place with "endEpoch" indicating that the data must available at that point. (An additional "flush" can be used) to request completed "get" operations within a period.

6 Summary and future work


Speedup of Non-blocking vs Blocking MPI_Get (20 ranks)
nm=2
nm=5
nm=10
nm=20
Speedup nonblocking/blocking
Chunk size (nd)


Speedup of Non-blocking vs Blocking MPI_Get (40 ranks)
nm=2
nm=5
nm=10
nm=20
Speedup nonblocking/blocking
Chunk size (nd)


Speedup of Non-blocking vs Blocking MPI_Get (80 ranks)
nm=2
nm=5
nm=10
nm=20
Speedup nonblocking/blocking
Chunk size (nd)


Speedup of Non-blocking vs Blocking MPI_Get (160 ranks)
nm=2
nm=5
nm=10
nm=20
Speedup nonblocking/blocking
Chunk size (nd)