INDEX

Madingley Model - OpenMP documentation

Parallelization of C++ version of Madingley Model using the OpenMP directives.

Developed by **Carlo Cristarella Orestano**.

Working on the following branch RefToPointer, github url below:

https://github.com/phlndrwd/MadingleyCPP/tree/RefToPointer

On the branch: **OpenMP_Version**

1. Configuration NetBeans for OpenMP

Right click on the project name from the **Projects** window on the left and select **Properties**, or select Project Properties from the File menu. When making edits on this dialog, ensure that **<All configurations>** is selected from the **Configuration** drop-down menu. Otherwise you'll need to perform the following instructions twice, both for *Release* and *Debug*. Under the **C++ Compiler** section, edit the **Additional Option** adding: -fopenmp

If it still not working, please follow the directives below:

Select **Tools** on the NetBeans' menu bar and select **Option**. Now you'll need to move on **C/C++** window and select **Build Tools**, click "Add..." button under the **Tool Collection** and the follow dialog will appear. For Base Directory specify the bin location of where OpenMP compiler is located (i.e. /usr/lib or /usr/bin). Select GNU in Tool Collection Family and Enter OpenMP as Tool Collection Name.

You should see the OpenMP in the Tool Collection now and you should be able to run the project with the option: `-fopenmp`

In alternative you should enter project details, right click the project name from the **Projects** window on the left and select **Properties**, and under **Build** select OpenMP from the Tool Collection. Ensure that **<All configurations>** is selected from the **Configuration** drop-down menu.

2. Implementation of the parallel function

Alternatively to the **RunWithinCells( )** function, which runs the model serially, a new function has been added to the **Madingley class** in order to run the model in parallel: **RunWithinCellsInParallel( )**. This function uses the OpenMP directives, for this reason both class and superclass where they are implemented have the link to the **OpenMP library**:

```
#include <omp.h>
```

The parallel function works at **grid cell level** because the **ecological process** spends most of the running time. This decision took place after doing many tests and having some confrontations with developers of the Madingley Model.

Changing the FOR syntax to make it usable by OpenMP

The FOR within **RunWithinCellsInParallel( )** is the one for looping each GridCell and it has been refactored using the more common syntax:

```
for(init; condition; increment;){}
```

In this way the OpenMP directive for the loops can work properly.

```
void Madingley::RunWithinCellsInParallel( )
{
    // Instantiate a class to hold thread locked global diagnostic variables
    // singleThreadDiagnostic-> extinctions, productions, combinations, NextCohortIDThreadLocked = NextCohortID;
    int extinctions = 0, productions = 0, combinations = 0;

    ThreadVariables singleThreadDiagnostics( 0, 0, 0, mNextCohortID );

    #ifdef _OPENMP
    std::cout<<"Running RunWithinCellsInParallel( )..."<<endl;
    double startTimeTest = omp_get_wtime( );
    #endif

    #pragma omp parallel num_threads(omp_get_num_procs()) firstprivate(singleThreadDiagnostics)
    {

        #pragma omp for schedule(dynamic) reduction(+:extinctions), reduction(+:productions), reduction(+:combinations)
        for( unsigned gridCellIndex = 0; gridCellIndex < Parameters::Get( )->GetNumberOfGridCells( ); gridCellIndex++ )
        {

            RunWithinCellStockEcology( mModelGrid.GetACell( gridCellIndex ) );
            RunWithinCellCohortEcology( mModelGrid.GetACell( gridCellIndex ), singleThreadDiagnostics );
            extinctions += singleThreadDiagnostics.mExtinctions;
            productions += singleThreadDiagnostics.mProductions;
            combinations += singleThreadDiagnostics.mCombinations;
            // Update the variable tracking cohort unique IDs
            #pragma omp critical
            {
                mNextCohortID = singleThreadDiagnostics.mNextCohortID;
            }
        }

    }//end parallel region

    // Take the results from the thread local variables and apply to the global diagnostic variables
    mGlobalDiagnosticVariables["NumberOfCohortsExtinct"] = extinctions - combinations;
```

3. OpenMP directives in RunWithinCellsInParallel( )

#pragma omp parallel num_threads(omp_get_num_procs( )):
This **parallel** directive creates a parallel region, where a specified number of threads are created in addition to the master thread for the duration of the next block/statement.
The **num_threads(N)** clause assigns N threads to the parallel region. In this case, using the function **omp_get_num_procs( ),** N will be the number of cores of the current machine.

#pragma omp for schedule(dynamic) [clauses]:
This **for** directive splits the work of the for-loop among the threads.
The clause **schedule(dynamic)** is used for reducing **workload imbalance**, the loop iterations are assigned to the first thread that requires them.

reduction(+:extinctions), … , … :
The **reduction(<operation> : <variable>)** clause is for storing partial results in private variables and combining partial results after the loop, OpenMP will do it automatically for the specified operation and variable.

firstprivate(singleThreadDiagnostics):
The clause **firstprivate(singleThreadDiagnostic)** is a special case of private, it creates a local copy of the variable for each thread as the **private** clause does, and moreover it initializes each private copy with the corresponding value from the master thread.

#pragma omp critical:
The **critical** directive is used to execute part of a code a thread at a time (sequentially).
It is used to protect the access to the shared variable **mNextCohortID**.

-**omp_get_wtime( ):**
OpenMP runtime library function to get timings.

4. OpenMP directives in Cohort and GridCell classes and added ResetRandom( ) in CohorthMerger

Have been made the following steps to handle these static variables.

-**Moving the vector mNewCohorts in the class GridCell.**

Making the static variables used within the parallel region local for each Thread:

-**mMassAccounting** and **mNextID** in Cohort class.

-**mNewCohorts** in GridCell class.

#pragma omp threadprivate(…):

The **threadprivate** directive is used to make global/static data local for each thread through the execution of parallel regions.

Cohorth:

```
static Types::Double2DMap mMassAccounting;
static unsigned mNextID;
#pragma omp threadprivate(mMassAccounting,mNextID)
```

Gridcell:

```
static std::vector<Cohort*> mNewCohorts;
#pragma omp threadprivate(mNewCohorts)
```

Reset the random cohort sequence in RunWithinCellCohortEcology

This is one of the randomized processes within the parallel region, if it's compared the parallel output with the sequential one, it's possible to see that these processes alterate slightly the result.

This means that these differences are correct anyway, so this random processes could be modified if we want same outputs.

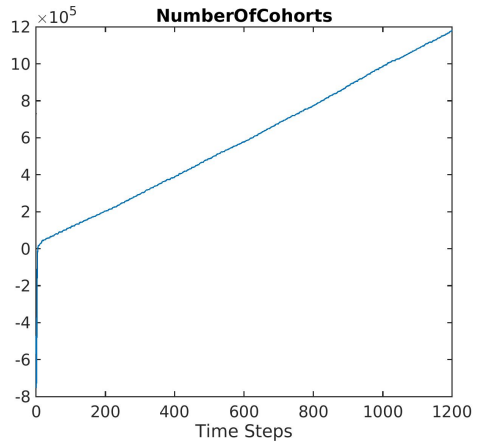
The new function ResetRandom( ) resets the mRandomNumber, it is used when it's needed to merge cohorts. Accordingly it will have the same Random Sequence everytime because it will have the same Seed.

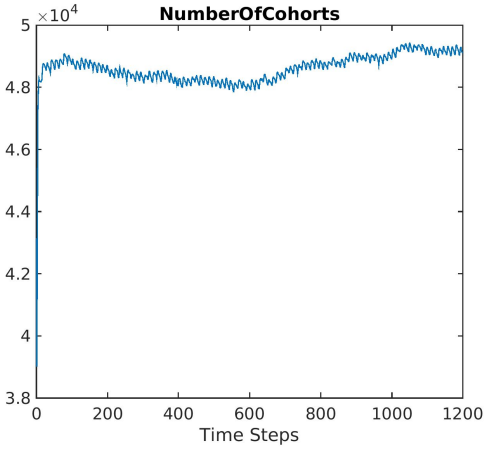
Other randomized processes present in the model could affect the output.

5. New Changes in RunWithinCellsInParallel( )

After running the model a couple of times, an anomaly showed up in the NumberOfCohorts chart, giving an initial negative value. Important changes have been made after that.

The diagnostics variables were updated incorrectly using the reductions, they were added up within each thread instead of updating and merging them at the end of parallel region.

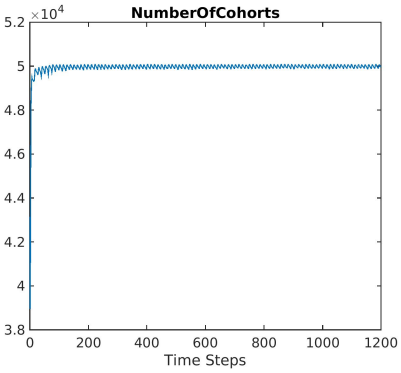




On the left there's the chart of the old parallel version; at the beginning the number of Cohorts is negative and thereafter there is a continuous increase.

On the right, instead, there's the same chart after the changes; the issue is fixed, the trend of Cohorts is always positive and remains constant.

Furthermore the last NumberOfCohorts chart is more similar to the **sequential** one:



The trend is more constant because the ecological process is running sequentially, with parallelism is more oscillating because different threads are working in the same time, this may due to the fact that threads increment or reduce the number concurrently in different GridCells.

Final version of the parallel function

The issues have been solved deleting the reductions and firstprivate directives and putting a list to handle the diagnostic variables.

A list of **ThreadVariables** called **partialsDisgnostics** has been created, each ThreadVariables contains extinctions, productions, combinations, nextCohortID, **which are the variables that need multi-thread protection**.

The list **partialDiagnostics is shared** between all threads and each element represents a ThreadVariables of a different GridCell.

shared(partialDiagnostics):

The list exists in only one memory location and all threads access the same address.

At the end of the parallel region all the values within the list are merged in the ThreadVariables variable called **globalDiagnostics**.

It will be used to update correctly the values in **mGlobalDiagnosticVariables.**

```
void Madingley::RunWithinCellsInParallel( ) {
    // Instantiate a class to hold thread locked global diagnostic variables

    #ifdef _OPENMP
    std::cout<<"Running RunWithinCellsInParallel( )..."<<endl;
    double startTimeTest = omp_get_wtime( );
    #endif

    list<ThreadVariables> partialsDiagnostics;

    #pragma omp parallel num_threads(omp_get_num_procs()) shared(partialsDiagnostics)
    {
        ThreadVariables singleThreadDiagnostics( 0, 0, 0, mNextCohortID );

        #pragma omp for schedule(dynamic)
        for( unsigned gridCellIndex = 0; gridCellIndex < Parameters::Get( )->GetNumberOfGridCells( ); gridCellIndex++ )
        {
            RunWithinCellStockEcology( mModelGrid.GetACell( gridCellIndex ));
            RunWithinCellCohortEcology( mModelGrid.GetACell( gridCellIndex ), singleThreadDiagnostics );
        }
        partialsDiagnostics.push_back(singleThreadDiagnostics);
    }//END PARALLEL REGION

    ThreadVariables globalDiagnostics( 0, 0, 0, mNextCohortID);
    for (list<ThreadVariables>::iterator it=partialsDiagnostics.begin(); it != partialsDiagnostics.end(); it++)
    {
        ThreadVariables tmp=*it;
        globalDiagnostics.mProductions+=tmp.mProductions;
        globalDiagnostics.mExtinctions+=tmp.mExtinctions;
        globalDiagnostics.mCombinations+=tmp.mCombinations;
    }

    // Update the variable tracking cohort unique IDs
    mNextCohortID = globalDiagnostics.mNextCohortID;

    // Take the results from the thread local variables and apply to the global diagnostic variables
    mGlobalDiagnosticVariables["NumberOfCohortsExtinct"] = globalDiagnostics.mExtinctions - globalDiagnostics.mCombinations;
    mGlobalDiagnosticVariables["NumberOfCohortsProduced"] = globalDiagnostics.mProductions;
    mGlobalDiagnosticVariables["NumberOfCohortsInModel"] = mGlobalDiagnosticVariables["NumberOfCohortsInModel"] + globalDiagnostics.mProductions - globalDi
    mGlobalDiagnosticVariables["NumberOfCohortsCombined"] = globalDiagnostics.mCombinations;

    #ifdef _OPENMP
    double endTimeTest = omp_get_wtime( );
    std::cout << "RunWithinCellsInParallel( ) took: " << endTimeTest - startTimeTest << endl;
    #endif

}
```

6. Speed-up and efficiency calculation with related graphs

It has also been calculated the overall speedup and efficiency of the Madingley Model running in parallel.

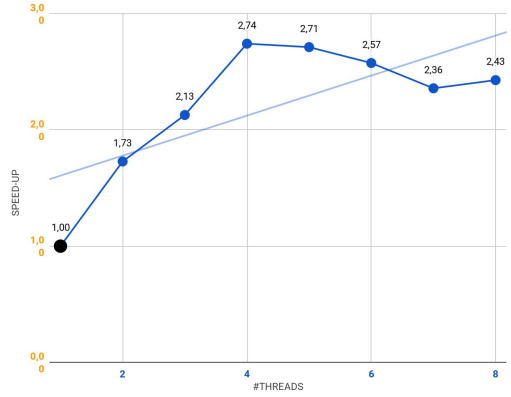
The speedup is based on the average of several speed tests, for several number of threads, done in a machine with the following specifications:

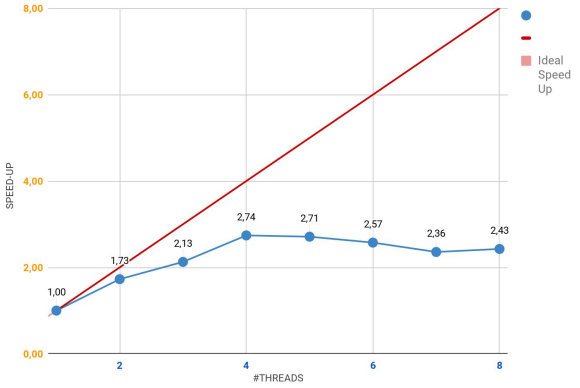
`AMD Quad-Core Processor A-10-7300 with Turbo CORE Technology up to 3.20 GHz`

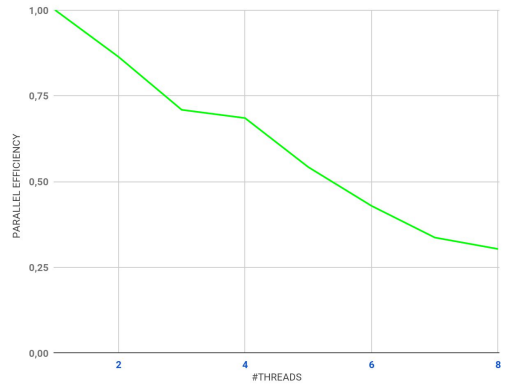
(the processor can manage at most 4 threads at a time)

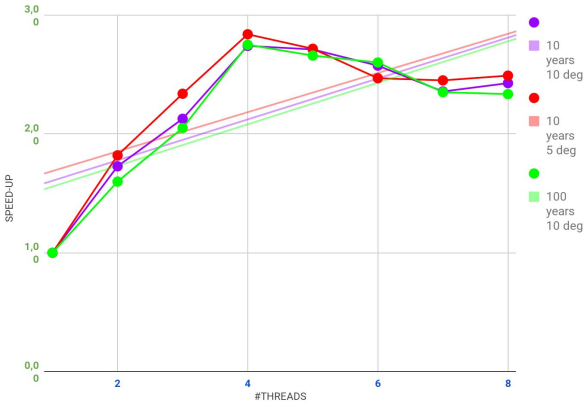
Speedup is defined as the ratio between the runtime of the sequential version (T_1) and the runtime of the parallel version with *p* processors (T_p): $S_p = T_1 / T_p$

Efficiency (E_p) is defined as the ratio of speedup between *p* processors (S_p) and the number of processors *p*: $E_p = S_p / p$







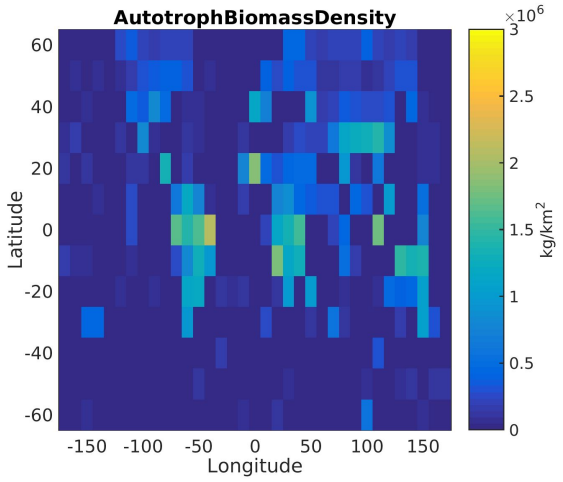


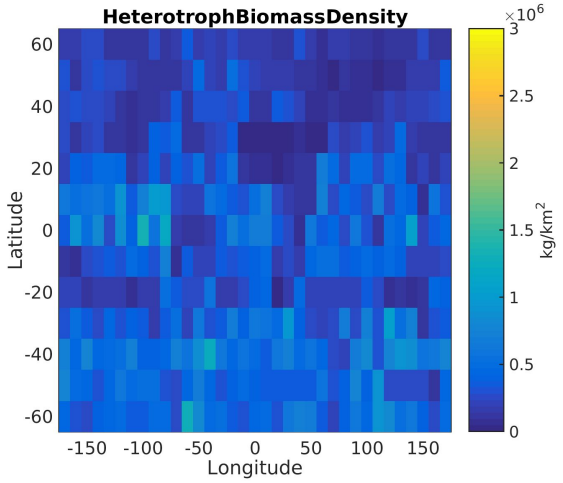
As expected, running the model with the same number of threads as the number of machine's cores give the best results.

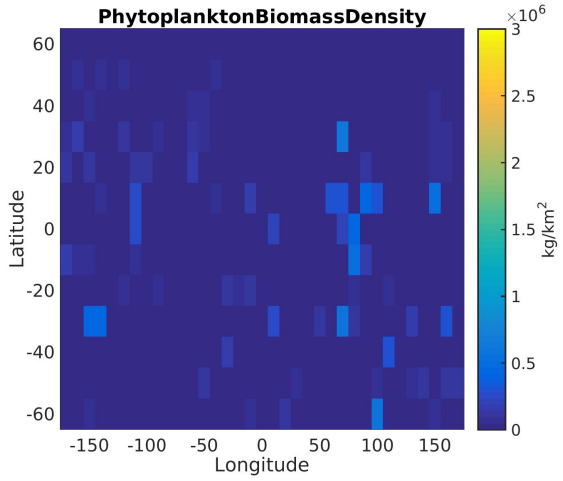
In the last graph it's possible to see that changing the input degree or simulation years doesn't bring any effects, using 4 threads remains the most efficient solution.

7. Comparison of sequential and parallel version output

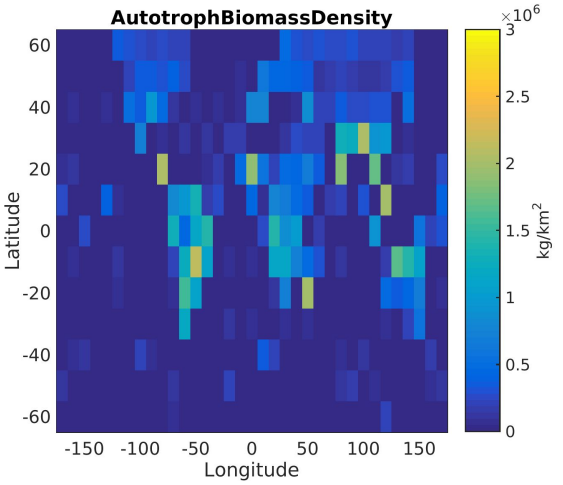
PARALLEL VERSION

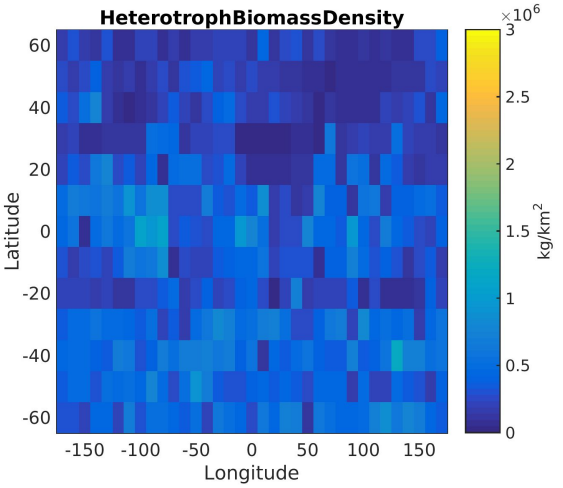


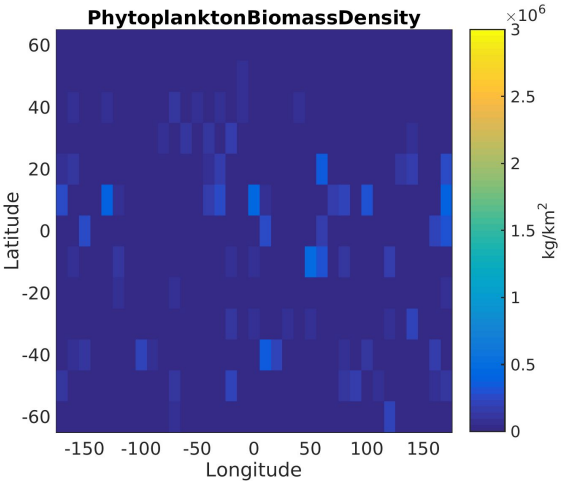




SEQUENTIAL VERSION

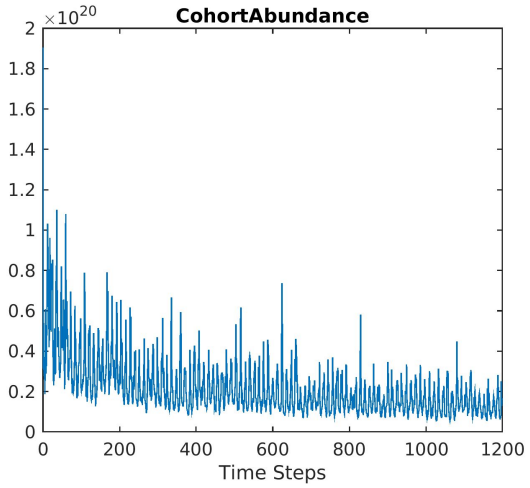


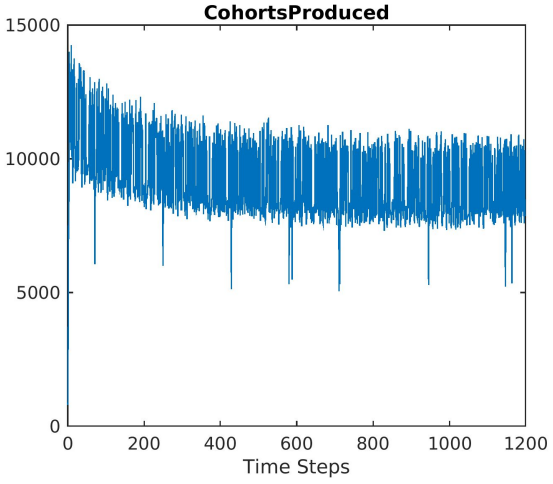


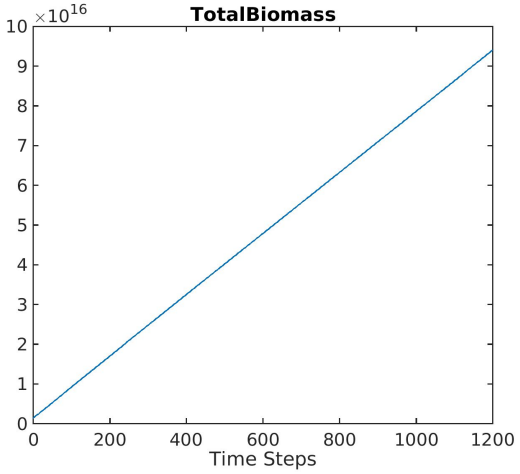


7. Comparison of sequential and parallel version output

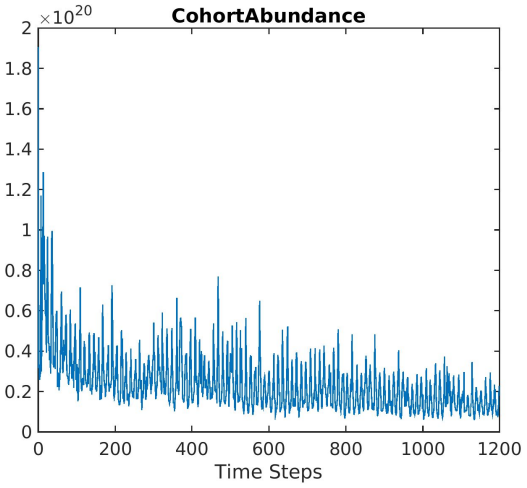
PARALLEL VERSION

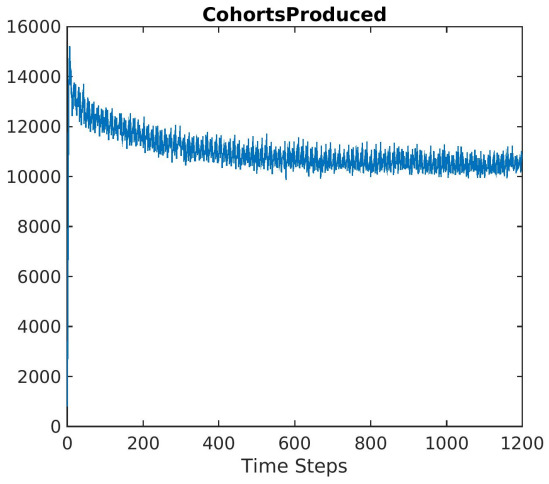


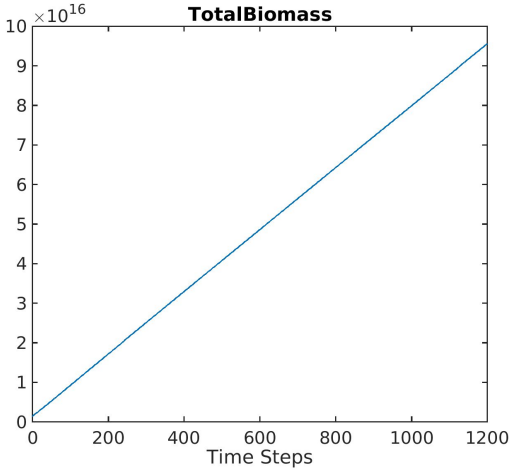




SEQUENTIAL VERSION







Conclusion

Over the last months, the C++ version of the Madingley Model has been refactored using the OpenMP library to allow parallelisation on multiple threads in a single machine and with the goal to increase the overall running speed of the model.
A new function for running the model in parallel has been implemented, keeping also the old serial version of it, so to allow the model to be used both ways.
Comparing the parallel function with the serial version it is possible to see how the use of threads cuts the running time considerably, depending also on the machine you are working with.
The ecological processes took most of the time, its computational speed has been improved with multi-threading approach.

As seen in paragraph 6 the tests of the parallel running have been made on a Quad-Core computer with the following results:

Speed-Up: Serial/Parallel = $S_1/P_4 \cong$ **2,7**

It would be interesting to test also against different machines with more cores and computational power.
Some differences have been shown between the Parallel and Serial outputs, but overall it looks like the parallelisation is following the right path.

The OpenMP library makes possible to run the model in parallel on a single machine by using multiple threads; for future developments it would be interesting to implement an MPI version of the model, making it possible to parallelise it across multiple machines instead of just using one.

Bibliography

1. **Introduction to Parallel Computing**.
   URL https://computing.llnl.gov/tutorials/parallel_comp/

2. **OpenMP, Lawrence Livermore National Laboratory**.
   URL https://computing.llnl.gov/tutorials/openMP/

3. **Official OpenMP website**. URL http://openmp.org/

4. **Rohit Chandra et al**. Parallel Programming in OpenMP, 2000.

5. **Michael J. Quinn**. Parallel Programming in C (with OpenMPI and OpenMP), 2003.

6. **OpenMP, Wikipedia**. URL http://en.wikipedia.org/wiki/OpenMP

7. **GCC Compiler Documentation**. URL http://gcc.gnu.org/onlinedocs/

8. **Corso di Laurea in Informatica, Università della Calabria**.
   URL https://www.mat.unical.it/informatica.