# Development of a multiscale simulation approach for forced migration

Derek Groen[1]

Brunel University London, Kingston Lane, UB8 3PH,
Derek.Groen@brunel.ac.uk,
WWW home page: http://people.brunel.ac.uk/~csstddg/

**Abstract.** In this work I reflect on the development of a multiscale simulation approach for forced migration, and present two prototypes which extend the existing Flee agent-based modelling code. These include one extension for parallelizing Flee and one for multiscale coupling. I provide an overview of both extensions and present performance and scalability results of these implementations in a desktop environment.

**Keywords:** multiscale simulation, refugee movements, agent-based modelling, parallel computing, multiscale computing

## 1 Introduction

In recent years, more and more people have been forcibly displaced from their homes [1], with the number spiraling to over 65 million in 2017. The causes of these displacements are wide-ranging, and can include armed conflict, environmental disasters, or severe economic circumstances [2]. Computational models have been used extensively to study forced migration (e.g., [3, 4]), and in particular agent-based modelling has been increasingly applied to provide insights into these processes [5–7]. These insights are important because they could be used to aid the allocation of humanitarian resources or to estimate the effects of policy decisions such as border closures [8].

We have previously presented a simulation development approach to predict the destinations of refugees moving away from armed conflict [9]. The simulations developed using this approach rely on the publicly available Flee agent-based modelling code (www.github.com/djgroen/flee-release), and have been shown to predict 75% of the refugee destinations correctly in three recent conflicts in Africa [9].

An important limitation of our existing approach is the inability to predict *how many* refugees emerge from a given conflict event at a given location. In a preliminary study, we approached this problem from a data science perspective with limited success [10], and as a result we are now exploring the use of simulation. As part of this broader effort, I have adapted the Flee code to enable (a) the parallel execution for superior performance, and (b) the coupling to additional models. The latter aspect is essential as it allows us to connect simulations of

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93701-4_69

smaller scale population movements, e.g. of people escaping a city of conflict, with simulations of larger scale population movements, e.g. refugee movements nationwide.

In this work, I present the established prototypes to enable parallel, multiscale simulations of forced migration in this context. In Section 2 I discuss the effort on parallelizing Flee, and in Section 3 the effort on creating a coupling interface for multiscale modelling. In Section 4 I present some preliminary performance results, and in Section 5 I reflect on the current progress and its wider implications.

## 2 Prototype I: A parallelized Flee

As a first step, I have implemented a parallelized prototype version of the Flee kernel, which is described in detail by Suleimenova et al. [9]. The Flee code is a fairly basic agent-based modelling kernel written in Python 3, and our parallel version relies on the MPI4Py module. In this prototype version, I prioritized simplicity over scalability, and seek to investigate how far I can scale the code, while retaining a simple code base. Overall, the whole parallel implementation is contained within a single file (pflee.py) which extends the base Flee classes and contains less than 300 lines of code at time of writing.

### 2.1 Parallelization approach

Within this Flee prototype I chose to parallelize by distributing the agents across processes in equal amounts, regardless of their location. The base function to accomplish this is very simplistic:

```
def addAgent(self, location):
    self.total_agents += 1
    if self.total_agents % self.mpi.size == self.mpi.rank:
        self.agents.append(Person(location))
```

Here, the total number or processes is given by self.mpi.size, and the rank of the current process by self.mpi.rank. I can instantly identify on which process a given agent resides, by using the agent index in conjunction with the "% self.mpi.size" operator.

Compared to existing spatial decomposition approaches (e.g., as used in RePast HPC [11]), our approach has the advantage that both tracking the agents and balancing the computational load is more straightforward. However, it has major disadvantages in that it currently does not support directly interacting agents (agents only interact indirectly through modifying location properties). Adding such interactions would require additional collective communications in the simulation. In the case of Flee, this limitation is not an issue, but it can become a bottleneck for codes with more extensive agent rule sets. Additionally, a limitation of this approach is that the location graph needs to be duplicated across each process, which can become a memory bottleneck for extremely large location graphs.

### 2.2 Parallel evolution of the system

The evolve() algorithm, which propagates the system by one time step is structured as follows (functions specific to the parallel implementation are italicized):

1. Update location scores (which determine the attractiveness of locations to agents).
2. Evolve all agents on local process.
3. *Aggregate Agent totals across processes.*
4. Complete the travel, for agents that have not done so already.
5. *Aggregate Agent totals across processes.*
6. Increment simulated time counter.

One requires two *MPI_AllGather()* operations per iteration loop. Our existing refugee simulations currently require 300–1000 iterations per simulation, which would result in 600–2000 AllGather operations. As these operations require all processes to synchronize, I would expect them to become a bottleneck at very large core counts.

## 3 Prototype II: A multiscale Flee model

As a second step, I have implemented a multiscale prototype version of the Flee kernel. In this prototype version, I again prioritized simplicity over scalability. Overall, our multiscale implementation is contained within a single file (coupling.py) which accompanies the base flee classes (serial or parallel, depending on the user preference). The multiscale implementation contains less than 200 lines of code at time of writing.

In the multiscale application, individual locations in the location graph are registered as *coupled locations*. Any agents arriving at these locations in the microscale model will then be passed on to the macroscale model using the coupling interface. The coupling interval is set to 1:1 for purposes of the performance tests performed here (to ease the comparison with single scale performance results), but it is possible to perform multiple iterations in the microscale submodel for each iteration in the macroscale submodel by changing the coupling interval value. This would then result not only in different spatial scales, but also differing time scales. In the prototype implementation, the coupling is performed using file transfers, where at each time step both models write their agents to file and read the files of the other model for incoming agents. As a result, two-way coupling is possible, and both models are run concurrently during the simulation.

In our implementation, the coupling interface is set up as follows:

```
c = coupling.CouplingInterface(e)
c.setCouplingFilenames("in","out")
if(submodel_id > 0):
    c.setCouplingFilenames("out","in")
```

And the coupled locations are registered using a c.addCoupledLocation(), which is called once for each location to be coupled. During the main execution loop, after all other computations have been performed, the coupling activities are initiated using the function `c.Couple(t)`, where $t$ is the current simulated time in days.

## 4 Tests and results

In this section I present results from two sets of performance tests, one to determine the speedup of the parallel implementation, and one to test the speedup of the multiscale implementation. All tests were performed on a desktop machine with an Intel i5-4590 processor with 4 physical cores and no hyper-threading technology.

For our tests, I used a simplified location graph, presented in Fig. 4. Note that the size of the location graph only has a limited effect on the computational cost overall, as agents are only aware of locations that are directly connected to their current location.

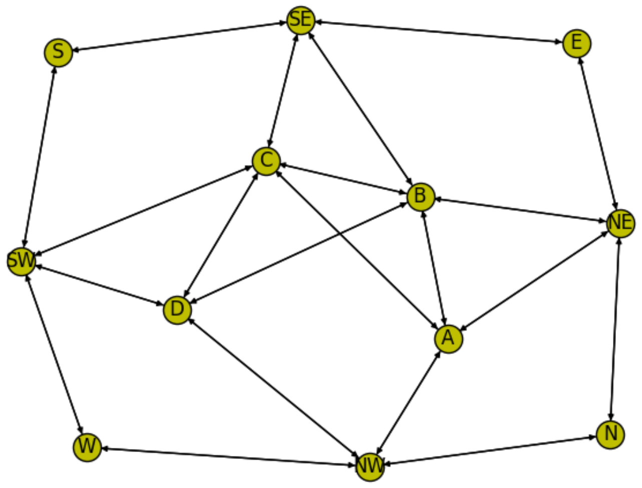

**Fig. 1.** Location graph of the microscale agent-based model. The location graph of the macroscale agent-based model has a similar level of complexity. This graph was visualized automatically using the Python-based `networkx` package.

## 4.1 Parallel performance tests

In these tests I run a single instance of Flee on the desktop using 1,2 or 4 processes. I measured the time to completion for the whole simulation using 10000 agents, 100000 agents and one million agents, and present the corresponding results in Tab. 4.1. Based on these measurements, Flee is able to obtain a speedup between 2.53 and 3.44 for $p = 4$, depending on the problem size. This indicates that the chosen method of parallelization delivers a quicker time to completion, despite its simplistic nature. However, it is likely that the slow single-core performance of Python codes result in apparent better scaling performance when such codes are parallelized. Consequently, I would expect the obtained speedup to be somewhat lower if this exact strategy were to be applied to a C or Fortran-based implementation of Flee. Given the low temporal density of communications per time step (time steps complete in $> 0.13s$ wall-clock time in our run, during which only two communications take place), it is unlikely that the scalability would be significantly reduced if these tests were to be performed across two interconnected nodes.

**Table 1.** Scalability results from the Flee prototype. All runs were performed for 10 time steps (production runs typically require 300–1000 time steps). Runs using 8 processes on 4 physical cores did not deliver any additional speedup.

| agents | processes ($p$) | time to completion | speedup |
|---|---|---|---|
| # of | # of | [s] | |
| 10000 | 1 | 3.325 | 1.0 |
| 10000 | 2 | 1.770 | 1.88 |
| 10000 | 4 | 1.315 | 2.53 |
| 100000 | 1 | 29.26 | 1.0 |
| 100000 | 2 | 14.63 | 2.0 |
| 100000 | 4 | 8.896 | 3.29 |
| 1000000 | 1 | 277.1 | 1.0 |
| 1000000 | 2 | 142.7 | 1.94 |
| 1000000 | 4 | 80.58 | 3.44 |

## 4.2 Multiscale performance tests

In these tests I run two coupled instances of Flee on the desktop using 1,2 or 4 processes each. Runs using 4 processes each feature 2 processes per physical core. I measured the time to completion for the whole simulation using 10000 agents, 100000 agents and one million agents, which were inserted in the microscale simulation, but gradually migrated to the macroscale simulation using the coupling interface.

I present the results from the multiscale performance tests in Tab. 2. Here the multiscale simulations scale up excellently from 1+1 to 2+2 processes, given

**Table 2.** Multiscale performance results using two Flee prototype instances. All runs were performed for 10 time steps (production runs typically require 300–1000 time steps). Note: runs using 4+4 processes were performed using only 4 physical cores.

| agents<br># of | processes ($p$)<br># of | time to completion<br>[s] | speedup |
|---|---|---|---|
| 10000 | 1+1 | 4.016 | 1.0 |
| 10000 | 2+2 | 2.436 | 1.65 |
| 10000 | 4+4* | 2.241 | 1.79 |
| 100000 | 1+1 | 31.08 | 1.0 |
| 100000 | 2+2 | 16.17 | 1.92 |
| 100000 | 4+4* | 14.07 | 2.21 |
| 1000000 | 1+1 | 326.7 | 1.0 |
| 1000000 | 2+2 | 161.4 | 2.02 |
| 1000000 | 4+4* | 112.8 | 2.90 |

that the model contains at least 100000 agents. Further speedup can be obtained by mapping 8 processes (4+4) to the 4 physical cores (i.e. 2 threads per core), leading to a speedup of 2.9 for coupled models with 1000000 agents in total. This additional scaling is surprising because the cores do not support hyper-threading themselves, but could indicate that individual processes can frequently run at high efficiency even when less than 100% of the CPU capacity is available.

Given that both the single scale and multiscale simulations have the same number of agents in the system, it is clear that the multiscale coupling introduces additional overhead. This is because multiscale simulations rely on two Flee instances to execute, and because file synchronization (reading and writing to the local file system) is performed at every time step between the instances. It is possible to estimate the total multiscale overhead by comparing the fastest single scale simulation for each problem size with the fastest multiscale simulation for each problem size. In doing so, I find that the overhead is smaller for larger problem sizes, ranging from 70% (2.241 vs 1.315) for simulations with 10000 agents to 40% (112.8 vs 80.58) for those with 100000 agents.

## 5 Discussion

In this work I have presented two prototype extensions to the Flee code, to enable respectively parallel execution and multiscale coupling. The parallel implementation delivers reasonable speedup when using a single node, but is likely to require further effort in order to make Flee scale efficiently on larger clusters and supercomputers. However, uncertainty quantification and sensitivity analysis are essential in agent-based models, and even basic production runs require 100s of instances to cover the essential areas for sensitivity analysis. As such, even a modestly effective parallel implementation can enable a range of Flee replicas to efficiently use large computational resources. The multiscale coupling interface enables users to combine two Flee simulations (and theoretically more than two), using one to resolve small scale population movements, and one to resolve large scale movements. Through the use of a plain text file format (.csv), it also becomes possible to couple Flee to other models. However, this implementation is still in its infancy, as the coupling overhead is relatively large (40-70%) and the range of coupling methods very limited (file exchange only). Indeed, the aim now will be to integrate the Flee coupling with more mature coupling software such as MUSCLE2 [12], to enable more flexible and scalable multiscale simulations, using supercomputers and other large computational resources.

A last observation is in regards to the development time required to create these extensions. Using MPI4Py, I found that both the parallel implementation and the coupling interface took very little time to implement. In total, I spent less than 40 person hours of development effort.

## Acknowledgements

I am grateful to Robin Richardson from UCL for his comments on the draft of this manuscript. This work was performed within the wider context of the EU H2020 project "Computing Patterns for High Performance Multiscale Computing" (ComPat, grant no. 671564).

## References

1. UNHCR: Figures at a glance. United Nations High Commissioner for Refugees. Available at: http://www.unhcr.org/uk/figures-at-a-glance.html (2017)
2. Moore, W.H., Shellman, S.M.: Whither will they go? A global study of refugees destinations, 1965-1995. International Studies Quarterly **51**(4) (2007) 811–834
3. Willekens, F. In: Migration flows: Measurement, analysis and modeling. Springer Netherlands, Dordrecht (2016) 225–241
4. Shellman, S.M., Stewart, B.M.: Predicting risk factors associated with forced migration: An early warning model of Haitian flight. Civil Wars **9**(2) (2007) 174–199
5. Kniveton, D., Smith, C., Wood, S.: Agent-based model simulations of future changes in migration flows for Burkina Faso. Global Environmental Change **21** (2011) 34–40
6. Johnson, R.T., Lampe, T.A., Seichter, S.: Calibration of an agent-based simulation model depicting a refugee camp scenario. In: Proceedings of the 2009 Winter Simulation Conference (WSC). (2009) 1778–1786
7. Sokolowski, J.A., Banks, C.M.: A methodology for environment and agent development to model population displacement. In: Proceedings of the 2014 Symposium on Agent Directed Simulation. (2014)
8. Groen, D.: Simulating refugee movements: Where would you go? Procedia Computer Science **80** (2016) 2251–2255
9. Suleimenova, D., Bell, D., Groen, D.: A generalized simulation development approach for predicting refugee destinations. Scientific Reports **7:13377** (2017)
10. Chan, N.T., Suleimenova, D., Bell, D., Groen, D.: Modelling refugees escaping violent events: A feasibility study from an input data perspective. Proceedings of the Operational Research Society Simulation Workshop (SW18) (in press) (2018)

11. Collier, N., North, M.: Repast hpc: A platform for large-scale agentbased modeling. Large-Scale Computing Techniques for Complex System Simulations (2011) 81–110
12. Borgdorff, J., Mamonski, M., Bosak, B., Kurowski, K., Belgacem, M.B., Chopard, B., Groen, D., Coveney, P., Hoekstra, A.: Distributed multiscale computing with muscle 2, the multiscale coupling library and environment. Journal of Computational Science **5**(5) (2014) 719 – 731