New Zealand eScience Infrastructure (NeSI)
Pacific Communities (SPC)

A review of the SEAPODYM code to determine its suitability for parallelization

by

Alexander Pletzer, Chris Scott and Inna Senina

A report submitted to SPC in response to Request for Quotation RFQ-4905

2023

This report describes NeSI's consultancy project entitled "SEAPODYM parallelisation" whose work was performed from May to August 2023.

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 undertake a review of the SEAPODYM computer code, and to provide potential solution(s) to enable a parallel execution while preserving the current functionality of the code.

2 Profiling

Key to improving the code's performance is to identify execution bottlenecks. This can be achieved by profiling the code. Here we list two ways to accomplish this on NeSI's mahuika platform. None of these approaches require the code to be recompiled and both methods involve calling the executable by prepending a special command.

2.1 Using Forge (ARM) map

Below is an example of a SLURM job that profiles the `seapodym_densities` executable.

```
cd example-configs/nesi-test
cat > skipjack_densities-map.sl << EOF
#!/bin/bash
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 1
#SBATCH -p milan
#SBATCH -t 00:30:00
#SBATCH -J skipjack
#SBATCH --error %x-%j.err
#SBATCH --output %x-%j.out
#SBATCH --hint=nomultithread
#SBATCH --mem 2gb
```

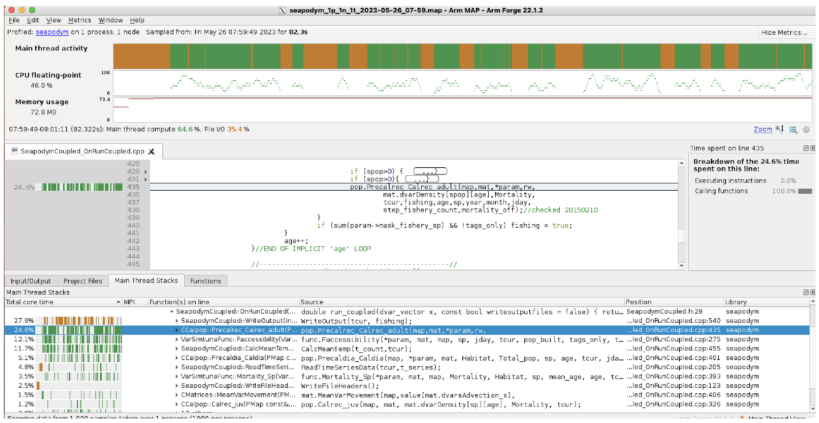

Figure 1: Screenshot of Forge (ARM) map profiler.

```
module purge
module load gimkl
module load forge

map --profile ../../bin/seapodym_densities initparfile.xml
EOF
```

Note the `map --profile <exec>` command. Upon completion of the run, a `.map` file will be created. Use the command `map <file.map>` to determine the execution time, memory footprint and other useful information. Figure 1 shows a screenshot. At the top we see the code and the time spent on some lines. At the bottom we see the call stack and the self time spent in some functions.

2.2 Using Intel's advisor

```
cat > skipjack_densities-advisor.sl << EOF
#!/bin/bash
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 1
#SBATCH -p milan
#SBATCH -t 00:30:00
#SBATCH -J skipjack
#SBATCH --error %x-%j.err
#SBATCH --output %x-%j.out
#SBATCH --hint=nomultithread
#SBATCH --mem 2gb
```

```
module purge
module load gimkl
module load Advisor

EXE=../../bin/seapodym_densities
SRC1=../../src
SRC2=/nesi/nobackup/pletzera/admb/

advisor --collect=survey --project-dir=./advisor-${SLURM_JOB_ID} \
--search-dir src:r=${SRC1} --search-dir src:r=${SRC2} -- \
${EXE} initparfile.xml
EOF
```

Submitting the job will produce an `advisor-<job number>` directory. The Intel Advisor GUI can be use to open the result, e.g. on https://jupyter.nesi.org.nz/:

1. Select Virtual Desktop from the launcher
2. Open a terminal and navigate to the directory with the result
3. Run `ml purge && ml Advisor`
4. Run `advisor-gui`
5. Select the option to open result and navigate to the `advisor-<job number>` directory and open the `advixeproj` file in there

See Fig. 2 for a screenshot of `advisor-gui`.

3 Parallelization strategies

Below we list some parallelization methods for CPUs, with their advantages and shortcomings. All these approaches first require the identification of a computationally expensive loop. Hence, collecting profiling information from the code is essential. Note that these approaches are not mutually exclusive and can be combined to reduce the execution time.

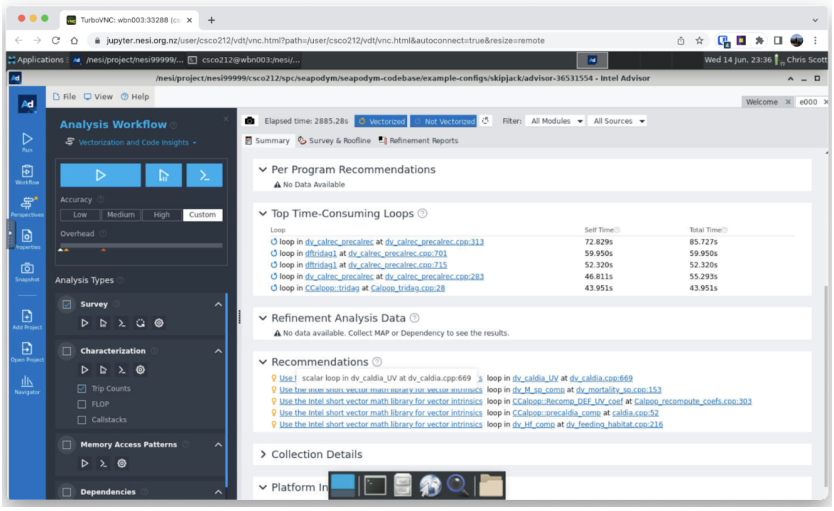

Figure 2: Screenshot of advisor-gui.

Approach	Pros	Cons
vectorization	gradual introduction uses existing compute resources code is portable	limited performance improvement no gain if memory is bottleneck Amdahl's law applies
OpenMP	gradual introduction portable code	number of cores per node limit no gain if memory is bottleneck Amdahl's law applies cost of starting threads
MPI	highly scalable less portable	development investment communication may limit scalability

4 Vectorization

Modern CPUs allow for multiple loop iterations to be executed simultaneously. For instance, Intel Skylake supports up to 512 bit wide vector instructions. Thus, in the case of a double array, up to 8 instructions can be executed in parallel, yielding a maximum 8x speedup.

Compilers will automatically vectorize some loops when some optimization switches are turned on. These are

Compiler	Vect. options	report missed optimizations
GNU	-O3 -march=native	-ftree-vectorizer-verbose=2
Intel	-O2 or higher	-qopt-report=3

It's important to realize, however, that not all loops can be vectorized. Some reasons preventing vectorization are

Issues preventing vectorization	Possible solution
Loop count is not known at runtime	replace while loop with a for loop
Dependency on other iterations	create a temporary variable
Data structures may overlap in memory	remove aliasing

When the compiler is hesitant to vectorize a loop and the programmer is confident that the loop can be vectorized, it is possible to coerce the compiler using the following pragma in C++

```
#pragma omp simd [options]
{
  for (int i = 0; i < n; ++i) {
   <loop code goes here>
  }
}
```

The `#pragma omp simd` tells the compiler that there is no dependency in the loop, which can then be safely vectorized. For GNU g++ the option is -fopenmp-simd. In the absence of this compiler option, the compiler will simply ignore the pragma. (Other, non-OpenMP compiler specific pragmas, e.g. `#pragma GCC ivdep Unroll Vector`, exist but these are not portable and thus we do not recommend them.)

The compiler might still be able to vectorize in the absence of the pragma in some cases but this requires `-O3` with the GNU compiler. It is possible to get a report of missed loop vectorization with -fopt-info-omp-vec-optimized-missed.

The advantage of vectorization is the code will run more efficiently without requiring additional resources. The disadvantage of vectorization is that the speedup tends to be rather modest, of the order of 2-8 at most depending on how whether the processor supports Intel MMX, SSE, AVX or AVX512 instruction sets (or equivalent on other processors).

We found loops that iterate over `dvector` and `dmatrix` objects to be non-parallelizable. The problem may be due to the compiler generating a try block when these objects are created in order for the object to be deallocated when an exception is thrown. Thus, vectorization works best with pointer arrays, which in general can be created by fetching the address of the first element

```
double* a_ptr = a[a.indexmin()];
```

and use `a_ptr` inside the loop. A similar problem exists with OpenMP parallelization, see next section.

5 OpenMP multithreading

OpenMP is an API for shared-memory multiprocessing programming with support for C, C++ and Fortran. Modern computers tend to have a large number of cores. For example the new Milan nodes at NeSI have 128 physical cores per node, while the older Mahuika nodes have 36 physical cores per node. By using OpenMP you can take advantage of multiple cores within the same machine (or node).

The number of threads can be specified with the environment variable `OMP_NUM_THREADS`, for instance

```
export OMP_NUM_THREADS=4
```

Under SLURM you will need, in addition, to specify somewhere at the top of your SLURM submission script,

```
#SBATCH --cpus-per-task=4
```

OpenMP primarily uses compiler directives (or pragmas) to indicate to the compiler which sections of code (often loops) can be parallelized and can be controlled by the use of compiler flags and environment variables. With g++ the flag to enable OpenMP is -fopenmp.

Before parallelizing a code with OpenMP it is important to profile the code using a tool such as ARM map, Intel Advisor or VTune. These tools will show which loops are taking the most time and therefore should be considered for parallelization first. Key considerations when deciding whether a particular loop can be parallelized include whether each iteration of the loop can be executed independently of other iterations, or whether it can be rewritten in such a way. This is important because if two OpenMP threads try to modify the same element of an array at the same time (for example), you have a data race condition which will lead to undefined behaviour. Another important consideration is whether any libraries are being used and, if so, it is important to make sure those libraries are "thread safe."

The following pragma can be used to parallelize a loop with OpenMP:

```
#pragma omp parallel for [options]
{
  for (int i = 0; i < n; ++i) {
   <loop code goes here>
  }
}
```

During this project we identified the `dv_calrec_precalrec` function as being one of the most time consuming functions in the code and have demonstrated how this function could be parallelized with OpenMP. The implementation can be found here `https://github.com/chrisdjscott/seapodym-codebase/tree/omp-dev`.

The use of dmatrix and dvector types made adding OpenMP a bit harder as they did not work well with OpenMP (or vectorization), due to the copy constructors of those types doing shallow copies rather that deep copies. The approach taken here was to create local copies of the pertinent data structures (`uvec_threads` for `uvec`, etc.) for each of the threads before entering the loops. Thus, each thread gets its own copy of these arrays to ensure they do not clash with each other. Here we use the `omp_get_thread_num()` function to get the thread ID, which will be unique to each thread. That ID is then used later in the loop to ensure each threads uses different temporary arrays, i.e. each thread will access its own version of `rhs_threads` in this example.

```
#pragma omp parallel for [options]
{
  #pragma omp parallel for
  for (int j = map->jmin; j <= map->jmax; j++)
  {
    const int tid = omp_get_thread_num();
    <...>
    for (int i = imin; i <= imax; i++)
      rhs_threads[tid][i] = <...>
    <...>
  }
}
```

In order to allow the same code to run with or without OpenMP, we have added an extra file "openmp_helper.cpp", which defines some values for OpenMP functions in the case the code is running without OpenMP enabled.

With the above changes we ran a benchmark on 4 cores. Overall execution time dropped from 25 minutes to approximately 22 minutes, which is a modest improvement although it is likely to drop further if more cores were used and if more loops in the code were converted to OpenMP. Memory usage did increase slightly, due to the creation of extra copies of data structures for each thread, going from around 1.8 GB to 1.9 GB. It may be possible to rewrite those loops to not need as many intermediate arrays if memory usage were a concern.

6 Message Passing Interface (MPI)

The Message Passing Interface is a high level of parallel programming approach, which is suitable to run on massively parallel architectures with thousands or more processing elements. (For simplicity we can think of a processing element as being a physical core.) In contrast to OpenMP and vectorization, each processing element acquires its own memory. Thus, the memory footprint of an MPI application can be spread across many cores and this can be an attractive feature. For example, an application requiring 1TB of memory could run on 100 processing elements, each requiring only 10GB of memory.

Communication between processing elements occurs through library calls. MPI bindings exist for C/C++ as well as many other programming languages.

Prior to running an MPI code, users typically request the number of processing elements (which remains constant during the simulation) with

```
mpiexec -n N <executable> <arguments>
```

where N is the number of processing units. At NeSI, the user will typically write a SLURM script to submit the job with

```
#SBATCH --ntask=N
...
srun <executable> <arguments>
```

where `srun` will be replaced by `mpiexec -n N` and the value N will be taken by the `#SBATCH --ntask=N` setting (or by the `sbatch` command line argument of the same name).

A large value N is in general desirable. Here we discuss a parallelization approach over age cohorts. The maximum number of processing elements in this case would be the number of age groups, of which there are anywhere between 10s to 100s in a simulation, depending on the case.

6.1 Parallelization over cohorts

A cohort describes a group of individuals born within the same time interval. As the simulation proceeds, members of a given cohort become old and a small fraction of a cohort that is left at defined maximal time (age) gets passed to the A+ group (a group that captures all the individuals still alive), or, in this toy model, dies out. Once initial conditions are prescribed, a cohort can be integrated forward until its members die. Naturally, the number of

time steps until death depends on the age of the cohort's members, which is different between cohorts. Associating a processing element to each cohort would therefore lead to significant load balancing issues.

It turns out that it is possible to assign different cohorts to the workers (or processing elements) in such a way that each worker accomplishes the same amount of work. Figure 3 shows an example of a configuration with 4 workers and $n_t = 5$ time steps. Each column represents a worker and each block a piece of work corresponding to the integration of the advection-diffusion-reaction system over a single time step. Clearly, the number of blocks is the same for each worker. Such a configuration works for any number of age groups and number of time steps.

Worker 0 starts with the youngest cohort (grey) and integrates the equation for four steps until the members of this cohort die. After this, worker 0 can integrate the members of cohort 7 for one time step after which point the simulation terminates (since there are 5 time steps). However, in order for cohort 7 to start integrating, data must be fetched from cohorts 6, 5, and 4 from the end of the previous step.

Worker 1 starts by integrating forward cohort 1. Since cohort 1 was born one step before cohort 0, only three steps are executed after which point worker 1 turns its attention to cohort 6. In order for cohort 6 to start, data produced by cohort 0, 5, and 4 at the end of the previous step must be accessed.

It is also possible to attach multiple columns to one worker. For instance, worker 0 could handle columns 0 and 1 and worker 1 columns 2 and 3 if one had only two processing elements

Choosing the unit of block's execution time to be one, the total serial execution time then is $n_t \times n_a$ units (n_t is the number of time steps and n_a is the number of age groups). When there are n_a workers, each worker can run in parallel and takes the same amount of time (n_t) to execute under the assumption of zero communication cost. Therefore the highest, achievable parallel speedup is n_a. In practice, the communication cost cannot be neglected for a large number of workers.

We have implemented a toy model in Python `https://github.com/pletzer/cohort_parallel` that emulates some of the behaviour of SEAPODYM when advancing the advection-diffusion-reaction equations forward in time. The critical features of the model are: (1) the user can set the execution time for one cohort time step and (2) the number of double float values that a worker needs to share with another worker to initialize a new cohort.

The cohort execution time for a time step depends on the configuration, the spatial resolution ($\sim$ 1 deg.) and the number of alternating direction implicit (ADI) iterations ($\sim$ 15). The execution time for one time step and

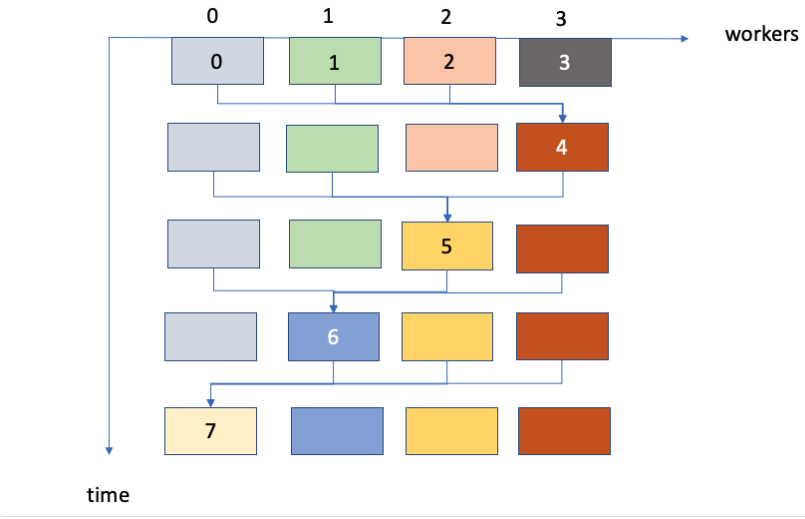

Figure 3: Execution pattern of the cohort parallelization approach for the case where there are 4 age groups, 4 workers and 5 time steps. Each square represents the execution of one time step. The different colours represent the different cohorts and the connecting arrows the data dependencies. Multiple arrows pointing to the same block represent a sum.

one cohort was estimated to be about 0.015 seconds. Up to 24000 doubles are summed across cohorts and we used this number in the following.

The toy model is organized around the following objects:

- The `Task` class. A task knows how to execute a time step. Data are attached to each task.
- The `TaskManager` class. A task manager knows how many time steps can be advanced, whose cohorts hold the initial condition data and what cohort can take over once the current one dies.
- The `Simulation` assigns workers, knows how to retrieve the data from the tasks, organizes the communication between workers and runs the simulation. All MPI calls are elevated to the simulation.

We propose to use MPI one-sided communication to simplify the logic of sending and accumulating the initial condition data. The advantages of one sided communication over point-to-point communication (send/receive) are: (1) the data producer does not need to know which worker will consume the data and (2) one-sided communication allows for some message transfer optimizations.

With one-sided communication, workers need to define a window of data. We decided to use the pull approach, i.e. the data consumer reads the remote data. Additional efficiencies can be achieved compared to using send/receive pairs because the exposed data window stores all the information about the data type and the size of the messages. Thus, a connection pipeline can be established prior to sending any data and this can reduce message latency.

The creation of a data window resembles

```
rcvWin = MPI.Win.Create(rcvData, comm=comm)
```

where `rcvData` is an array of doubles and `rcvWin` the MPI window object.

The following then sums the data from all workers

```
rcvWin.Accumulate(srcData, \
                  target_rank=worker_id, \
                  op=MPI.SUM)
```

where `srcData` is the name of the array that stores on the data on the remote processes. Array `srcData` has the same size and is of the same type as `rcvData`.

It is critical to understand that `Get` and `Accumulate` operations require synchronization across workers to ensure that the data are ready to be transferred at the right time. This is achieved with "fence" calls where all the workers come together. In practice, some workers might finish their task faster because of random fluctuations of load on the platform or for other reasons. When this happens, some workers will have to wait while other workers catch up. Synchronization reduces parallel scaling, but cannot, unfortunately, be fully avoided.

Fences come in two flavours: first there must be a call to indicate that the data to be fetched must be ready,

```
rcvWin.Fence(MPI.MODE_NOPUT | MPI.MODE_NOPRECEDE)
```

before invoking `rcvWin.Accumulate`. Since `rcvWin.Accumulate` is non-blocking, we must also inform the program when the received data are ready

```
rcvWin.Fence(MPI.MODE_NOSUCCEED)
```

In general, it is best to have the first "fence" as early in the program execution stream as possible and the second as late (i.e. when the data are required). This allows for communication and computation to overlap.

Figure 4 shows the parallel efficiency, defined as the ratio of the ideal parallel execution time over the observed execution time, for the toy model. A parallel efficiency of 1 indicates that the code has achieved a speedup equal to the number of cores (or workers). This can only arise if there is no communication cost and perfect load balancing between the workers. A

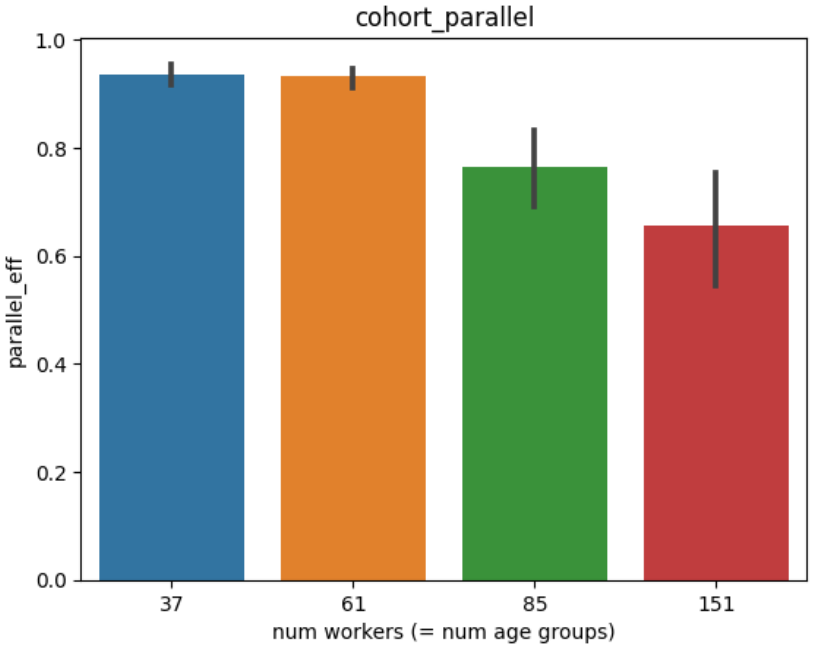

Figure 4: Parallel efficiency of the toy model for $n_a = 37$ (skipjack), 61 (yellowfin), 85 (bigeye) and 151 (albacore). A parallel efficiency of 0.6 for the 151 worker model corresponds to a speedup of 91x.

parallel efficiency of 0.5 occurs if half the execution time is spent either in communication and/or because workers had to wait for other workers to complete their task.

Different configurations (skipjack, yellowfin, bigeye and albacore) were used in Fig. 4. All cases were run on AMD Milan nodes (1 node for skipjack, yellowfin and bigeye; 2 nodes for albacore). The corresponding speedup over serial execution varies between 34 and 90, depending on the case.

The parallel efficiency decreases from $> 90\%$ to $\sim 60\%$ as the number of age groups (and hence workers) increases. This is expected since the amount of work attached to each worker remains the same while the amount of data to send at the end of each time step increases from 7MB to 29MB. This is confirmed in Fig. 5 where the profile shows that MPI communication increases from 7% to 34%, in line with the observed reduction of parallel efficiency.

The number of workers need not match the number of age groups. Figure 6 shows the parallel speedup obtained on NeSI's Milan cores for the larger albacore case as the number of workers increases. Note that the number of age groups $n_a = 151$ is not required to be divisible by the number of workers.

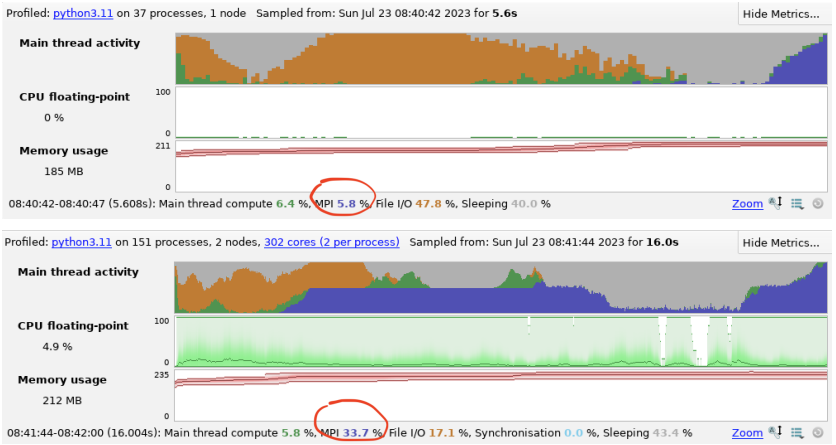

Figure 5: Forge's map profiling information for the 37 (top) and 151 worker cases (bottom). Note how the MPI contribution increases from 6% to 34%.

Here, the number of workers was 2, 10, 16, 32, 76 and 151. All the runs were performed on a single node except for the 151 worker case, which ran on two nodes. This run also exhibits the largest execution time variability (4.8-10.6 seconds). In contrast, running with 76 workers produced reliable timings (9.2-10.3 seconds) and a parallel efficiency $> 88\%$.

In the context of an optimization problem where the simulation parameters are inferred by maximizing the likelihood function, many sequential solves need to be computed to build the likelihood. The likelihood calculation represents an accumulation of advection-diffusion-reaction solves across cohorts and time steps, in other words, it several function evaluations (full model solves) may be required before making a successful iteration step. The gradient descent method currently used does not lend itself to parallelization because of the sequential nature of the algorithm. Nevertheless, additional parallelism can be exploited in the computation of the likelihood, e.g., by looping over fisheries and spatial domain in parallel. Depending on how computationally intensive the gradient descent method is, a worker (MPI rank) can be assigned to the optimizer or, if the gradient descent method is lightweight, one MPI rank (e.g. rank 0) can take over this additional task. This will likely work well if other workers don't have to wait long for the gradient descent step to complete.

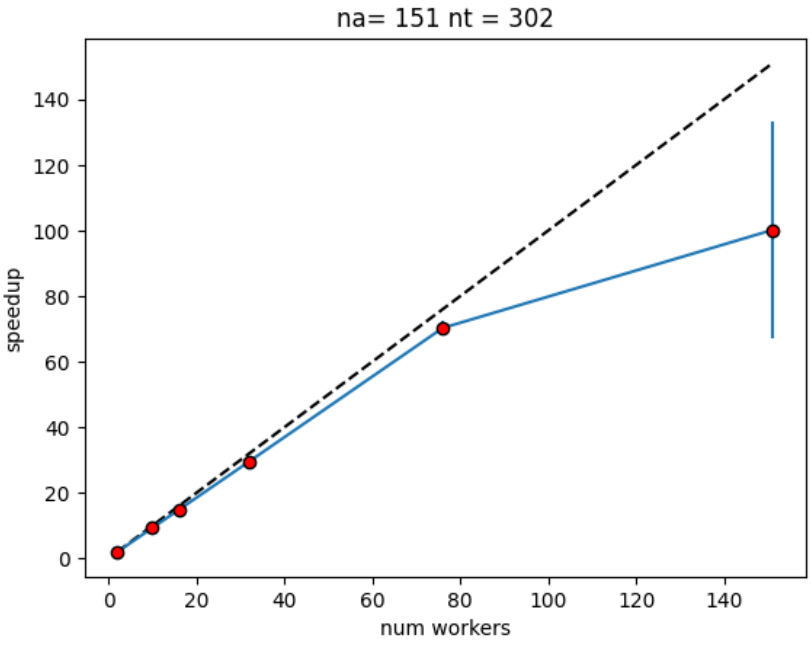

Figure 6: Parallel speedup over a serial run vs number of workers for the albacore ($n_a = 151$ and $n_t = 302$) test case. Each point represents the mean value of 10 runs and error bars the corresponding standard deviation. The ideal scaling is shown as a dashed line.

7 Other performance improvements

7.1 Explicit time stepping for the forward problem

The ADI solver can have a significant impact on the execution time. Here we explore an alternative method, which is explicit in time and does not involve any solve. This would open up the possibility for applying an additional level of parallelism within the time stepping loop (recall that ADI is not readily parallelizable).

The time stepping scheme proposed below is stable provided the time step $\Delta t < \frac{h^2}{2D+uh}$ where D and u are measures of the diffusivity and velocity and h is the (uniform) spatial cell size.

Because the algebra becomes quickly tedious, we flesh out the idea in 1D with the assumption that the approach will work in any number of dimensions. This will need to be checked, however.

Our starting point is

$$\frac{\partial C}{\partial t} + \frac{\partial}{\partial x}(uC) = \frac{\partial}{\partial x}\left(D\frac{\partial C}{\partial x}\right) - MC \tag{1}$$

for cohort concentration C, where u is the velocity, D is the diffusion coefficient and M is the mortality rate. Equation (1) becomes

$$\begin{aligned} C_i(t+\Delta t) &= C_i - \tfrac{\Delta t}{h}\left(u_i C_i - u_{i-1}C_{i-1}\right) \\ &\quad + \tfrac{\Delta t}{h^2}\left(D_{i+\frac{1}{2}}C_{i+1} - (D_{i+\frac{1}{2}} + D_{i-\frac{1}{2}})C_i + D_{i-\frac{1}{2}}C_{i-1}\right) \\ &\quad - \Delta t M C_i \end{aligned} \tag{2}$$

after discretizing C on a uniform grid. All the terms on the right hand side are evaluated at the current time t. The only term at $t + \Delta t$ is on the left hand side. This discretization accounts for spatially varying velocity and diffusivity. Note that the diffusivity D is evaluated at the mid cell locations $(i \pm \frac{1}{2})$. An upwind scheme is applied to the convection term – here we assumed $u > 0$. For $u < 0$ the convection is proportional to $u_{i+1}C_{i+1} - u_i C_i$.

Key here is to determine the amplification factor A from one time step to the next,

$$C_i(t + \Delta t) = AC_i. \tag{3}$$

To proceed we assume u and D to be constant and take a single Fourier mode $C_i \propto e^{ikx_i}$. From (2) we then get

$$A = 1 - \frac{\Delta t u}{h}\left(1 - e^{-ikh}\right) - \frac{4\Delta t D}{h^2}\sin^2(kh/2) - \Delta t M. \tag{4}$$

To achieve numerical stability we need to enforce $|A|^2 < 1$ for all $0 < kh \leq \pi$.

Note that as $kh \to 0$ we get $A \to 1 - \Delta t u i k - \Delta t D k^2 - \Delta t M$, which shows that the discretization is consistent with the operators in the long wavelength limit. The other interesting limit is $kh = \pi$; this corresponds to the highest frequency mode supported by the grid. In general, the most unstable modes have the highest frequency so we can focus $kh = \pi$ for simplicity. In this case

$$|A|^2 = \left(1 - \frac{4\Delta t D}{h^2} - \frac{2\Delta t u}{h} - \Delta t M\right)^2 \quad (kh = \pi) \tag{5}$$

All the individual terms in the squared brackets are positive. Hence the largest, admissible time step would have $\frac{4\Delta t D}{h^2} - \frac{2\Delta t u}{h} - M \approx -2$ since this would give $|A|^2 \approx (1-2)^2$. From this we derive the (approximate) stability criterion

$$\Delta t < \frac{h^2}{2D + uh}. \tag{6}$$

after neglecting h^2 terms in the denominator.

We have shown that a single Fourier mode's amplitude gets multiplied by A at each time step. Given the form of (2) we expect $|A| < 1$ and this leads to criterion (6) for the maximum, admissible time step. This criterion does not account for $O(h^2)$ and $O(\Delta t^2)$ terms, which can arise with spatially varying coefficients.

Note that the diffusion coefficient D and velocity u not only vary spatially but also between cohorts. Accordingly the maximum Δt will also vary. Choosing a different Δt for each cohort will cause load imbalance and we therefore recommend, at least initially, to set the same, uniform Δt using the maximum D and u values across all cohorts.

7.2 Implicit time stepping for the adjoint problem

An additional requirement comes from parameter optimization, a step that involves maximizing the likelihood function. The steepest gradient descent method for determining the "best" parameters from the likelihood function requires the gradient of the likelihood function to be evaluated.

The currently implemented adjoint method in SEAPODYM offers an elegant and numerically efficient approach, where finite differences of the objective/likelihood function are replaced by solving the adjoint problem (see e.g. `https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf`). This can represent a significant computational saving since the adjoint problem only requires one additional advection-diffusion-reaction solve instead of the N_p solves to estimate the gradient of the likelihood function (N_p being the number of parameters). The adjoint problem has the same structure (form) as the forward problem with some notable differences: (1) the source terms

are different and (2) time runs backwards from the last to the initial time. The communication flows in the adjoint problem are also reversed; instead of gathering data at the beginning of a cohort's life the data are scattered.

Care must be taken when solving the adjoint problem as the discretized equations must also be adjoint to the forward discretized problem. This requires the discretized operators to be such that integrating backwards in time recovers the original, initial condition when using the same sources. To recover the original solution after integrating back in time we need the time stepper to produce an amplification factor that is the exact inverse of the forward stepping scheme (i.e. $1/A$). This is achieved by:

- making the time step negative
- using a downwind discretization for the convective term
- and applying a full implicit scheme in time.

Fully implicit time stepping will have the terms proportional to Δt on the right hand side in (2) moved to the left hand side,

$$\begin{aligned} C_i + \tfrac{\Delta t}{h}\left(u_i C_i - u_{i-1} C_{i-1}\right) & \\ -\tfrac{\Delta t}{h^2}\left(D_{i+\frac{1}{2}} C_{i+1} - (D_{i+\frac{1}{2}} + D_{i-\frac{1}{2}}) C_i + D_{i-\frac{1}{2}} C_{i-1}\right) & \\ + \Delta t M C_i &= C_i(t) \end{aligned} \tag{7}$$

where the terms on the left hand side are at time $t - \Delta t$. Compare (2-3) with (7) and we find that (7) can be rewritten as

$$A C_i(t - \Delta t) = C_i. \tag{8}$$

Therefore, each reverse time step divides the solution by the factor A of (4).

Figure 7 shows that it is possible to recover the original solution after integrating the system using the forward discretization (2) and the adjoint discretization (7) with Δt close to the stability limit.

Apart from differences in discretization, the impact of parallelization on the adjoint problem will be on the computation of the source terms and the initial conditions. The source terms involve the adjoint differentials of the likelihood function, which must be distributed across the MPI cohort workers. This would amount to a `MPI_Bcast` operation (`https://www.mpich.org/static/docs/v3.1/www3/MPI_Bcast.html`). In the forward problem, the initial cohort conditions are gathered and accumulated at the start of each cohort whereas in the adjoint problem, the reverse process takes place, i.e. chunks of data are sent to other workers. The latter can be implemented with an `MPI_Scatter` operation (`https://www.mpich.org/static/docs/v3.1/www3/MPI_Scatter.html`).

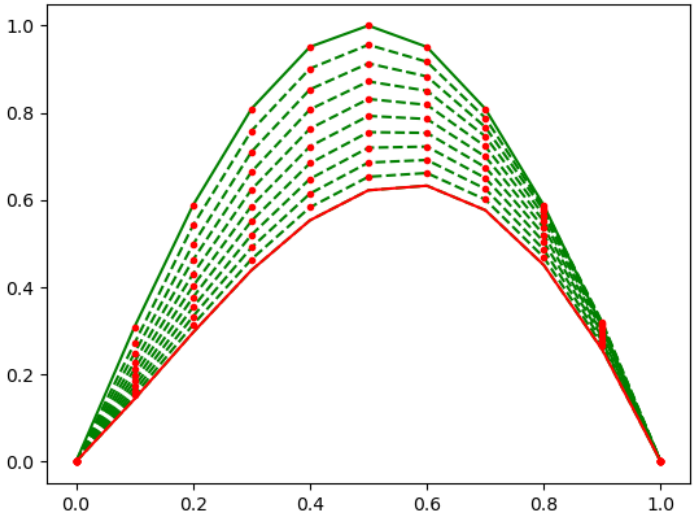

Figure 7: One-dimensional simulation with constant $D = 0.5$, $u = 1$, $M = 0$ and $h = 0.1$. Zero Dirichlet boundary conditions were applied at $x = 0$ and 1. The initial condition is the solid green line. Ten time steps ($\Delta t = \frac{0.9}{2D+uh}$) were applied forward (dashed green lines) and backward (red dotted lines).

8 Summary and future work

Using a toy model, we demonstrated the parallel scaling potential of SEAPODYM. The toy model emulates the time stepping of SEAPODYM with a "sleep" instruction that lasts only 0.015 seconds. After each time step, 24000 double values are remotely accessed by one processing element. Despite the short time steps, a parallel speedup of approximately 80-90x was achieved for the 151 age group setup (albacore) using 76 or 151 processors and single threading. Increasing the time step would reduce the cost of communication compared to the work load. Adding more age groups on the other hand increases the communication costs and hence decreases the parallel efficiency.

The toy model assumes that each time step takes the same amount of time for all cohorts. In reality, access to the memory, IO and other operations could impact scalability as different processors compete for resources.

Further improvements could be achieved by parallelizing within each time step, e.g. through domain decomposition. Explicit time schemes (and some implicit time step schemes as well) are amenable to domain decomposition.

On top of MPI, workers could allocate multiple OpenMP threads and we have shown how some loops can be parallelized with OpenMP pragmas. In addition, vectorization provides yet another avenue for higher performance, often without the need to modify the code as compilers have been shown to be smart enough to recognize most vectorizable loops.

Our recommendation would be to tackle the MPI parallelism over the age cohorts first as this has the potential to deliver the largest speedup. The Python implementation of the toy problem can serve as a guide. It would be fairly straightforward to translate the Python MPI calls to C++. To proceed, we would need to define a bridge between the MPI framework and SEAPODYM, possibly requiring a refactoring of the code to expose the parts that are required by the `Task` object defined in Section 6.1.

Beyond vectorization, multi-threading and distributed computing, we have investigated whether the implicit ADI solves can be replaced with explicit solves. Although these are early investigation, where we focused on the 1D problem for simplicity, it is likely that the replacement of ADI calls by explicit time stepping could further improve the efficiency of SEAPODYM. The derivation of a Courant-like criterion for the advection-diffusion-reaction problem give an estimate for the maximum time step. Furthermore, a domain decomposition could be applied to leverage another level of parallelism. Integrating backwards, however, requires a fully implicit scheme to recover the initial density field. Libraries such as PETSc and others could be used to efficiently solve the sparse linear system in this case.

NeSI has research software engineers who can assist with implementing

the described cohort parallelization approach.