Implementation of cohort parallelisation in the Seapodym code

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

September 11, 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 fish 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. The implemented manager/worker design differs from traditional implementations in that tasks can only be started when other tasks have completed specific internal steps.

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 over cohorts. 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. This 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 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:

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

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 ($N_a = 3$). 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 $i_{end} <= N_a$. In our example, we have 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

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

To instantiate the new cohort, data will need to be communicated from the cohorts at the previous time step.

3 Cohort task dependencies

Figure 1 shows the dependency of the new cohorts on the older cohorts at different steps.

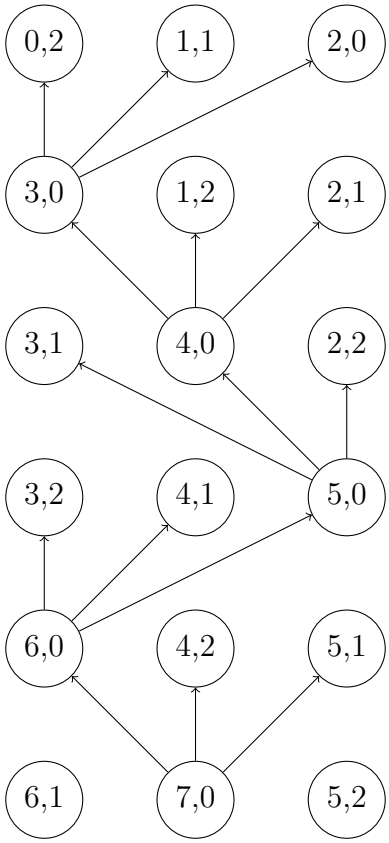


Figure 1: Dependency of cohort tasks on other cohort, step tuples. The first digit is the cohort Id and the second the step.

4 Parallel task farming with dependencies

We opted for a task farming approach. 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 in time.

In classical task farming, 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 take a various amount of time to execute.

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

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 `taskid` 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 specific step for a task has been completed. This is crucial for managing task dependencies.

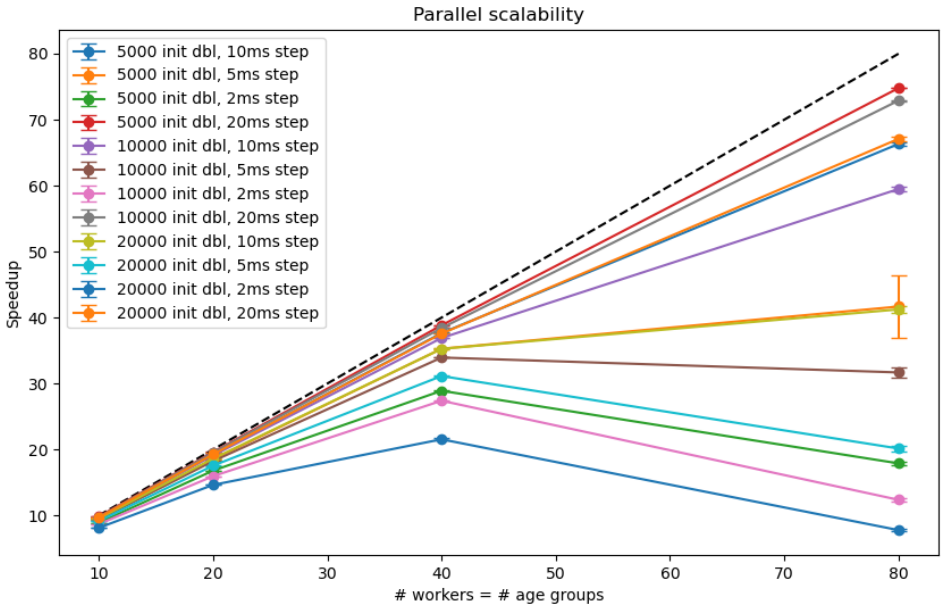


Figure 2: Parallel scalability for different step execution times and number of doubles initially fetched from the manager. The number of age groups matches the number of workers. These results assume no initialisation setup time.

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 them. This is done 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 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

Algorithm 3 The manager's pseudo-code.

```
1: while not taskQueue.empty() or not assigned.empty() do
                                   ▷ Look for messages "task-step complete"
2:     while true do
                                              ▷ Probe for messages from workers
3:         msg = getMessageFromAnyWorker
4:         if msg.type != "task-step complete" then
5:             break
6:         end if
                                                          ▷ Store the result
7:         results.insertmsg.output
8:         taskid = output[0]
9:         step = output[1] completed.inserttaskid, step
10:        if step == stepEnd - 1 then assigned.erasetaskid;
11:        end if
12:    end while
                                  ▷ Assign ready tasks to any available worker
13:    for taskid in taskQueue do
14:        taskDependencies = getTaskStepDependenciestaskid
15:        bool ready = true
16:        for dep in taskDependencies do
17:            if completed.finddep == completed.end then
18:                ready = false
19:                break
20:            end if
21:        end for
22:        if ready and !activeWorkers.empty then
23:            worker = activeWorkers.begin
24:            activeWorkers.eraseworker
25:            sendTaskToWorker(taskid, worker)
26:            assigned.inserttaskid
27:            taskQueue.erasetaskid
28:        end if
29:    end for
                                    ▷ Drain all worker-available messages
30:    for worker in workers do
31:        msg = getMessageFromAnyWorker
32:        if msg.type == "worker is available" then
33:            activeWorkers.insert(worker)
34:        end if
35:    end for
36: end while
                                                6     ▷ Send stop signal to workers
37: stop = -1
38: for worker in workers do s(e)ndStopSignalToWorkerworker
39: end for
```

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 $> 50\%$ 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, when using 80 workers, i.e. 75% parallel efficiency.

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 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 balancing. Some concurrently executed 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 woprker to finish their task. While the evolution of the reaction-diffusion equations takes the same time for all cohorts, the initialization of a new cohort, which occurs at every time step, can lead to severe load imbalance and can thus affect the parallel speedup. Fig. 3 shows the 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.

There are different ways to amortize the high cost of initialization. One possibility is to avoid constructing a new cohort at every time step. Instead a "vanilla" cohort is created initially, when the simulation start. This cohort, reads the restart files, the input files and set things up. Only the final stage of the construction, which involves reading forcing data and spawning, is executed when launching the new cohort.

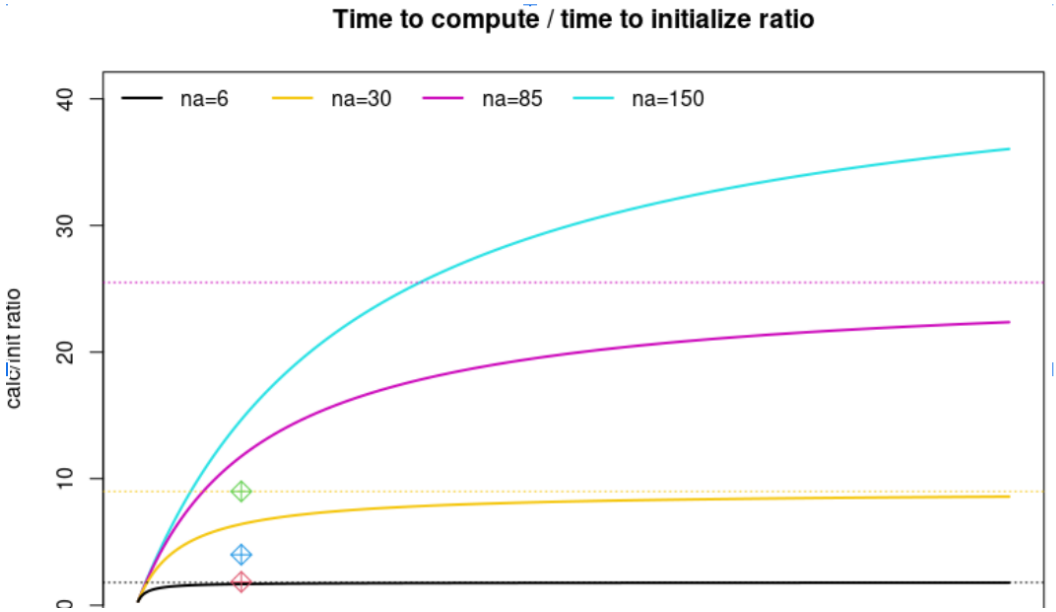


Figure 3: Ratio of compute over initialization time for Seapodym.

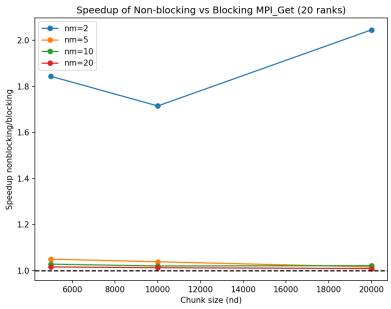
7 Comparing blocking and non-blocking get operations

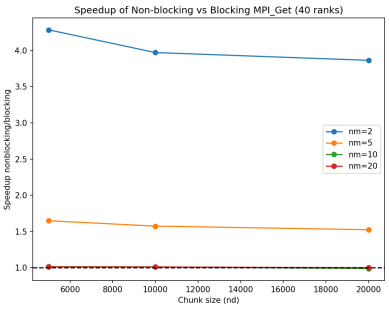
A significant part for the initialization time involves fetching data from the manager. Our library support asynchronous get operations, which allow one to overlap communication with computation.

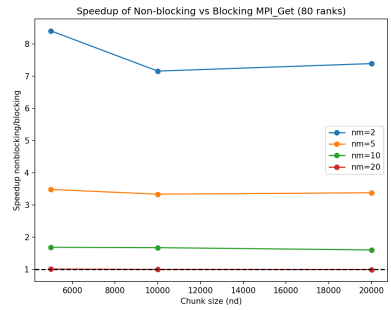
The default mode to `get` data is blocking, that is the retrieved data are ready when the call completes. Alternatively, the programmer may rely on `getAsync` calls. In this case the programmer indicates his/her desire to fetch the data but the data need not be immediately available. A `startEpoch` and `endEpoch` calls around the `getAsync` operations determine the time interval when remote memory access operations take place. The retrived data are guaranteed to be available only when `endEpoch` is called. (An additional "flush" medthod can be invoked to synchronize operations within a period.)

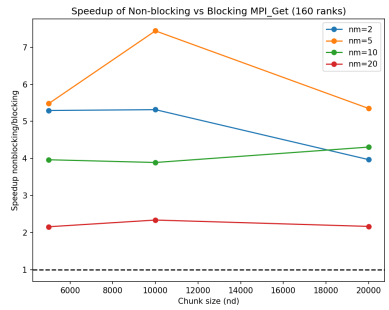
In Fig. 5 we compare the execution time of MPI `get` operations using the blocking and non-blocking variants. Each test (implemented in `testAsyncPutGet`) involves reading N chunks (= number of workers) of nd values from rank 0 and summing the result. In the blocking test, the chunks are read one at 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 particulalry apparent for large chunks nd and small amount of work (low nm).

8 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)