universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Benchmarking a parallelized agent-based simulation in economics

verfasst von / submitted by

Mag.rer.soc.oec. Oliver Reiter, Bakk.rer.soc.oec, Bakk.rer.soc.oec

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna, 2018

Studienkennzahl lt. Studienblatt/ degree programme code as it appears on the student record sheet:	A 066 940
Studienrichtung lt. Studienblatt/ degree programme as it appears on the student record sheet:	Scientific Computing UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, M.Sc.

I, Oliver Reiter, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Agent-based simulations are gaining popularity in the social sciences, also in economics. As such simulation models grow in size and features, there is a growing need that these simulations are executed in parallel.

We will study the performance behaviour of a parallelized agent-based model, the so called "benchmark" model by Caiani et al. (2016), implemented in the FLAME framework. The parallel performance of the model is tested on a commodity hardware laptop, a multicore machine and the Vienna Scientific Cluster[1].

We find that the performance of the simulation is a) poor, especially when MFLOPS per second are used as benchmark but also that b) super-linear speedups are possible. We trace both these finding back to an inefficient design of how the agent's data is read and stored. An exaggerated amount of data is transferred to the CPU, which makes the application I/O bound and leads to the observed poor performance. Furthermore, when the simulation is executed on more processes, data transfers are better distributed leading to the super-linear speedups.

[1]The simulation results presented here have been achieved in part using the Vienna Scientific Cluster (VSC).

Zusammenfassung

Agenten basierte Simulationen erfreuen sich steigender Beliebtheit in the Sozialwissenschaften. Auch in der Ökonomie sind sie auf dem Vormarsch. Das Wachsen der Modelle in Größe und Umfang macht es notwendig, die Simulationen in parallel und möglichst effizient auszuführen.

Wir werden die Performance Eigenschaften des sogenannten "Benchmark" Modells von Caiani et al. (2016) untersuchen, welches mithilfe des FLAME frameworks implementiert wurde. Wir testen die Performance auf drei verschiedenen Maschinen: auf einen gewöhnlichen Laptop, auf eine Multicore Maschine und am Vienna Scientific Cluster[2].

Die Resultate zeigen, dass die Performance der Simulation einerseits schlecht ist, speziell wenn die erreichten MFLOPS pro Sekunde herangezogen werden dass aber andererseits auch super-lineare Speedups möglich sind. Unsere Untersuchungen zeigen, dass sich beide Ergebnisse auf ein ineffizientes Design auf die Art und Weise wie Daten der Agenten geladen und gespeichert werden zurückführen lassen. Da eine stark überhöhte Menge an Daten transferiert wird, ist diese Applikation I/O bound was zur enttäuschenden Performance führt. Wenn dieser Datentransfer dann auf mehrere Prozesse aufgeteilt wird, kommt es zu den super-linearen Speedups.

[2]The simulation results presented here have been achieved in part using the Vienna Scientific Cluster (VSC).

Acknowledgements

I would like to thank my supervisor Prof. Wilfried Gansterer for his support and guidance throughout creating this thesis.

My girlfriend Gerlinde deserves my gratefulness for her encouragement and support to finish my studies even when this meant putting aside her own interests.

My sincere thanks also go out to my sister Karina for reading and correcting this thesis.

Furthermore, I would like to thank my family and friends for their support during my studies.

Table of Contents

Chapter 1

Introduction

1.1 Problem setting

Since the economic crisis in 2008 and the failure of the economic profession to warn about the turmoils to come, economic theory is orientating itself anew. Renowned economists are calling for new approaches and techniques. Agent-based models (ABM) are one of those new methods that have been gaining traction recently. Caiani et al. (2016) propose a "agent-based stock-flow consistent macroeconomic" benchmark model in this area of economic research.

Since agent-based modelling is computationally intensive it is a natural field of investigation for computer scientists. As such model grow in size, i.e. as the modelled population increases, and in features, e.g. through more realistic modelling of the agent's behaviour, performance considerations are becoming more and more important.

However, performance analyses for this class of models are still scarce. With this thesis I try to fill this gap and implement a parallelized version of the benchmark model of Caiani et al. (2016) and carry out performance measurements on three different machines: a commodity hardware laptop, a multicore machine and computer cluster.

1.2 Motivation

My motivation to undertake this research is twofold. First, from the perspective of computer science, agent-based models are a form of simulation and simulations are considered to be a " 'third pillar' of scientific inquiry"[1].

[1]See Reed et al. (2005), p. 1.

Furthermore, agent-based models are nowadays used in all sciences. Computational social science is one new research area that uses simulations and agent-based models heavily. I want to contribute to this literature by providing a thorough performance analysis of a seminal model in agent-based economics. Additionally, I want to use the opportunity and experiment with that model on an Austrian supercomputer, the Vienna Scientific Cluster (VSC).

Secondly, as an economist I am interested trying out new models and methods that can help us to better understand the causes of, e.g., the financial crisis in 2008. Agent-based models show promising properties for this endeavour.

1.3 Synopsis

The upcoming chapter 2 reviews the related work. It begins by emphasizing the role of scientific simulations, especially agent-based models. Then the link from agent-based models to object-oriented programming to parallelization is described and two prominent frameworks for implementing agent-based models are presented.

The next section of this chapter contains additional topics in parallelization of simulations, such as load balancing, shared memory multithreading and applications on GPGPU and supercomputers.

The ensuing section shifts the focus to economics. It surveys the shortcomings of traditional macroeconomic models, especially in light of the failure of economists to forecast the financial crisis and proposes agent-based stock-flow consistent models as a possible remedy.

The chapter closes with a summary of results of my preparational research. I implemented a simple agent-based economic model in two frameworks (FLAME and Repast) and compared their runtime behaviour for a range of problem sizes and parameters. FLAME showed great potential as it outperformed Repast in all settings.

Chapter 3 describes the investigated “benchmark model” in greater detail. It also gives an overview of the implementation and parallelization of the model. How the initial state is calibrated so that it obeys the requirements of stock-flow consistency is presented in section 3.2.1.

The first part of chapter 4 describes the hardware of the machines that are used in the performance tests. For each of the three machines (a laptop, a server at the University of Vienna called “Lewis” and the Vienna Scientific Cluster), the system’s topology, its theoretical peak performances and the compiler and MPI library that are used to compile the code are presented.

The next part of the chapter explains how the experiments were conducted and which metrics were used to characterize the performance behaviour of the parallel application.

The remainder of chapter 4 presents the findings of the performance tests. It consists again of three parts. The first part is about the results on the laptop. It shows that even for small problem sizes, the parallel FLAME code is able to achieve a considerable speedup compared to the serial code.

The results on Lewis are inconclusive: While we see large speedups, we also find falling MFLOPS/s with increasing problem size. The experiments show further that cache misses occur proportionally more in simulation runs with a low number of processes. The performance metrics show efficiency values (sometimes considerably) larger than one, meaning we see super-linear speedup.

All in all are the results on the VSC similar to the results from Lewis. Again, there is the occurrence of super-linear speedup, though at a lower level. The achieved MFLOPS/s are disappointing in light of how many GFLOPS/s would have been possible. Additional tests on the VSC show that a) the parallel FLAME code weakly scales almost linearly, b) hyperthreading does not yield any big additional performance improvements and c) load imbalance can be quite high, especially in small problem sizes and a high number of processes.

The rest of the chapter tries to explain the results. The main driver behind the results can be found in an inefficient memory design which leads to exaggerated amount of data being transferred between memory and CPU. This leads to a) the observed poor performance as the CPU is constantly waiting for data retrievals to complete and b) super-linear speedups when the data fetching can be distributed to separate hardware.

Chapter 2

Related work

My master thesis research combines the two fields of science that I am most interested in, computer science and economics. I want to apply my knowledge about computational techniques to problems in the domain of economics. Thus, my research is focused on computational intensive problems in economics that cannot be solved by "standard" software but require special knowledge and attention.

This field of research has already existed for quite some time and is called Computational Economics, a subfield of Computational Social Science.

2.1 Agent-based modelling

My thesis is centered around simulations, more exactly simulations which are based on agent-based models. These kinds of simulations will henceforth be referred to as ABMs; similarly I will abbreviate agent-based simulation as ABS.

Agent-based models are used when researchers want to model a (whole) system. However, this is done not by modelling the system *as a* whole, but by modelling the parts that make up the system and the interactions of the parts with one another.

These parts are what we called the "agents". An agent can be any part of a system that can act on its own, that has its own behaviour. The behaviour of one agent can then influence other agents. This mutual influence may lead to the endogenuous emergence of global, i.e., system-wide, phenomena. An example is the emergence of business cycles in economic simulations.

In applications of ABMs in biology, an agent could be a cell and the system a fiber/cell structure such as an organ. In the social sciences, where ABM have recently gained more attention, an agent is typically a person, institution or another (legal) entity and the system to be modelled is a group or a society. In this thesis, I will concentrate on applications of agent-based models in economics, where consumers, firms, banks and governments constitute the agents and the system is, for instance, a (national) economy.

As my thesis combines two quite diverging areas of science, I will present the related literature in parts: First, I will give an overview of work in computer science concerned with simulations and how to carry them out most efficiently. Special focus will be laid on parallelization and high performance computing. Secondly, a review of work in economics concerned with agent-based models will be presented. I will mostly focus on the methodological advancements, not on how models were implemented.

2.2 Computer Science

Methodological advances in computer science as well as increases in computational power have made it possible to carry out ABM simulations with an ever larger number of agents and increasingly complex, i.e. computationally intensive, behaviour.

Simulations are a topic in its own right in computer science and touch upon a series of other topics currently discussed in computer science, such as parallelization, cloud computing, "big data" and heterogeneous computing.

2.2.1 Simulations

The following list provides an overview of key terms and definitions used in the thesis at hand:

- A model is an abstraction of a real-world phenomenon, process or system.
- A simulation, then, is the operation of a model.
- A simulation is, thus, a dynamic imitation of the modelled real-world system.

Simulations are used in all fields of science. In 2005, the U.S. Presidential Information Technology Advisory Committee published a report that stresses the importance of computational science in scientific discovery and claims that posits it as the "third pillar"[1] of doing science, besides theory and experiments.

[1]See Reed et al. (2005), p.1.

There is a long list of types of simulations that have been developed over the years which include, for example, physical simulation, interactive simulation, continuous simulation and many more. Of those, *discrete event simulation* is the most interesting type of simulation for agent-based models in economics. Discrete event simulations model time passing by means of a timeline of discrete events. At the time of each event, the system modelled can change its state. Since it is assumed that the system does *not* change between those events, it is not necessary to model the time between discrete events.

Events can take place either at regular or irregular time intervals, depending on which model is being operated. Regular time intervals could be, for instance, every quarter of a year in an economic simulation or every day in a simulation of an ant population. Irregular events, on the other hand, could be aircraft arrivals or departures in a simulation of an airport.

Computer scientists are, however, not only concerned with the implementation of such a simulation. The need and desire of researchers to model increasingly large and complex systems led to a growing focus on the efficient execution of simulation. This field, called parallel discrete event simulation (PDES), is concerned with developing strategies to efficiently carry out simulations using more than just one process[2].

PDES is an active topic of research, see Fujimoto (2016) and Carothers et al. (2017) for an overview of currently pursued research avenues. Collier & North (2013) review current work on PDES with regard to agent-based modelling.

Before going deeper into current work in simulation and modelling, I want to establish an understanding of a what an agent-based model is. This will help us to better see the requirements for the efficient (parallel) execution of an agent-based simulation.

2.2.2 Agent-based models in computer science

Bandini et al. (2009) give an overview of agent-based modelling from the perspective of computer science. They define three key elements of an ABM:

An agent-based model contains:

1. Autonomous, interacting agents with (possibly) heterogeneous behaviour. The agents' behaviour can be either reactive or pro-active. Reactive agents are typically defined by simple condition-action rules and are memoryless. Pro-active or cognitive agents, on the other

[2] I use the most common definition of 'processes' or 'threads': A process is (an instance of) a computer program that has its own address space. A 'thread' is a part/subroutine of a process and shares the address space of the "parent" process.

hand, keep track of their states and their behaviour may depend on external and internal influences.

2. An environment that manages the (ensemble of) agents and the overall system. The environment may also govern some laws to which the agents are subject (e.g. the maximum number of interactions an agent is allowed to undertake).

3. A mechanisms for interaction between the agents: an infrastructure that enables the agents to communicate and/or influence each other. There are two possibilities of how the agent interaction can be handled: either through direct, point-to-point communication or through indirect communication. In a *direct* communication infrastructure, agents usually have a unique identifier that allows an agent to actively and directly address any other agent. Direct communication infrastructure is the more prevalent option among the ABM frameworks. In *indirect* communication, agents communicate through spatial features or a special artifact. A spatial feature could be a position in space, from which agents emit certain message(s) upon arrival (e.g. a traffic light or a gravitational force "message" in an particle motion model). A communication artifact is similar, however, it lacks the explicit spatial aspect: Think of a bulletin board where the status updates of agents are passively posted in certain points in time.

   The way agents interact is an important property of the simulation. A given ABM might much more easily be realized and run more efficiently with one ABS programming framework than with another. The question is how well the ABM (and its three components described above) can be mapped to available functionality of the programming framework.

   Furthermore, the interaction mechanism is very crucial when it comes to parallel execution of the simulation as it is a main predictor of the parallel performance (in terms of execution time).

2.2.3 From ABM to OOP and Parallelization

Almost all ABM frameworks rely on object-oriented programming methods. The types of agents are defined as agent classes where class member variables are the *memory* while *state* of the agent and methods of the class refer to the *behaviour* of the agent. In the simulation, multiple objects (the "agents")[3] are then instantiated from one agent class and manipulated in the course of the simulation. As each object contains its own data fields, different objects (from the same agent class) can act very differently (to the same external

[3]"Agent" and "object" will be used synonymously from here on out.

influence)[4]. Then, as each object is self-contained, these ABM simulations lend themselves very well to parallelization[5].

Parry & Bithell (2012) argue that there are two strategies of how to successfully parallelize an ABS: either "agent-parallel" or "environment-parallel". In the "agent-parallel" approach, a given agent population is distributed (in some optimal way) over all available processes. Thus, load balancing tends to be easier but increased communication between the processes may be needed to synchronize events on separate processes. "Environment-parallelization** is used if there is a geographical space in the simulation which can be distributed to the processes. This approach reduces the communication costs of local agent interactions but increases communication costs if the agents are very mobile and need to be migrated from one process to another frequently. Additionally, as agents can be distributed very unevenly among the processes, load balancing is difficult to achieve.

2.2.4 ABM frameworks and toolkits

I will now present two frameworks in the agent-based modelling domain that each follows another one of the above mentioned parallelization strategies. Repast (Recursive Porous Agent Simulation Toolkit) HPC is an example of an environment-parallel approach but in the course of which the environment does not necessarily need to be a spatial dimension. The environment can also be defined on the basis of to relations between the agents (modelled as nodes and edges). The second framework, FLAME (Flexible Large-Scale Agent Modelling Environment), follows the agent-parallel approach.

There is a large number of agent-based modelling tools available nowadays. Rousset et al. (2014) and Abar et al. (2017) both provide an overview of the available tools and their functionality. Abar et al. (2017) survey more than 80 different applications.

2.2.4.1 FLAME

FLAME (Coakley et al. (2006), Coakley et al. (2012)) follows a unique approach to ABMs: It uses "communicating X-machines" as a formal basis for the agents' memory and behaviour. A communicating X-machine is similar to a finite state machine, but extended with random access to memory.

[4]This mapping from agents to objects is so self-evident that researchers in the ABM domain are now even adopting the UML specification for describing their models (see Bersini (2012)).

[5]Parallelization of applications is one of the most important topics in performance critical computations since improvements in the clock rate of a CPU came to a halt around 2002.

An X-Machine is defined as a 8-tuple

$$X = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$$

that consists of

- the input (Σ) and output (Γ) alphabets
- a finite set of states Q,
- memory M,
- state transition functions Φ that map the input alphabet and memory address to the output alphabet and (possibly different) memory
- a state transition diagram F that maps one state to the next
- q_0 and m_0 are the initial state and initial memory

At every state transition, the X-machine is able to access its memory, read input messages and send output messages. The state transitions chained together, then, define the "acyclical state machine"[6] that is also the behaviour of the agent during an iteration of the simulation[7].

Although an X-machine is very close to an agent class definition (as in Repast below), it significantly reduces the complexity of the code as well as the amount of code a user has to write as FLAME takes care of, for instance, scheduling the state transition functions[8].

For communication, the X-machine can use incoming and outgoing messages. These messages can be used as point-to-point messages or as broadcast messages without a specific receiver. Internally, FLAME uses "message boards" to coordinate the communication process. All outgoing messages are collected at a message board and synchronized across all processes. FLAME, as Repast, uses the MPI library for the inter-process communication. During the synchronization of a message board, agents can execute the next state transition function(s) until they need to read from this (or another) message board. If the synchronization of the message boards has already finished, the agents are free to read the messages available in the message board. If this is not the case, the agents have to wait until the synchronization is finalized. Thus, the simulation is only synchronized at message board reads even though FLAME is also an event-based simulation like Repast.

By means of clever setting (or re-arranging, if possible) of the state transition functions and their input/output messages, the user can increase the work

[6]See Coakley et al. (2012), p. 539

[7]In this section, "agent" and "X-machine" are synonymous.

[8]In principle, the tight corset of an X-machine offers great potential for optimizing the execution of the state transitions. However, this potential is only tapped in the next version of FLAME, FLAME II. See Chin et al. (2012) for a detailed explanation of the optimization plans.

an agent does while waiting for the next message board to be ready for reading and, thus, reduce the time a process remains idle. The content of a message can be freely set by the user.

FLAME consists of a parser called *xparser* and a separate message board library called *libmboard* (which exists in a serial and a parallel version). FLAME requires the user to write the model definition file (in XMML, which is a XML-dialect), where the user defines

- environment variables,
- the agents as well as
  - the variable that define the memory of an agent,
  - the states and the transition functions (with possible input and output messages) between the states and
- the messages and their variables.

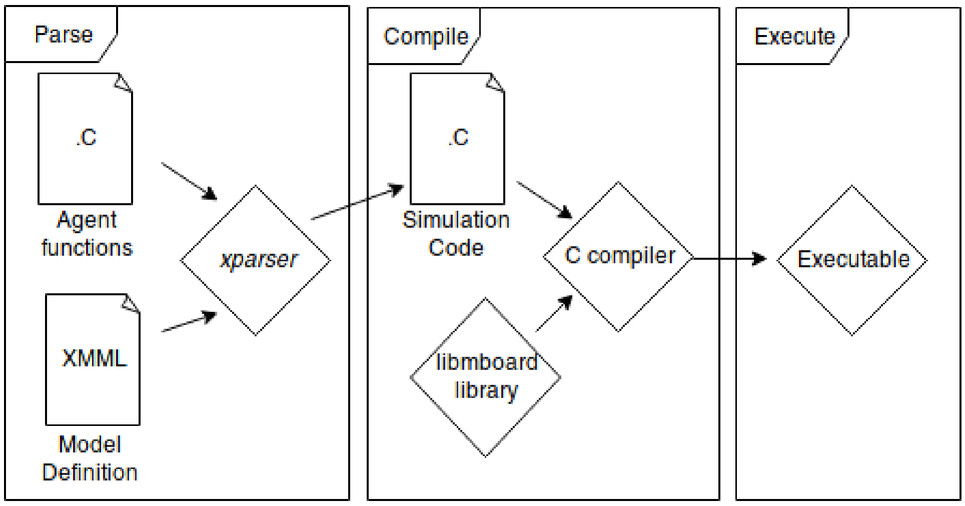

Figure 2.1: Production workflow with FLAME

The agent transition functions themselves are written in C in separate files. *xparser* parses the model definition, combines it with the agent functions and generates the simulation code. The produced simulation code can be either serial or parallel. The simulation code can then be compiled and linked with the corresponding serial or parallel message board library. Figure 2.1 gives an overview of the steps that need to be carried out.

All parallelization is actually done through the usage of message boards. The agents do not send and receive messages themselves, they only post message to their local message board. Similarly, the users do not need to work with the message board library as the *xparser* will generate the methods that make use of the functionality provided by *libmboard*. Section 3.3 explains in further detail how the message boards work and how parallelization is achieved.

Algorithm 1 gives a rough overview of the structure of the `main` function that is generated by FLAME.

```
begin
    Initialisation of variables, MPI, message boards, etc;
    Distribution of agent data etc;
    N is the number of iterations to be done
    for $i \leftarrow 1$ to $N$ do
        the following code fragment is generated for every state transition
          function f of an agent {
        while message boards needed for f not yet synchronized do
            block;
        end
        the set X contains all agents on this process that do this transition
          function
        for $x \in X$ do
            $x.f()$;
        end
        start synchronization of messages posted in $f$;
        }
    end
end
```

Algorithm 1: Schematic structure of the generated `main`

2.2.4.2 Repast

Collier & North (2013) present the Repast framework, which is, after NetLogo[9], probably the best known ABM framework. There are two versions of Repast available:

1. Repast Simphony, a Java-based modelling tool that is meant for learning and implementing small simulations.
2. Repast HPC, a C++-based modelling system that is targeted at performance-critical applications and can be run on large computing clusters and supercomputers.

For the topic of this master thesis, Repast HPC is clearly the more interesting alternative.

[9]See Tisue & Wilensky (2004)

Repast HPC uses the object-oriented programming paradigm to implement the agent's memory and behaviour. Every agent has an identifier, an *id*, which has the ability to uniquely identify an agent (across all processes). Every process manages its status and is responsible for a subset of the agent population (how this subset is defined is explained further below). Furthermore, each process has access to a scheduler that executes functions at a pre-defined schedule. A schedule is made up of "ticks"; a tick indicates a particular stage of the simulation. At each tick, all processes are synchronized ensuring that all processes coincide (?) at the same tick. Collier & North (2013) call this a "conservative PDES synchronization"[10]. Thus, it is the user's task to ensure that the load is evenly distributed over the processes.

Several options exist as to how agents can interact in Repast:

1. Since all agents are objects, agents can simply call methods from other agents. However, the programmer has to make sure in advance that the agent object accessed is present on the current process.

2. *Projections:* Collier & North (2013) introduce the notion of a projection. A projection imposes a structure on the agents. This can be a simple graph structure where edges between agent nodes define some sort of relationship between them. Repast then ensures that all neighbours of a certain agent X (i.e. all agents which are connected through edges with X) are copied/synchronized to X's process.

   In Repast terminology, agents can be *local* or *non-local* to a process: Local agents are agents that are located at the current process and non-local agents are agents that are located on another process but can be copied to the current process (e.g. if there is a connecting edge between them). Repast takes care of the synchronization so that all non-local agents are up-to-date.

   There are two types of projections that have a spatial meaning: a grid space and a continuous space. In a grid space, the position of an agent is defined by two integers while the position is defined by two floating point numbers in continuous space. Grid space is helpful because it allows a very fast query on neighbours of an agent X (i.e. agents that are geographically close to X). A process is responsible for a certain part of the grid (or continuous) space and manages all agents that are – at that moment – situated in this part of the grid. If an agent moves across the boundary of a process, Repast transfers the agent to the correct process.

[10]See Collier & North (2013), p. 9

The projections define which agents have to be synchronized or moved from one process to another. This communication between processes is established by means of serialization. Repast maintains a list of agents that have to be synchronized for every process. The "sender" process serializes the agents and sends them to the "receiver" process using MPI functions. The receiver process then de-serializes the received, non-local agents and either updates an existing agent copy or creates a new agent.

Though Repast takes care of the task of keeping non-local agents synchronized, the user still has to supply the methods for serializing and de-serializing.

Projections are a very powerful feature, especially when implementing a simulation with a spatial dimension and agent movement. In economic simulations, however, there is rarely an explicit spatial dimension. Furthermore, most economic interactions between agents involve exchanging relatively small bits of information (e.g. consumers sending an order for goods to a firm requires only two floating point numbers: the amount of good requested and the maximal price they are willing to pay. The firm responds again with two floats: the amount it can deliver and the price it charges). When Repast synchronizes agents across processes, much more information than really needed is transferred. Thus, the communication structure is not well-suited for accommodating an economic ABM[11].

2.2.5 Topic in Parallelization

2.2.5.1 Load balancing

There is another aspect of ABMs that influences the performant execution of a parallelized simulation: the distribution of the agents to the processes. This distribution determines the amount of interaction that happens between agents that are on different processes and, subsequently, the amount of inter-process communication.

As inter-process communication is much more costly than intra-process communication, minimizing the former while increasing the latter is a profitable strategy for increasing performance. The problem of minimizing inter-process communication boils down to a graph partition problem where the agents represent nodes and the interactions between the agents constitute edges. Antelmi et al. (2015) evaluate five approaches of how to partition a graph (of a distributed agent-based model) and relate them to the execution time of the simulation. They find that the graph partition algorithm and the partition strategy have a strong influence on the simulation's performance .

[11]See the results of the practical course that are presented in section 2.2.6.

Collier et al. (2015) set out to test a graph partitioning with the Repast HPC framework and a large-scale epidemiological ABM. The agents, reflecting persons in this simulation, move around a city (Chicago) and may meet other, possibly already infected agents. Collier et al. (2015) distribute the agents over the processes according to the agents' known activity profiles (the graph-separating problem) and achieve a speedup of 1.25 compared to a random separation of the agents.

Márquez et al. (2013) follow a similar path: they extent the FLAME framework with agent migration routines and a load balancing algorithm. Based on the execution times of the last iteration, every process computes an "imbalance factor" and a certain number of agents is then moved from an overloaded process to an underutilized one. They are able to decrease the overall execution time of their example simulation by 17 to 27 %. In a follow-up research, Márquez et al. (2015) extend the load balancing algorithm with an indicator for message connectivity. This allows them to move agents not only based on the execution time of the process, but also depending on the connectivity pattern of the agent. They can, thus, try to minimize the necessary inter-process communication. In several tests, Márquez et al. (2015) manage to achieve a reduction of up to 30 % of the execution time.

2.2.5.2 Shared memory multithreading

Despite the seeming agreement of researchers to use MPI for parallelization, there is also the alternative of using OpenMP. Massaioli et al. (2005) argue that there is a potential to use OpenMP for accelerating ABMs. A disadvantage of OpenMP is that it was designed to be used with "classic numerical codes, where there are arrays and loops on indices"[12]. In ABS however, agents are objects that are often kept in linked links. Iteration then happens over those lists, which cannot simply be parallelized with a `#pragma omp for` directive. Still, Massaioli et al. (2005) show three ways[13] to speed up the list traversal and are able to achieve efficiencies close to 1 from two to 16 threads (given a large agent population).

2.2.5.3 GPGPU computing platform

Lysenko & D'Souza (2008) proposed using a GPU for carrying out an ABS. They show that porting a simple model (insects moving around a 2D plane) to a GPU can lead to a speedup of around 9000.

[12]See Massaioli et al. (2005), p. 7

[13]One way is to use `#pragma omp parallel` before and `#pragma omp single nowait` after the declaration of the for-loop. This way, only one thread will work on one iteration while the remaining threads move on to the next iteration.

A few years later, researchers at the University of Sheffield presented an extension of the FLAME framework that is able to offload the computation of an ABM on to a GPU (Kiran et al. (2010), Richmond et al. (2010)). The FLAME GPU framework is used by Heywood et al. (2015), who model road networks and their utilization. This sort of simulation is well-suited for being computed on a GPU as there is only one type of agents (a car) and the agents' only action is to iteratively update its current position (depending on the position of cars in front of them). They are able to model up to 260000 agents in reasonable time. FLAME GPU has also been used in the domain of biology: Konur et al. (2015) model a "pulse generator" with up to 100000 cells and Richmond et al. (2009) are able to achieve speedups ranging from 100 to over 10000 (depending on the problem size) for a model of keratinocyde cells.

There exist several other agent-based models whose computation is done on a GPU. These models are, however, mostly simple, demonstrational models which were specifically programmed for that purpose and which often have a spatial dimension. Research in this area has been conducted by, for instance, Wang et al. (2014) (investigate how mood spreads) or Tang & Bennett (2011) (construct a model of spatial opinion diffusion).

A problem with porting an ABM to the GPU is that a parallel ABM is closer to the SPMD[14] technique while GPU are well-suited for SIMD[15] applications. Thus, only ABMs that are simple rather than complex and contain a large agent population can be ported to the GPU and achieve good performance results.

2.2.5.4 Applications on supercomputers

Though many research articles concerning ABM tag themselves with the label of "High performance computing", not much of them are executed on a supercomputer. High performance computing is mostly used to refer to "non-standard" computing hardware, such as an Intel Xeon Phi accelerator.

An older, but well-known agent-based model that has been executed on a supercomputer is an epidemiological model for pandemics, described in Epstein (2009). Said model builds on a previous one concerning the spread of disease, but is scaled up considerably: It enables the modelling of an agent population of 6.5 *billion*. This model is implemented on the Global Scale Agent Model (GSAM, Parker & Epstein (2011)). GSAM is written in Java and follows a hybrid parallelization strategy: At every node, a JVM runs

[14]SPMD stands for Single Program, multiple data in Flynn's taxonomy of computer architectures, see Flynn (1972).

[15]SIMD stands for Single Instruction, Multiple Data in Flynn's taxonomy of computer architectures, see Flynn (1972).

with several worker threads at its disposal. GSAM follows an environment-parallel strategy. Communication is achieved through Remote Method Invocation where RMIs are transferred in bulks between nodes. Parker & Epstein (2011) also provide tables of performance measurements, weak/strong scaling, etc. When scaling the application from one to 32 nodes, they achieve the following efficiencies: a value of 40.6% when considering weak scaling and 31.5% with strong scaling.

Collier & North (2013) report timings of a rumour-spreading model that they implemented in Repast HPC. The model simulation is then run on an IBM Blue Gene/P machine with up to 40k processes.

The EURACE model, described further below, is an agent-based macroeconomic model and is implemented in FLAME. In Deissenberg et al. (2008), EURACE is claimed to have been let run on a supercomputer but there are no mentions of any time measurements.

In another contribution to the literature, in Kiran et al. (2008), they do test and report the parallel performance of the EURACE. Their results remain inconclusive: On one tested machine, a Cray XT4 scalar supercomputer with a total of 22,656 cores, they run the EURACE model on four to 16 processes and find only modest improvements in average iteration execution time. On two other machines, the average iteration time either increases with an increasing number of processes or fluctuates heavily.

Makarov et al. (2014) report supercomputer applications of ABMs. They develop a socio-economic simulation for Russia. Even though they know of the existence of FLAME and Repast HPC, they choose to develop their own software stack. Using a supercomputer with up to 4000 processes, they are able to model 100 million agents over 50 years (one iteration is a year) in just 90 seconds.

2.2.6 Preparational results

As a preparation and first inquiry into the area of parallel agent-based modelling, I chose to implement and study a simple agent-based computational model, specifically the model given in Lengnick (2013)[16].

This agent-based model features a set of households and a set of firms. The households sell their labor to a firm and receive a wage in return. From the wage they buy goods that the firms produce. If a household is unemployed, it searches for work by asking firms for open vacancies. Firms employ households/workers to be able to produce goods. They sell the goods to households (for a small profit) and distribute the profits back to their owners,

[16]This work was done in the course of a practical course that serves as a preparation for the master thesis at the University of Vienna, faculty of Computer Science.

again the households. This model is – despite its simplicity – able to show the endogenous emergence of business cycles, without any "disturbance" or "shock" from outside.

The model primarily captures the day-to-day actions of households and firms. It is mainly concerned with the consumption planning of the households and the planning of production by the firms. It is not, contrary to the benchmark model that is examined in this master thesis, focused on correctly modelling the flows of money between institutional sectors. Nevertheless, this simple model is a good indicator of the suitability of an modelling framework for a typical ABM in economics.

I implemented the model in both FLAME and Repast and studied their runtime behaviour in different parameter settings to gain an understanding of which of the two frameworks is better suited for a more sophisticated economical ABM.

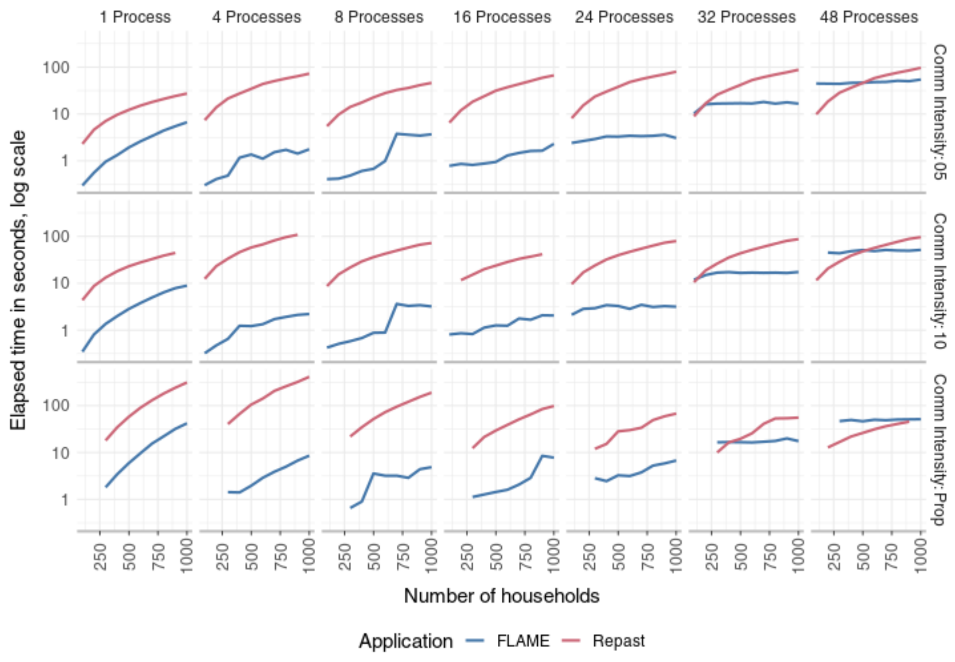

Figure 2.2: Performance comparison of FLAME and Repast

Figure 2.2 shows the key results of the performance experiments. Every panel depicts the results for a given combination of a number of processes and communication intensity (abbreviated "Comm Intensity"). Communication intensity measures the number of firms a household interacts with on a daily basis, i.e., potentially buys goods from. When a households addresses more firms, there is more communication between agents and thus also more (costly) inter-process communication. A communication intensity of five is meant to be the lower bound and ten a upper bound[17]. Additionally, I created a model with *proportional* communication intensity, where the number of firms a household interacts with rises with the total agent population. These three communication intensity specifications are shown in the rows of figure 2.2.

The x-axis shows the problem size, given as the number of households that are simulated.

We can see that FLAME outperforms Repast in every parameter setting. Especially with a smaller number of processes is the discrepancy very large. The y-axis (log scale) shows that the difference in execution time, e.g. for four processes, is almost in the magnitude of 100. The difference in runtime gets smaller as more processes are used to execute the simulation, but FLAME tends to do better every time.

Another interesting observation is that the curves of execution times of FLAME tend to start at a higher level but also to get flatter as more processes are employed. That means, with enough processes, the execution times of FLAME are independent of the problem size (at least for the range of problem sizes that are tested here).

Repast tends to gain some ground the more processes are utilized, but all in all it performs worse than FLAME. This might be a consequence of the design of Repast: Repast creates a graph of connections between the agents and makes sure that the neighbours (all agents an agent has connections with) of each agent are synchronized to the process of that agent. However, there is already a large number of networks present in this simple model: a work-search network, a goods-price-inquiry network, a goods-order network, a goods-delivery network and so on. This leads to an almost p-fold duplication of the agent population (where p is the number of processes) and consequently a large amount of inter-process communication that has to be done several times during one iteration.

FLAME on the other hand uses message boards to exchange information between the agents. Nearly all of the exchanged data between agents contains only one or two floating point numbers or integers (such as a price quote, amount of goods demanded by the household or a boolean indicator whether there is an open vacancy at the firm). Messages are synchronized between the processes while the agents can continue their actions. Taking this all together, FLAME seems to be a much better fit for the implementation of a sophisticated ABM.

[17]This values are taken from Lengnick (2013).

2.3 Economics

In economics, ABMs are part of *Agent-based Computational Economics* and are, thus, often called ACE models. I will use ABM and ACE, in the context of economics, interchangeably.

After the economic crisis of 2008, where a large part of the economics profession failed to see that a crash was coming, voices grew louder that called for new and better models, e.g. in Farmer & Foley (2009), Colander et al. (2009) or Korinek (2015). The then most-used class of models, called DSGE (Dynamic Stochastic General Equilibrium) models[18], were amended to be able to show behaviour observed in real life (i.e. the ability to crash). However, these amendments to the model proved to be unsatisfactory (as they still cannot fully reproduce the economic dynamics of the financial crisis and have to rely on exogenous shocks to bring an economy out of equilibrium).

Looking for other ways to model an economy, some proposed using agent-based simulations[19]. Agent-based models in economies are not new, Leigh Tesfatsion and Robert Axtell are two longstanding proponents of this approach[20]. Just recently, the Oxford Review of Economic Policy published a whole issue on "rebuilding macroeconomic theory". Several authors, e.g. Haldane & Turrell (2018), highlighted the benefits of ABMs and urge to draw more attention and research to the development of these kind of models.

A very basic ACE model is given in Lengnick (2013). It serves mainly as an introduction to agent-based modelling, featuring only consumer and firm agents. However, even in this simplistic setting, business cycles can be observed.

Gerst et al. (2013) use an ABM to model climate policy and the multiple feedback loops that are present in an ecological system made up of an economy, energy technology and ongoing climate change.

Gualdi et al. (2015) explore "tipping points" in an ACE model: A tipping point is when an economy which is in a "good" steady state with low unemployment suddenly transitions to a "bad" steady state where the unemployment rate is high. Gualdi et al. (2015) build a minimal model that exhibits these properties and give policy recommendations based on their findings.

A special type of ACE models has recently received growing attention: stock-flow consistent (SFC) models. SFC models are models that follow national accounting principles, meaning that, for instance, national output (GDP) is the sum of consumption, investment and government expenses or that

[18]See e.g. Smets & Wouters (2003)

[19]See Nature (2009), Farmer & Foley (2009) and Economist (2010)

[20]See their books Tesfatsion & Judd (2006) and Epstein & Axtell (1996)

consumption plus taxes plus saving of the households must be equal to the income of the households. Godley & Lavoie (2007) is the main reference for this type of model. A disadvantage of these models is that they are aggregate macroeconomic models and they only contain the (aggregated) institutional sectors (private sector, i.e. households and firms, public sector, financial intermediation). As a response to this shortcoming, economists developed agent-based models that followed the stock-flow consistent accounting principles, adequately named agent-based stock-flow consistent (AB-SFC) models.

The first models to adopt these AB-SFC principles is the EURACE model by Deissenberg et al. (2008). It includes nine types of agents (households, consumption good firms, investment good firms, banks, malls, a clearinghouse, a government, a central bank and a central statistical agency) and "many hyper-realistic features"[21]. These features, however, severely increase the complexity of the model and make it hard and time-intensive to be re-used, modified or extended by other economists.

To circumvent this problem, Caiani et al. (2016) developed a simpler "benchmark" AB-SFC model. It is a compromise between a simple, very abstracted (and thus unrealistic) model and a very elaborate, complex (but realistic) model. This benchmark model features six types of agents: households, capital and consumption firms, banks, a central bank and a government. The behaviour of those agents is simplified but the model is still able to reproduce important stylized facts about aggregate economic variables[22]. There are already several papers that build on the benchmark model, such as Caiani, Russo, et al. (2017), Caiani, Catullo, et al. (2017) and Schasfoort et al. (2017). Thus this model is a good starting point for my planned inquiry into the performance characteristics of an agent-based macroeconomic model.

[21]See Caiani et al. (2016), p. 380

[22]I will describe the model in more detail in the next chapter.

Chapter 3

Investigated model

3.1 Description of the benchmark model

The so-called "benchmark" model of Caiani et al. (2016) is, as has already been mentioned, an agent-based stock-flow consistent model.

It consists of six types of agents: households, consumption good and capital good firms, banks, a cenral bank and a government.

- Households buy goods from consumption good firms and deposit their savings at banks. They sell their labor to firms and receive wages as income. From the income, one part goes to the government as income tax. If the household is unemployed, it looks for work but receives unemployment benefits from the government. Every household owns firms proportional to its wealth and receives dividends accordingly.
- Consumption firms use labor and capital goods (which they buy from capital good firms) to produce goods which they then sell to the consumers/households. They plan their production and hire/fire workers and take out loans as needed. Taxes are paid from profits. A part of the remaining profits are paid out as dividends and a part is kept for financing future production. Capital firms only use labor for production and only sell to consumption good firms.
- Banks hold deposits of the households and firms and give loans to firms. They pay interest on the deposit and receive interest on loans. Taxes are paid from profits, remaining profits are fully distributed to the households. If needed, banks can ask for cash advances from the central bank.
- The central bank is the issuer of the legal currency. It holds the banks' reserves and hands out cash advances to banks. Furthermore, the central bank buys any government bonds that are not bought by the banks themselves.

- The government collects taxes from households, firms and banks. It pays unemployment benefits to unemployed households and employs a certain share of the households itself. Deficits are financed by issuing government bonds.

Households make up the biggest part of the agent population. Therefore, I will indicate the problem size of a simulation with the number of households.

Every iteration in the simulation represents one quarter of a year.

The benchmark model is concerned with correctly modelling

- the creation of money by the central bank (by purchasing government bonds and by providing cash advances)
- the creation of money by banks when new loans are granted to firms and the destruction of money when loans are repaid.

For a more detailed description of the model, its behaviour and results I refer the interested reader to the original paper of Caiani et al. (2016).

3.2 Implementation of the model

I implemented the model of Caiani et al. (2016) as described in the paper. Some agent behaviour, however, would have been very difficult to implement in FLAME:

- Household agents have a certain budget to be consumed. Additionally, households have a fixed number of consumption firms that they do business with (this list of firms changes from iteration to iteration). In the paper, households choose a random firm from their list and buy as much goods as possible (either the household runs out of money or the firm out of inventories). If the household still has money left, it again randomly selects a firm from its list and tries to satisfy as much of its consumption demand as possible.

  This behaviour is not implementable in an easy way in FLAME. Households and firms are separate agents, they can only communicate through messages. Agents are allowed to receive and send messages only once per transition function[1]. It is thus not possible – as would be needed here – to send an order for goods, receive a message containing the amount of delivery and then send again a message to another firm.

[1]They can however send/receive multiple messages from multiple message boards.

In my implementation, households send inquiries to all firms on their business partners list (in one transition function), receive information about the inventories of the firm and respond with an order (in the next transition function) and finally receive a delivery of goods by the firms. This behaviour is close to the described behaviour in the paper, but it can lead to unsatisfied consumption: As all consumers send their orders at the same time, firms can – unexpectedly – run out of inventories and consumers have then no way of compensating this unsatisfied demand with other firms, since all orders have already been sent. This difference in behaviour, of course, also influences the macroeconomic outcomes of the model, but this difference is insignificant for our performance experiments.

- The behaviour of households, firms and banks is influenced by some macroeconomic trends: Households adjust their reservation wage depending on the "global" unemployment rate; firms' wage offer to new workers depends on the average wage in the economy and banks' interest rate for loans and deposits moves towards the average of the last quarter. To be able to model this behaviour, I had to introduce an additional agent to my implementation: a statistical office. The sole task of this agent is to aggregate the data that is sent to it (e.g. households send their employment status, i.e. 0 if they are employed and 1 if they are unemployed) and send the aggregate again out to the agent that need it.

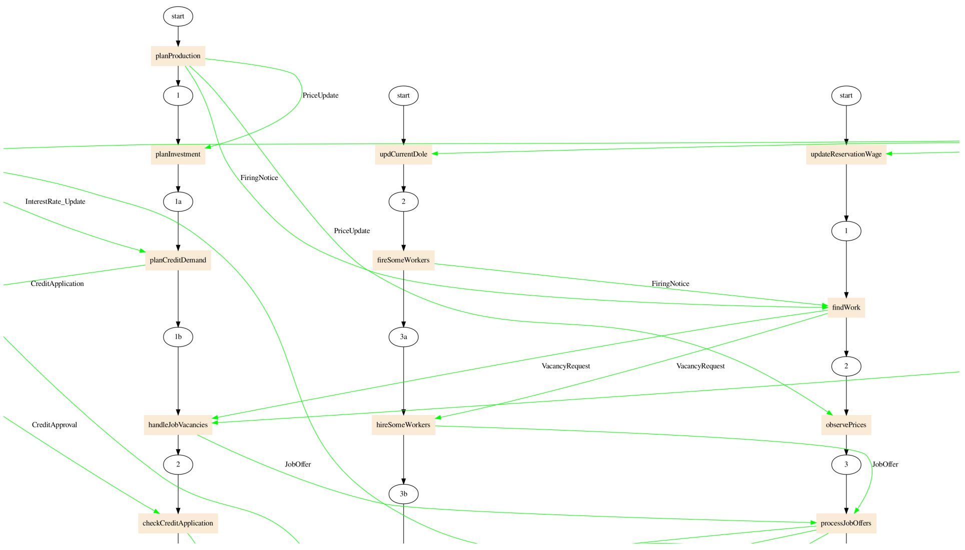

Figure 3.1: Extract of the stategraph generated by FLAME for the model of Caiani et al. (2016)

Figure 3.1 presents a part of the state graph that is generated by FLAME to organize the transition functions of the agents. States are ellipses with white background, the transition function are the orange rectangles and the green arrows between them depict messages.

The transition function and their dependencies among each other are given in the model XMML file. The state graph is generated from this information and used by FLAME to schedule the transition function is such a way to achieve the biggest possible parallelization.

3.2.1 Calibration of the initial state

Stock-flow consistency begins with the initial state. Caiani et al. (2016) present their proposed solution in their appendix: Ensuring that the initial state is, indeed, stock-flow consistent means solving a system of non-linear equations.

The number of households should be the main indicator of the size of the initial state. Thus, I define the number of consumption firms in dependency of the number of households: Φ_C as $\Phi_C = \max(1, \lceil \Phi_H * \phi_C \rceil)$, where Φ_C is the number of consumption firms, Φ_H is the number of households and $\phi_C = 0.05$ a scaling factor for the consumption firms. The number of capital firms[2] and the number of banks[3] is defined accordingly. Thus, the number of households in a simulation is the most relevant indicator of the problem size and I will use the number of households to represent the problem size.

With the numbers of households, firms and banks determined as described above and given the exogenous parameters from Caiani et al. (2016), I can generate stock-flow consistent initial states for all problem sizes needed. These are then the initial states that are used for conducting my performance experiments. The initial state for a certain problem size is always fixed. What differs in each run are the random choices the agents make (the random seed is not fixed).

3.3 Parallelization[4]

FLAME achieves parallelization of the generated simulation code through the usage of message boards. These message boards have the task of collecting, synchronizing and providing the messages that agents send to one

[2]Defined as $\Phi_K = \max(1, \lceil \Phi_C * \phi_K \rceil)$ with $\phi_K = 0.2$.

[3]Defined as $\Phi_B = \max(1, \lceil \Phi_C * \phi_C \rceil)$ with $\phi_B = 0.1$.

[4]This section is based on the description of FLAME/libmboard given in Kiran et al. (2008), Coakley et al. (2012) and Chin et al. (2012)

another. More technically, the message boards are distributed data structures that ensure that the messages are replicated on all processes (that need them). Messages are `structs` that can contain any information.

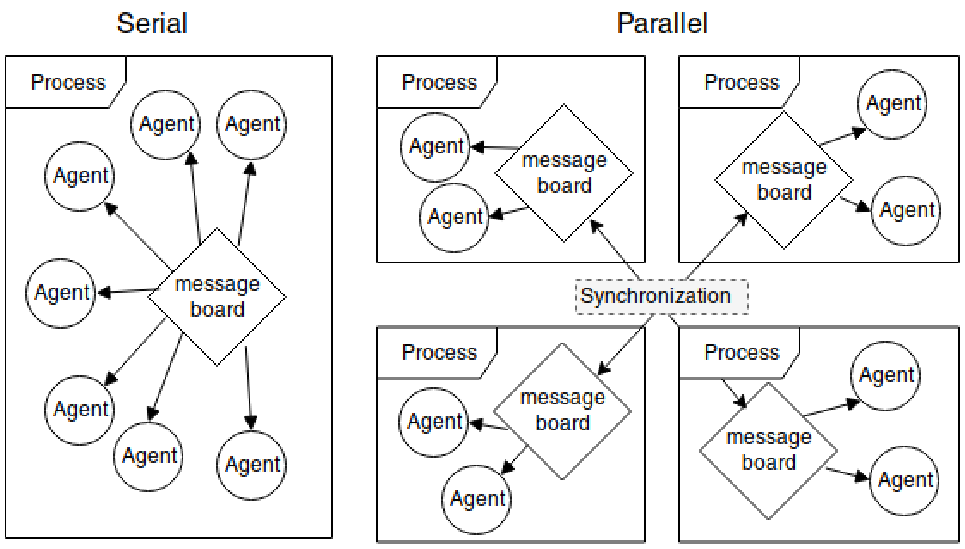

Figure 3.2: Messageboard parallelization[5]

Every process starts its own *communication thread*. These threads manage the message boards (one message board for every type of message) of the simulation. Agents can post messages to these boards using a `MB_AddMessage()` routine. Once a message is posted, the agents can continue with their work. After the last message is posted to the message board, synchronization of that message board is started. Agents can read the messages of the boards after the synchronization is finished. If the synchronization is still ongoing, the agents are put on hold.

Thus, the best parallelization efficiency is achieved if messages are sent as early as possible, so that a) the communication threads have enough time to synchronize and b) the agents can do as much as possible useful work while the message boards are being synchronized.

By default, the message boards and their messages are fully replicated on all processes. This can of course lead to a lot of unnecessary duplication of messages. To circumvent this, FLAME provides filters. If the user adds a `filter` tag to the model description in the XMML file, libmboard will only synchronize the messages that a certain process needs. E.g., a filter tag requests that the `ReceiverID` variable of the message be equal to the `ID` variable of the receiving household. Every communication thread informs the other threads – prior to the synchronization – which messages it needs: in

[5]This figure is based on a similar presentation in Chin et al. (2012)

this case, which agent `IDs`. Each process then "tags" every written message with the processe(s) that need this message. This allows the message boards to be only "partially" synchronized and to avoid a full replication of the message board on all processes.

Chapter 4

Numerical simulations

4.1 Hardware setup and configuration

4.1.1 Laptop

The first machine to run the experiments is my personal laptop. As most researchers in economics have “normal” commodity hardware at their disposal, this is a very interesting benchmark.

The processing unit of the laptop is an Intel Haswell i5 4200U DualCore with 1.6 GHz and enabled hyperthreading. Thus it only makes sense for the experiments to use at most four concurent processes. 8 GB of memory are built in.

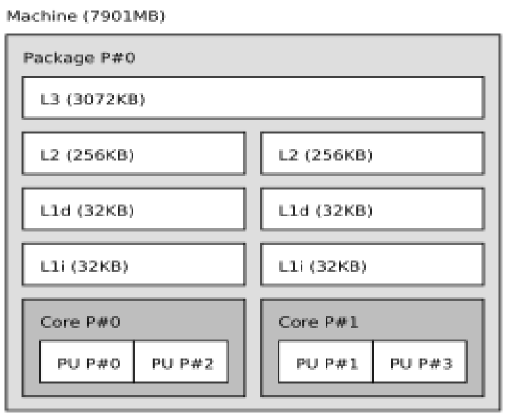

Figure 4.1: Topology on the laptop

Figure 4.1 shows the topology of the system as generated by the utility `lstopo`. The laptop has one 3MB level 3 cache, and two cores with each a 256 KB level 2 cache and a 32 KB level 1 cache.

The operating system is an up-to-date Arch Linux. Specifically, the gcc 8.2.0 and OpenMPI 3.1 will be used. Furthermore, the compiler flags `-O3 -march=native -ffast-math -funroll-loops` are activated throughout all experiments[1].

4.1.2 Lewis

The second machine is a multicore called "Lewis" and is a server at the University of Vienna. With four processors and twelve cores each as well as 256 GB of memory it is considerably more powerful than the laptop. Lewis can thus host up to 48 cores. Every of the 48 AMD Opteron cores is able to run at 2.2 GHz. Figure 4.2 gives an overview about the topology.

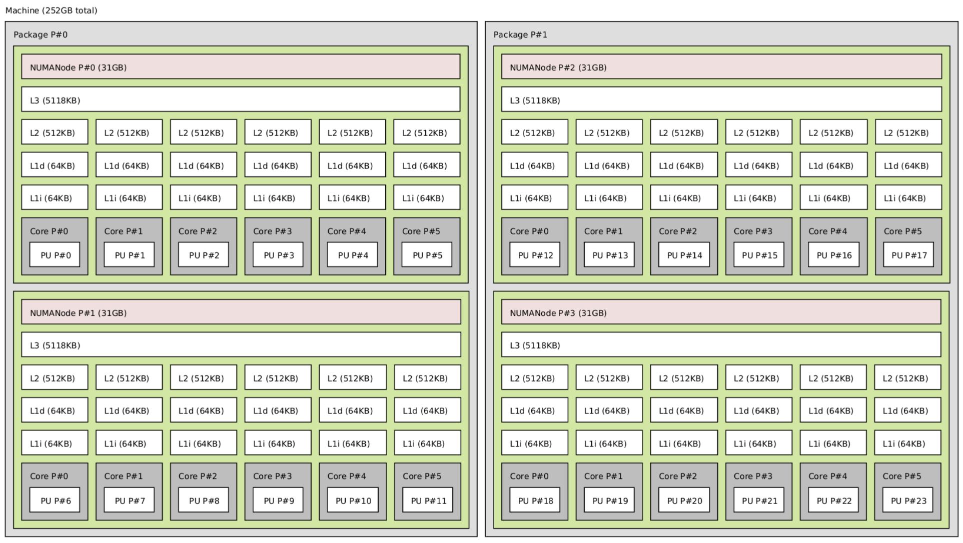

Figure 4.2: Topology on Lewis (shown are two of four sockets)

The operating system is again a Linux variant: Ubuntu 16.04.01 LTS runs on Lewis. The FLAME application code is compiled with gcc 5.4.0 and MPICH 3.1.

AMD also recommends using `-march=native` when compiling locally (and using gcc)[2]. Thus the same compiler flags are the same as on the laptop: `-O3 -march=native -ffast-math -funroll-loops`.

The theoretical peak performance for one AMD Opteron processor is at

[1]See the appendix section A.3 for detailed information which flags are set by `-march=native`.

[2]See https://developer.amd.com/wordpress/media/2012/10/CompilerOptQuickRef-61004100.pdf (accessed 4.7.2018)

110 GFLOPS/s[3]. As there are four processors built into Lewis, its peak performance is 440 GFLOPS/s. Similarly, dividing by twelve gives a figure of 9 GFLOPS/s per core. Or calculated from another side: each core has a frequency of 2.2 GHz and could complete four floating point operations per instructions, which would amount to 8.8 GFLOPS/s.

4.1.3 Vienna Scientific Cluster (VSC)

Finally, I am given the opportunity to run my experiments on the Vienna Scientific Cluster, an Austrian supercomputer.

The current system is called VSC-3[4] and is a Linux-cluster with 2020 nodes. Each node contains two Intel Xeon E5 processors with 2.6 GHz and eight cores. Thus, it can accomodate up to 16 processes (or even 32 processes when hyperthreading is enabled). The architecture of a compute node at the VSC-3 is given in figure 4.3.

The theoretical peak performance of the VSC-3 is 681 TFLOPS/s[5], which means one compute node could achieve 337 GFLOPS/s. Calculating from the bottom up, we multiply the frequency of a core (2.7 GHz) with the number of floating point operations it can do per cycle (eight) and multiply the result with the number of cores on a compute node (16) we arrive at 346 GFLOPS/s which is close enough to the figure above.

A VSC node features 32 GB of memory for both sockets. The level 3 cache is with 20MB considerably bigger, compared to the 5 MB at Lewis above. Every socket has its own interface to a dual-link Infiniband fabric, which internally connects all compute nodes.

The operating system on VSC-3 is CentOS. There are a lot of different compilers and MPI libraries available on the VSC. I wanted to compare the open source compiler gcc with the Intel compiler as well as the recent release of OpenMPI 3.0 with its previous version 2.0.

Thus I chose to build the FLAME application with

- gcc 7.2.0 and OpenMPI 2.0.2 (abbreviated as “OpenMPI 2”),
- gcc 7.2.0 and OpenMPI 3.0.0 (abbreviated as “OpenMPI 3”) and
- Intel compiler 18 and Intel-MPI 2018.3 (abbreviated as “Intel-MPI”).

Additionally, for every configuration a message board library with the same settings was compiled. The compiler flags `-O3 -march=native`

[3]See https://www.amd.com/Documents/AMD_Opteron_ideal_for_HPC.pdf (accessed 15.8.2018)

[4]I will use “VSC” and “VSC-3” synonymously.

[5]See https://www.top500.org/system/178471 (acessed 18.8.2018)

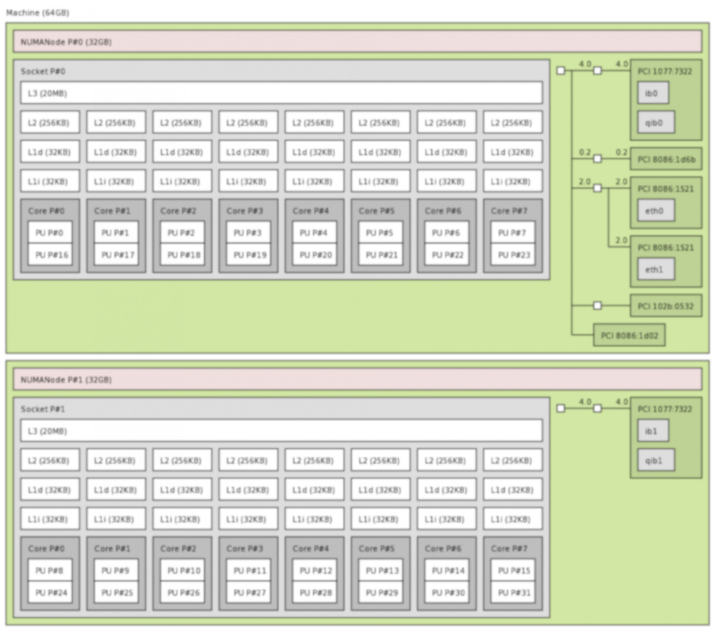

Figure 4.3: Topology of a VSC node

`-ffast-math -funroll-loop` are activated when compiling with gcc. With Intel, the flags `-O3 -AVX` are set[6].

4.1.4 Event measurement

To measure the execution time, I use the timing that is readily supplied by the generated code. This timing is taken by using the function `MPI_Wtime()` on the root node (node 0). In addition to the elapsed time, the PAPI framework, described in Terpstra et al. (2010), measures several events during the simulation. PAPI has the advantages that it has a relatively low overhead and that it is available on all three machines.

Table 4.1 shows the events that are measured on each machine. As the number of available counters for these hardware events is different on every machine – e.g. there are only two counters available on Lewis – it was necessary to run the experiments twice: once to get the measurements for total instructions and floating point operation and once to get the measurements of the cache misses. This, in turn, means that the elapsed time was mea-

[6]This settings are proposed by NASA High-End Computing Capability. See https://www.nas.nasa.gov/hecc/support/kb/recommended-compiler-options_99.html (accessed 21.7.2018). Additionally, for the application to take advantage of the processors' vector processing features it is necessary to include the `-lirc`-flag during linking with mpiicc.

sured twice as much (as it is measured in every run) as the other events and can thus be considered to be a more stable and reliable result.

Table 4.1: PAPI Events that are possible to be measured on each machine

Event	Description	Laptop	Lewis	VSC
PAPI_TOT_INS	Total instructions	X	X	X
PAPI_FP_OPS	Floating point operations		X	X
PAPI_L1_TCM	Level 1 cache misses	X	X	X
PAPI_L2_TCM	Level 2 cache misses	X	X	X
PAPI_L3_TCM	Level 3 cache misses	X		X

Table 4.1 also shows that two events are not available: there are no exposed hardware counters for floating point operations on the laptop, which has a Intel Haswell architecture (see also next section). Furthermore, level 3 cache misses cannot be measured on Lewis.

4.1.5 Performance tests setup

To investigate the performance properties of FLAME for the benchmark model, I will carry out a series of tests. Each test corresponds to a simulation run with a certain number of processes and a certain problem size (i.e., with a given number of households). The execution time as well as some of the events above, e.g. cache misses, are recorded and stored for later analysis. Every test will be carried out at least five times, so that we can average the time and event observations and arrive at robust measurements.

FLAME is able to generate serial and parallel code. In both cases there is one separate thread (per process) that carries out the computations of the message board library. I will use the serial code for simulations with one process[7], and the parallel version for two or more processes.

The number of simulation iterations is set to 10 so that even bigger problem sizes terminate within reasonable time. In practice, researchers will want to execute a lot more iterations[8]. However, since iteration n and $n+1$ are similar to each other, in the type and amount of work they do, we could – with reasonable accuracy – simply extrapolate the time needed for 400 from the result for 10 iterations.

I controlled the generated simulation data exemplarily. They all generate sensible data which could be, e.g., used in an economic research project.

[7]It would also be possible to use the parallel version with just one process. However, the serial version would then have no advantage over the parallel version, as it would also have to cope with the MPI overhead.

[8]In Caiani et al. (2016), the authors simulate 400 iterations, i.e., 100 years

4.2 Metrics

The following metrics of the experiments will be examined. **Elapsed time**, which is simply the time from start to end of a simulation, is the most interesting figure. For analysing the work done during the simulation, we will look at the number of **floating point operations** that was carried out. For the Intel Haswell architecture of the laptop, we will use **total retired instructions** instead.

From these two measurement (elapsed time and work done), we can calculate several performance metrics. **Speedup** $S(p)$ is calculated as

$$S(p) = \frac{T(1)}{T(p)}$$

where $T(p)$ is the elapsed time with p processes. Similarly, **efficiency** $E(p)$ is defined as

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{T(p) \times p}$$

Redundancy $R(p)$ and **utilization** $U(p)$ are then calculated as

$$R(p) = \frac{O(p)}{O(1)}$$

and

$$U(p) = R(p) \times E(p)$$

where $O(p)$ is the number of (floating point) operations performed. Finally, we will also have a look at the level 1, 2 and 3 **cache misses** as this will help us understand certain performance issues of the application. Specifically, we will use the cache miss rate calculated as the number of cache misses per instruction.

4.3 Results

The upcoming section presents the results of the performance tests. As I carried out these tests in three different hardwares, the results are presented in turn, beginning with the hardware with the "smallest" computing power, my laptop.

On the laptop and on Lewis, the "smaller" problem sizes are tested while the "bigger" problem sizes are tested on VSC.

4.3.1 Laptop

4.3.1.1 Elapsed time

As there are up to four slots available for processes, the simulation is tested with one, two and four processes. We immediately see from figure 4.4 that, as the agent population increases and, as a result, the problem gets more difficult, more processes finish the simulation in less time. Since the elapsed time on the y-axis is on a log scale, the reductions in elapsed time are considerably higher than they appear: The simulation, carried out with only one process, terminates after 1300 seconds (more than 20 minutes), while the average elapsed time for two and four processes is 800 seconds (13 minutes) and 550 seconds (9 minutes), respectively. Only for very small problem sizes we see that a single process can match the time achieved by two processes. Starting from around 600 households, two processes begin to achieve faster times and from 2000 households onwards, four processes are always the fastest.

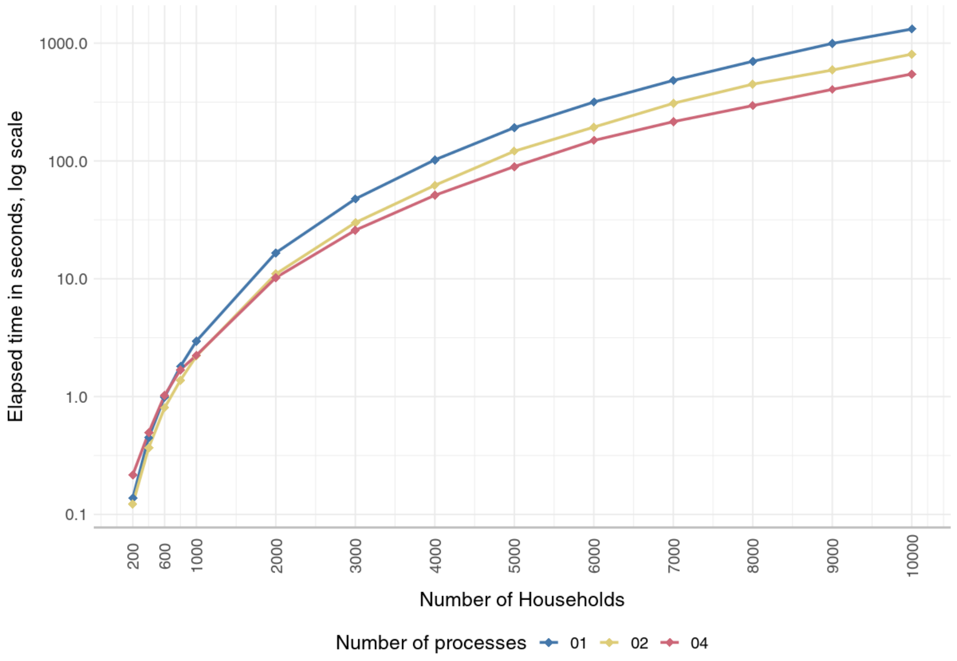

Figure 4.4: Elapsed time comparison on laptop

4.3.1.2 Instruction count

As has already been mentioned in section 4.1, the Intel Haswell architecture of the laptop does, unfortunately, not expose any counters for floating point operations. As a replacement, only the number of "retired instructions"[9] can be used. Using instructions instead of floating point operations has, however, a serious drawback: retired instructions are proportional to execution time. Retired instructions only measure the instructions that are executed, not counting speculative instructions. In fact, there are less retired instructions with four processes[10] than with one process. Thus, it does not make sense to go further into detail here and look, at the instructions per second[11].

4.3.1.3 Cache misses

Figure 4.5 displays the cache miss rates of the three levels of caches. Again, the number of processes are colour-coded, while the cache levels are represented by the various linetypes.

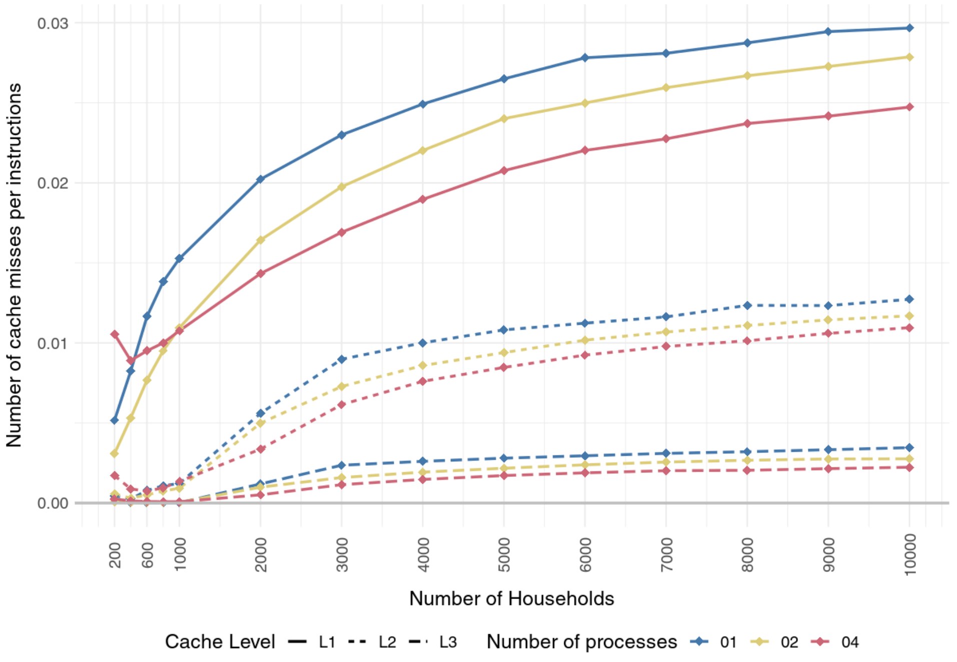

Figure 4.5: Cache miss rate on laptop

We see that the level 1 cache misses are the "most common", occuring in between 2.5 and 3% of all instructions (considering the largest problem sizes).

[9]Measured by PAPI with `PAPI_TOT_INS`. The PAPI Wiki has a longer discussion about the difficulty (or in case of Intel Haswell, impossibility) of measuring floating point operations on a series of Intel processors, see http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops (accessed 22.7.2018)

[10]This value is the sum of the retired instructions of the four processes.

[11]The results for the measured instructions can be found in Appendix A.1 for completeness

Level 2 cache miss rate is with above 1% slightly larger than usual. Level 3 cache range at 0.25%.

We can also observe that the cache miss rate is always lower for simulation runs with more processes (except for very small problem sizes). This observation indicates that there is an issue with data locality: If we distribute the data of the agents to several processes, we can decrease the amount of cache misses that the application runs into. This means further that the serial code generated by FLAME does probably not use the most efficient algorithm for traversing the agent population and their data.

4.3.1.4 Performance metrics

Figure 4.6 shows the calculated performance metrics for the laptop. We see that the FLAME framework can achieve impressing speedups for two and four processes, especially for the bigger problem sizes. Using two processes, we end up with a speedup of around 1.6 while the speedup for four processes approaches approximately 2.45.

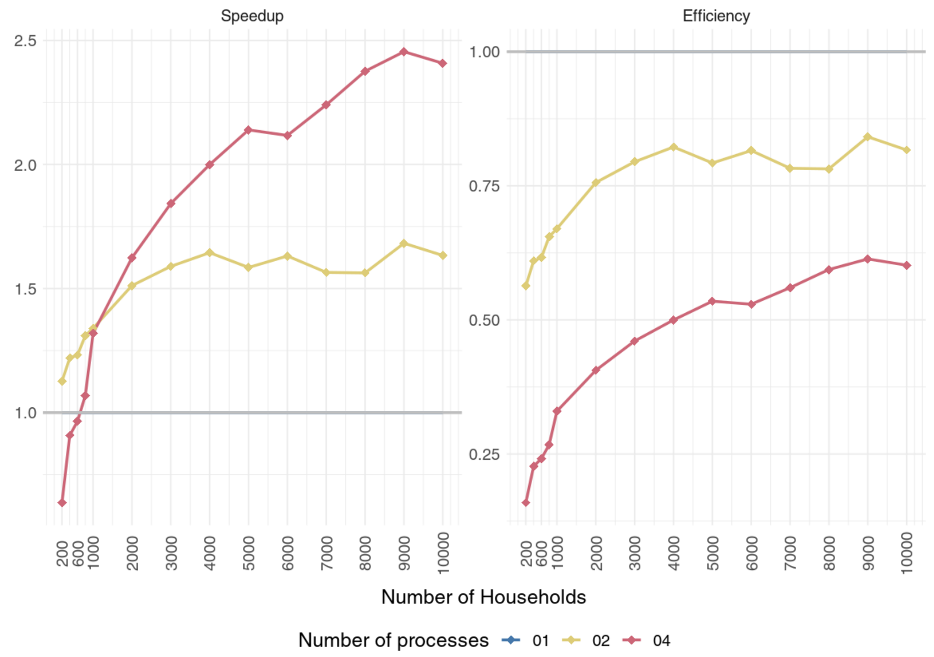

Figure 4.6: Performance metrics on laptop

Efficiency is slightly increasing with the problem size, but converges to approximately 0.8 for two processes and 0.6 for four processes.

Since we lack a good method for measuring the work done during the execution of the simulation, we are unable to calculate sensible results for redundancy and utilization[12].

4.3.2 Lewis

The following section will present the results of the machine tested next, Lewis.

4.3.2.1 Elapsed time

As in the previous section, we start by inspecting the runtime behaviour of the simulation when run with various numbers of processes as shown in figure 4.7

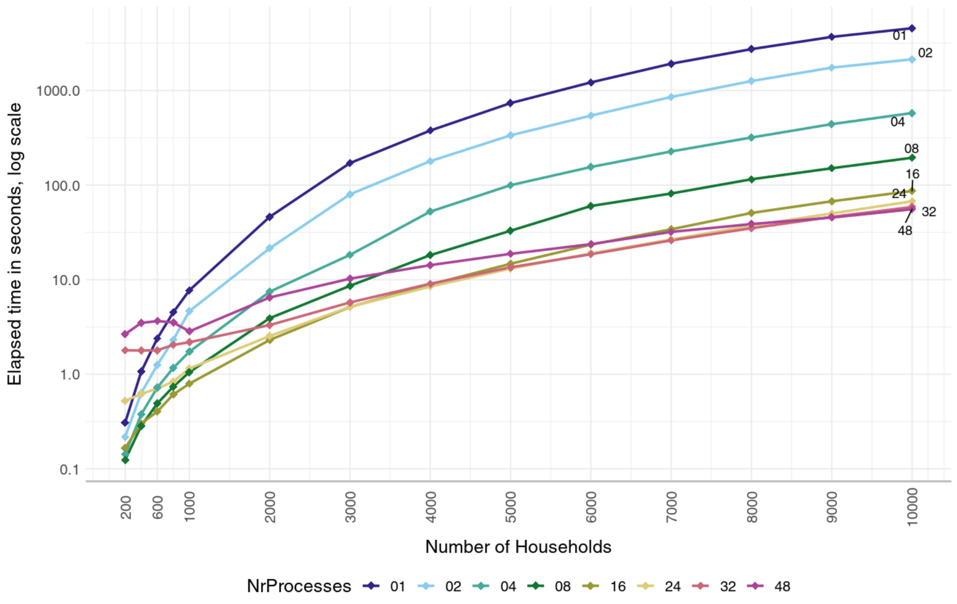

Figure 4.7: Runtime comparison on Lewis

We can see that – similar to when using the laptop – more processes tend to reduce the time needed for the simulation to terminate, especially as problem sizes increase. So even for very small problem sizes, it is advisable to use a parallel version of the simulation.

Another interesting observation to be made is that the results for 24, 32 and 48 processes start with very high values: 0.4, 1.8 and 2.7 seconds, respectively, are needed to carry out the simulation with 200 households. However, the execution times rise much less markedly with increasing problem size

[12]The results for redundancy and utilization are given in the appendix A.1 for completeness.

than simulations with less processes do – thus, they scale better. Between 600 households and 2000 households, 16 processes is the fastest configuration. At 3000 households, then, 24 processes operate quickest but are in turn outperformed by 32 processes at around 6000 households. 48 processes are fastest for problem sizes of 9000 households and more.

4.3.2.2 Achieved MFLOPS per second

On Lewis, we can measure the number of MFLOPS per second carried out during the execution of the simulation; figure 4.8 shows the results. Looking at the plot, we immediately detect a very surprising behaviour: the number of MFLOPS/s decreases in accordance with problem size when the number of processes is smaller than 16. For simulations using more processes, the MFLOPS/s increases or stays roughly the same. This means that, that as there the work load increases (the problem size grows) less work is actually carried out per second.

This circumstance might imply that this program is, in fact, I/O bound. As the problem size increases, more data is moved around and more cache misses occur (see figure 4.9). The CPU is, then, in turn slowed down by the waiting for the data retrievals to finish.

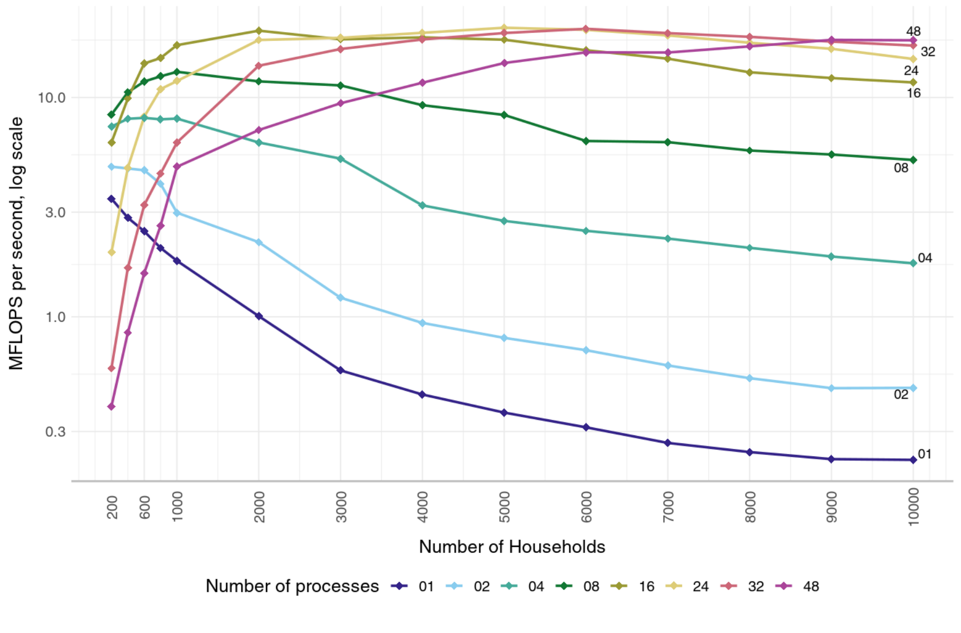

Figure 4.8: MFLOPS/s comparison on Lewis

The results further show that the level of MFLOPS/s achieved is very low. The scale on the y-axis ranges from 0.3 to 20 MFLOPS/s. The experiment with 48 processes achieves the highest performance with 18.2 MFLOPS/s.

Given that Lewis would – in theory – be able to accomplish 8.8 GFLOPS/s per core and 440 GFLOPS/s in total[13], the results shown in figure 4.8 are very disappointing.

4.3.2.3 Cache misses

Finally, we will have a look at the cache miss rates. AMD has no counters for the level 3 cache misses (or accesses), thus we have to make due with level 1 and level 2 cache miss rates. As indicated in the previous section, the cache miss rate is calculated as the number of cache misses divided by the number of instructions.

We observe that, similar to the results on the laptop, the cache miss rate tends to be lower for experiments with a higher number of processes. Except for the difference between 32 and 48 processes, differences are quite small.

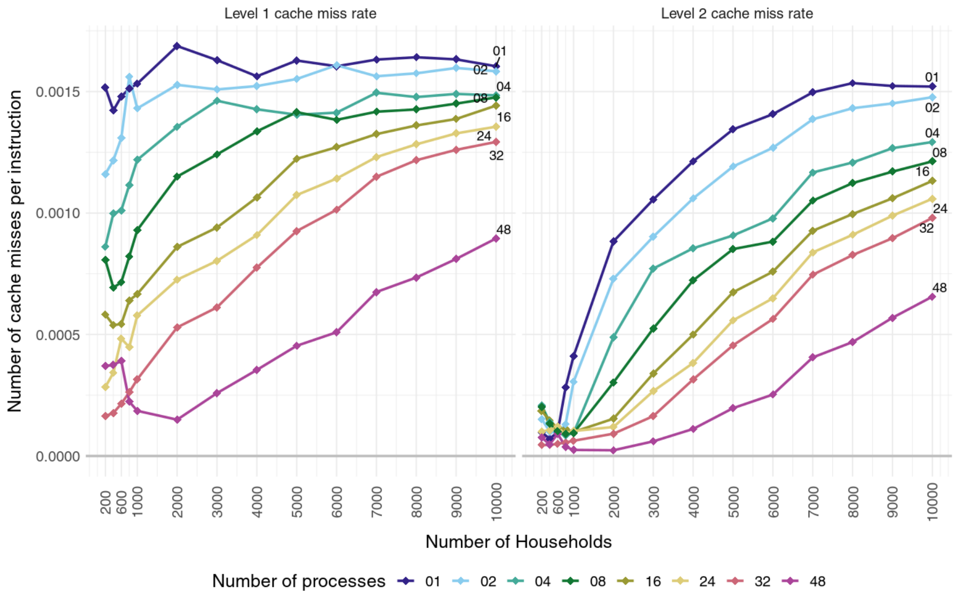

Figure 4.9: Cache miss rates for Lewis

Level 1 cache miss rates are stable for one and two processes, rise and level off for four and eight processes and increase throughout the problem size space for 16 processes and more. Level 2 cache miss rates start low for smaller problem sizes, increase constantly and are almost equal to level 1 cache miss rates in the end.

[13]See section 4.1

4.3.2.4 Performance metrics

Now, we turn to the performance characteristics of the experiments on Lewis. All processes achieve large speedups, e.g. with 48 processes and 10000 households, the speedup amounts to 82.1. Speedups for two to 16 processes converge to a certain point, e.g. for eight processes this value is about 23.4. Speedups for larger process counts (24, 32 and 48 processes) increase with problem size but seem to level off eventually. This indicates that there is a certain maximal speedup to be achieved with a given number of processors. For two to 16 processors this value is already reached within the tested range of problem sizes and lies at 2.1, 7.9, 23.4 and 52.6, respectively. For 24, 32 and 48 processes, however, it is not clear if this maximal value is already attained.

In the panel for efficiency metrics, we see that super-linear speedups are achieved for most problem sizes and number of processors. Already with a problem size of 2000, super-linear speedups are possible. The bigger the problems, the more processes achieve a super-linear speedup.

There are a number of reasons which can explain this rather surprising result: First, single process performance is bad. If the serial version of the generated FLAME code has some inefficiencies, it is relatively easy for the parallel version to reach high speedups.

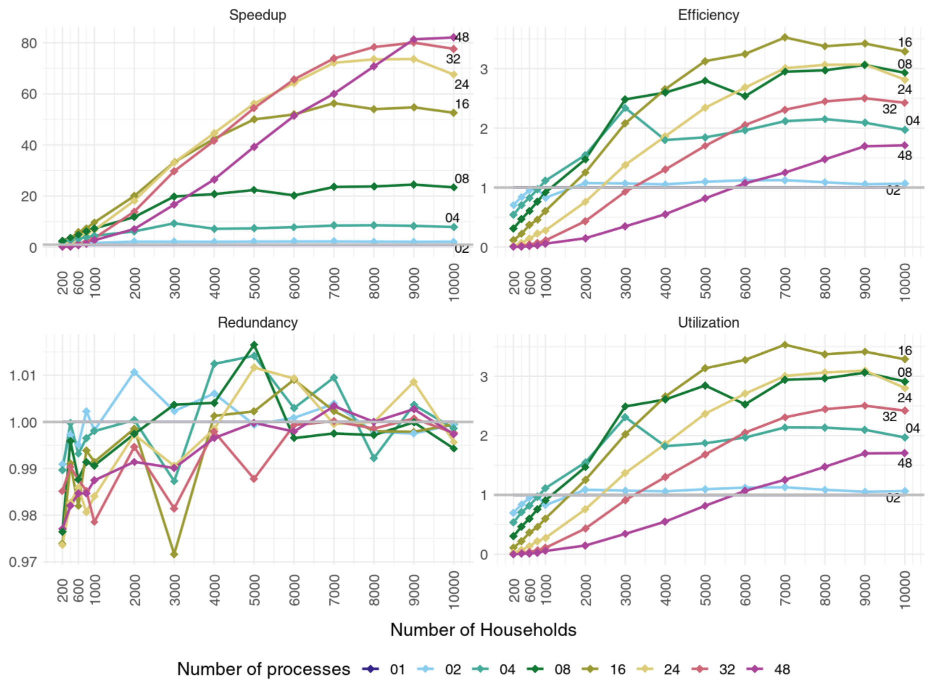

Figure 4.10: Achieved speedups on Lewis

Second, since many processes fare so much better than a single serial process, another reason can probably be found in the locality of the processes. In figure 4.9, we see that cache misses are lower for more processes, which means that cache locality is better when using more processes. However, this could also imply that the algorithm for traversing the list of agents and carrying out their transition functions is inefficient, so the parallel version can distribute this inefficient algorithm onto more processes.

Redundancy fluctuates around one, but only very little: Most values are within +/- 2%. This means that the amount of FLOPS done by the application is essentially constant and not affected by the number of processes that execute the simulation.

Utilization, the product of redundancy and efficiency, measures how well available computational capacity is being used. Since redundancy is practically equal to one, utilization shows the same patterns as efficiency.

4.3.3 Vienna Scientific Cluster

The third machine tested is the Vienna Scientific Cluster (VSC).

Every compute node at the VSC consists of 16 cores. Since we have seen from the experiments at Lewis that serial performance of the FLAME generated code is rather low, I chose to only use "full" compute nodes. This means that we will look at at least 16 processes or multiples of 16.

As the number of processors that can be utilized is much higher on the VSC compared to those on Lewis, considerably bigger problem sizes were tested here. The smallest problem size involves 5000 and the biggest 45000 households[14].

4.3.3.1 Elapsed time

Figure 4.11 depicts the elapsed runtimes of the experiments using three different compiler/MPI library configurations. Each panel in the plot refers to a different number of utilized cores, beginning from 16 cores (one compute node) and ranging up to 512 cores (32 compute nodes)[15].

There are several observations to be made here. First, runtimes tend to decrease up until the point where 256 cores/16 compute nodes are used. From 256 to 512 cores, execution time stays roughly the same.

[14]Testing even bigger problem sizes would have been possible, however this would have required the generated source code to be patched manually and the batch jobs to be submitted to compute nodes with bigger memory.

[15]The number in parenthesis above the panels indicate the number of compute nodes.

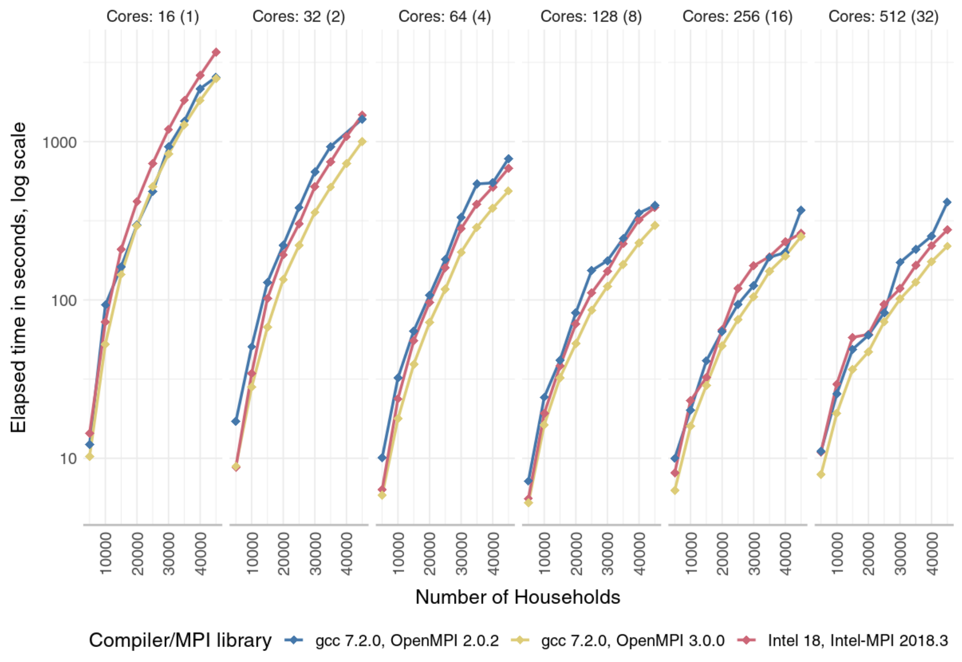

Figure 4.11: Elapsed time comparison on the VSC

Secondly, OpenMPI 3 does seem to perform better than OpenMPI 2. Notice that, again, the y-axis is on a log scale, thus the differences between the runtimes of the two OpenMPI versions are bigger than it might look like. The performance of Intel-MPI lies somewhere between the OpenMPI versions.

Third, we see the same pattern as on Lewis: The more processes we utilize, the smaller the reaction of the application to increasing problem size (i.e., the curves tend to flatten out the more processes we use).

Figure 4.12 shows the same results as figure 4.11 but in a different format. We group the results by problem size and plot the number of utilized cores on the x-axis. This way we can better see the strong scaling behaviour of the simulation. In every panel, we notice a linear decline in the execution time with a small upward bend at the end.

4.3.3.2 Achieved MFLOPS/second

Results for the achieved MFLOPS/s are low – similar to the results on Lewis – and tend to decrease with problem size but increase with the number of processes. The downward sloping curves are especially pronounced with OpenMPI 2. The MFLOPS/s for OpenMPI 3 and Intel-MPI are downward

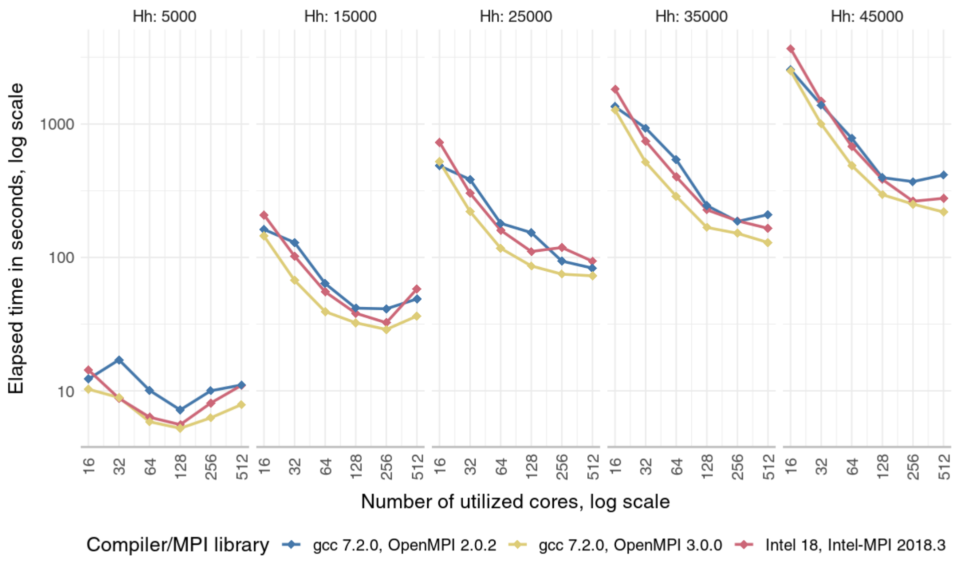

Figure 4.12: Elapsed time comparison on the VSC

sloping for 16 and 32 processes, stable for 64 processes and slightly upward sloping for 128 and more processes.

Furthermore, it is interesting to note that OpenMPI 2 seems to carry out a lot more work with 128 and more cores, while only achieving roughly the same execution times as OpenMPI 3 and Intel-MPI. This indicates that OpenMPI 2 does more – though useless – work while OpenMPI 3 and Intel-MPI just run idle.

In terms of computational efficiency, again, the results are disappointing. We saw in section 4.1 that the upper bound of *one* compute node lies at 337 GFLOPS/s. Only OpenMPI 2 achieves more than 1 GFLOPS/s (though with 16 and 32 compute node, where the theoretical peak performance is already in the range of TFLOPS/s).

As OpenMPI 3 shows the best performance with respect to execution times, I will report only the results for OpenMPI 3 in the remainder of this chapter. Appendix A.2 contains the results for all compiler/MPI libraries.

4.3.3.3 Cache misses

Cache miss rates on the VSC are presented in figure 4.14. The pattern we see here is by and large similar to that of Lewis and the laptop. Rates rise with increasing problem size but the level of the cache miss rate is lower the more processes are used in the simulation experiment.

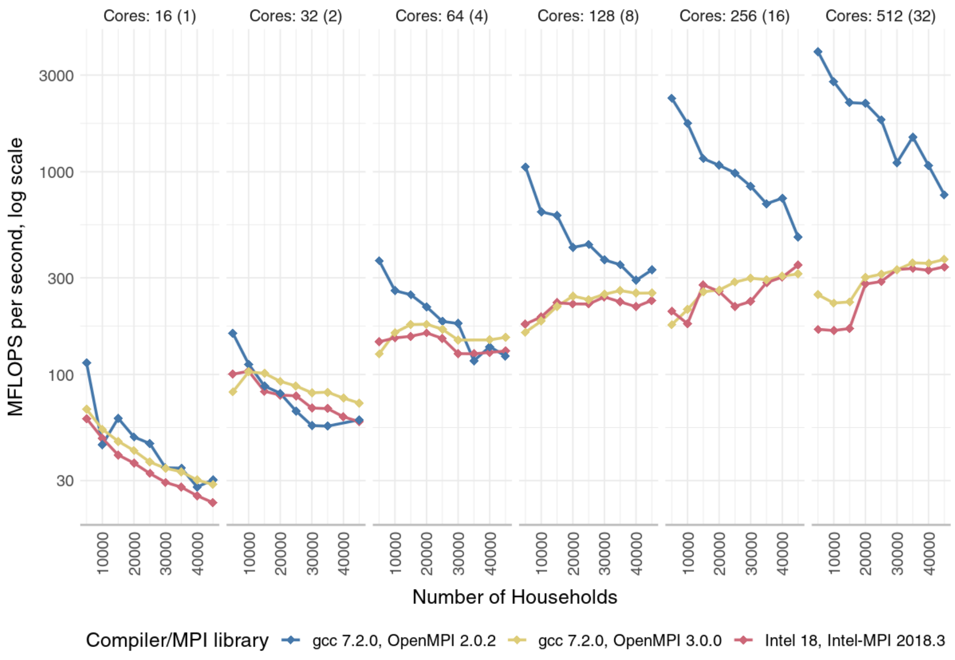

Figure 4.13: MFLOPS comparison on the VSC

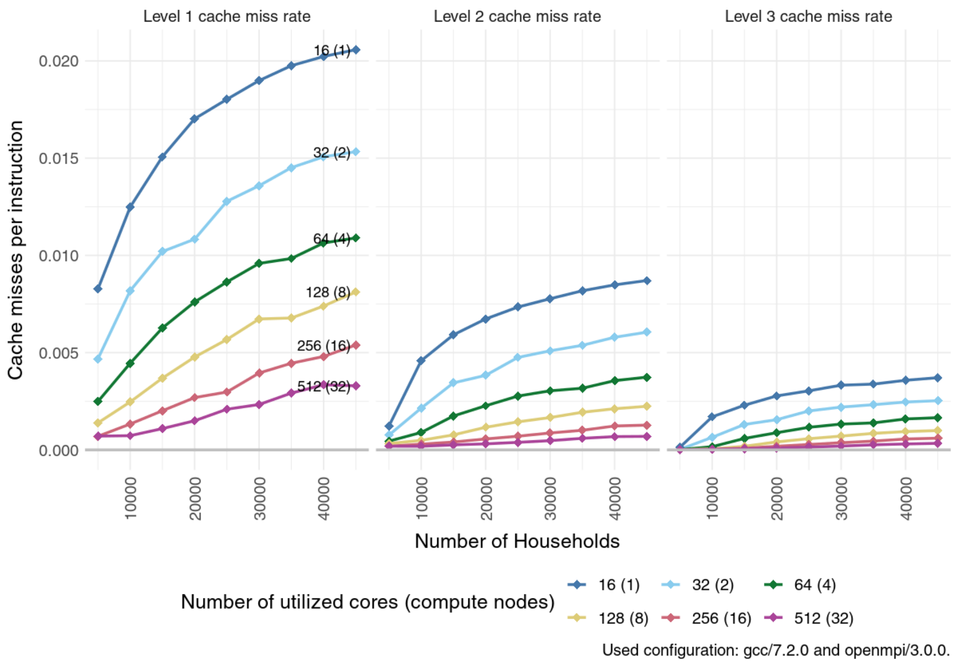

Figure 4.14: Cache miss rates on VSC

Contrary to the previous results, we see here a big gap between the cache misses of the serial code and the cache misses of the parallel code with two processes.

We should also take note of the absolute level of cache misses: It is with 2% rather high (considering we are using number of retired instructions in the denominator).

4.3.3.4 Performance metrics

Performance metrics were calculated with the results for *one* compute node, i.e. 16 cores, as reference. Using a "true serial" process, i.e., one process on one compute node, as reference for the performance metrics would have unrealistically exaggerated the results.

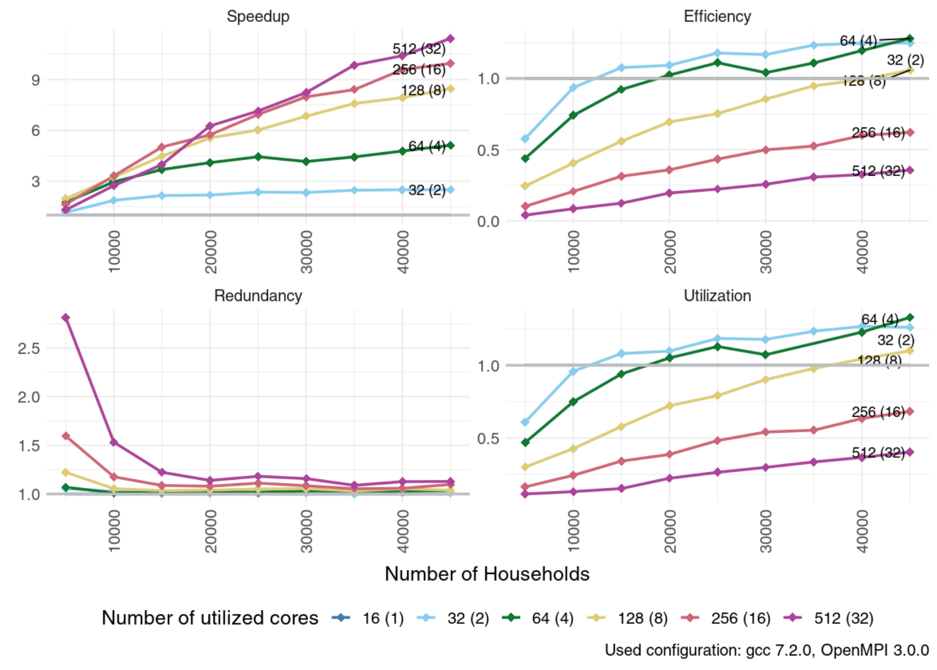

Figure 4.15: Performance metrics on VSC

Figure 4.15 shows the performance metrics for OpenMPI 3. We see that speedups rise almost linearly with problem size and the slope for each line seems to be steeper the more processes are employed. Efficiency similarly rises with problem size. 32 cores (two compute nodes) and 64 cores (four compute nodes) achieve an efficiency greater than one already at problem sizes smaller than 20000. 128 cores (eight compute nodes) also reach an efficiency larger than one at 40000 households.

Results for redundancy show a very different picture. Though all values are greater than one – as would be expected – they are not much bigger than one. At 45000 households, the biggest problem size, the values for redundancy range from 1.04 for 32 cores to 1.15 for 512 cores.

The values for utilization are very close to the values for efficiency since redundancy is close to one in most cases.

4.3.3.5 Hyperthreading

On VSC, there is the possibility of using hyperthreading. Hyperthreading enables a second, virtual, core for every physical core. It is thus possible to assign two processes to only one physical core.

Since we saw above that more processes tend to achieve lower execution times, it might be worthwile to investigate if hyperthreading can be used to accelerate the execution a bit further. Figure 4.16 displays these results.

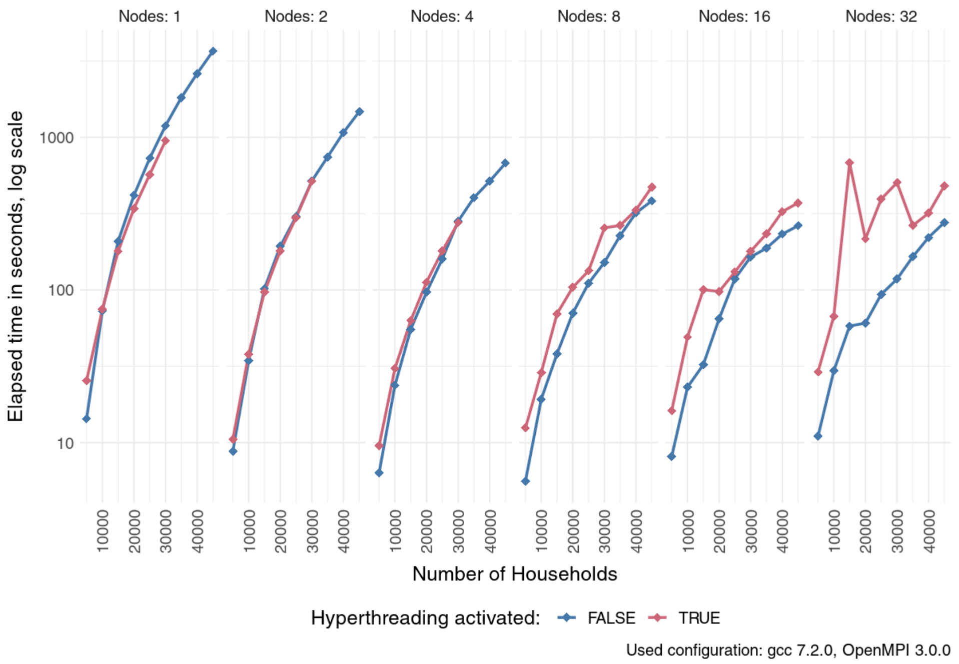

Figure 4.16: Hyperthreading on VSC

The results in figure 4.16 are grouped by the number of utilized compute nodes. Keep in mind that the results *with* activated hyperthreading use double as many processes as the results without hyperthreading.

We see that the results with hyperthreading are hardly distinguishable from the results without hyperthreading when using up to four compute nodes.

With eight and more compute nodes, hyperthreading seems to impact the performance negatively. Especially with 32 compute nodes (i.e. 1024 processes), the results are very volatile.

These results seem surprising, since we saw that the execution time always sank when using more processes, especially when starting from a low number of processes. The results favour, however, the interpretation that the program is I/O bound and the cores simply cannot get enough data to compute, no matter how many processes we use.

4.3.3.6 Load imbalance

Load imbalance is an important factor in assessing the performance of a parallel application. Since there is no automatic load balancing done by the FLAME code, we have to hope that the "round-robin" partition of the agents is good enough and does not lead to big imbalances and wasted computational ressources. The number of FLOPS per process are used to calculate average and maximum FLOPS. The load imbalance factor is calculated as

$$\mathrm{LIF} = \frac{\max_p(\mathrm{FLOPS}_p)}{\mathrm{mean}_p(\mathrm{FLOPS}_p)}$$

where p indexes the processes. Figure 4.17 depicts the average load imbalance factor during the experiments.

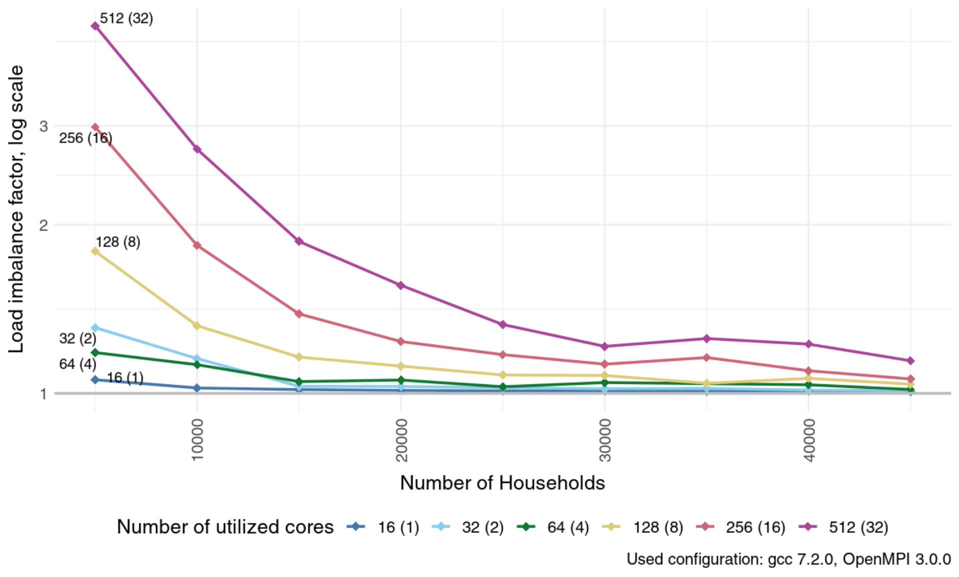

Figure 4.17: Load imbalance

We can see that the load imbalance factor tends to decrease with increasing problem size and that simulations with a smaller number of processes also has a lower load imbalance factor.

4.3.3.7 Weak scaling

Up until now, we only considered strong scaling. Strong scaling keeps the problem size fixed, while increasing the number of processes that work on solving it. Weak scaling holds the problem size *per process* fixed level, thus the overall problem size increases with the number of processes.

In general, Strong scaling is more useful but also more difficult to achieve. From the performance metrics above we can see that FLAME does quite well with strong scaling, weak scaling is only of small importance here.

Nevertheless, figure 4.18 shows the results of a weak scaling exercise. The number of households (and thus the problem size) are set at 80 per process. The number of processes is shown on the x-axis with the corresponding problem size in parentheses.

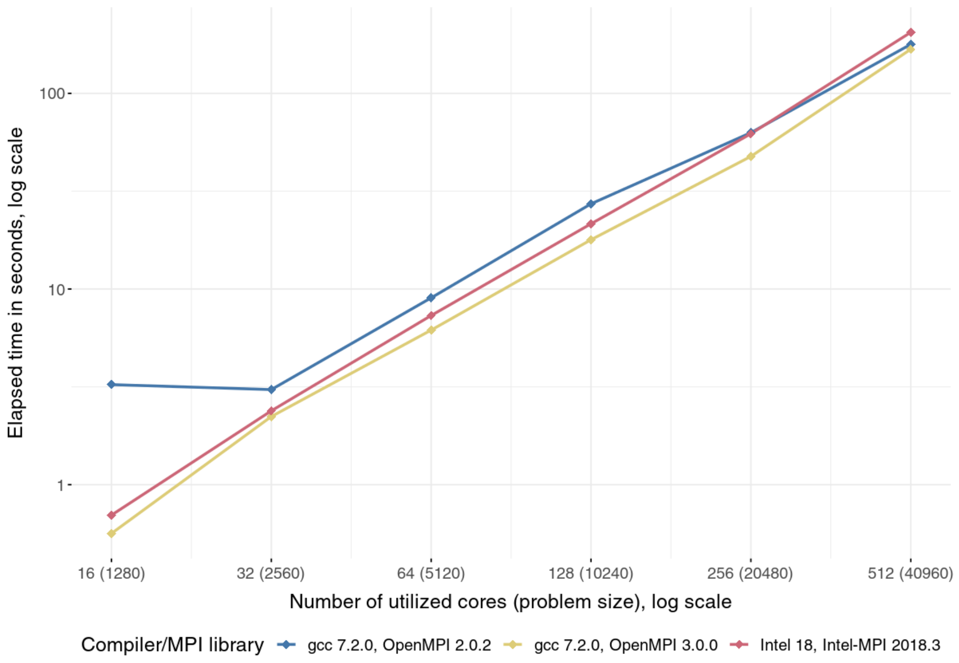

Figure 4.18: Weak scaling on VSC

It is of little surprise that the application exhibits linear weak scaling. Open-MPI 2 deviates from linearity with 16 cores but otherwise there is no doubt that there is linear weak scaling.

4.4 Interpretation of the results

There are two issues that need explaining:

1. the, in absolute perspective, bad performance
2. the super-linear speedups

The two issues are interrelated. In part, we see the super-linear speedups simply because the performance of the FLAME generated code is below our expectations.

Figure 4.19 and 4.20 show memory and floating point intensity of the simulations, based on results obtained from Lewis. Memory intensity is defined as as the number of load and store instructions[16] per instruction and floating point intensity is similarly defined as the number of floating point instructions per instruction.

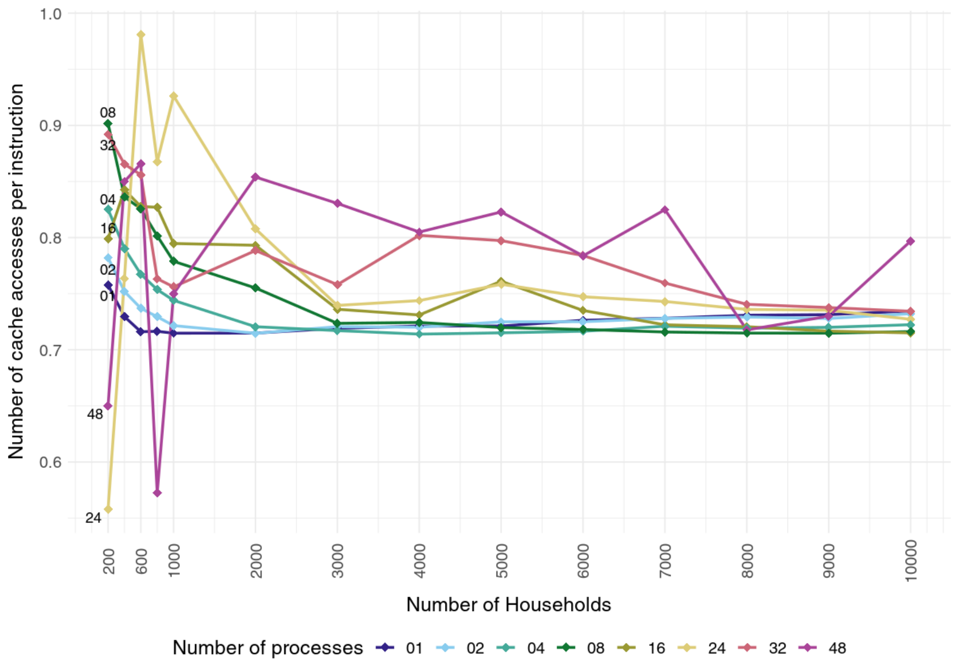

Figure 4.19: Memory intensity

The memory intensity shows us the percentage of memory operation that are done during the execution of the simulation. We see that the results fluctuate quite a bit, but tend to converge to roughly 0.72 when the problem size grows. So about 72% of all operations are inputting or outputting data from/to the memory, while less than 1% are floating operations.

[16]Or, equivalently, the number of L1 cache accesses.

This means that the program is mostly occupied with loading and storing data and is not able – by far – to utilize the computing capacity of the CPU. Thus this program is clearly I/O bound.

One reason for this extraordinary high percentage of memory operations can be attributed to the design decisions of the generated FLAME code. As each agent is considered to be its own self-contained X-machine and each transition function can, in theory, access all memory variables of the agent, the program will always load the "full" agent into memory when it has to carry out a transition function on this agent.

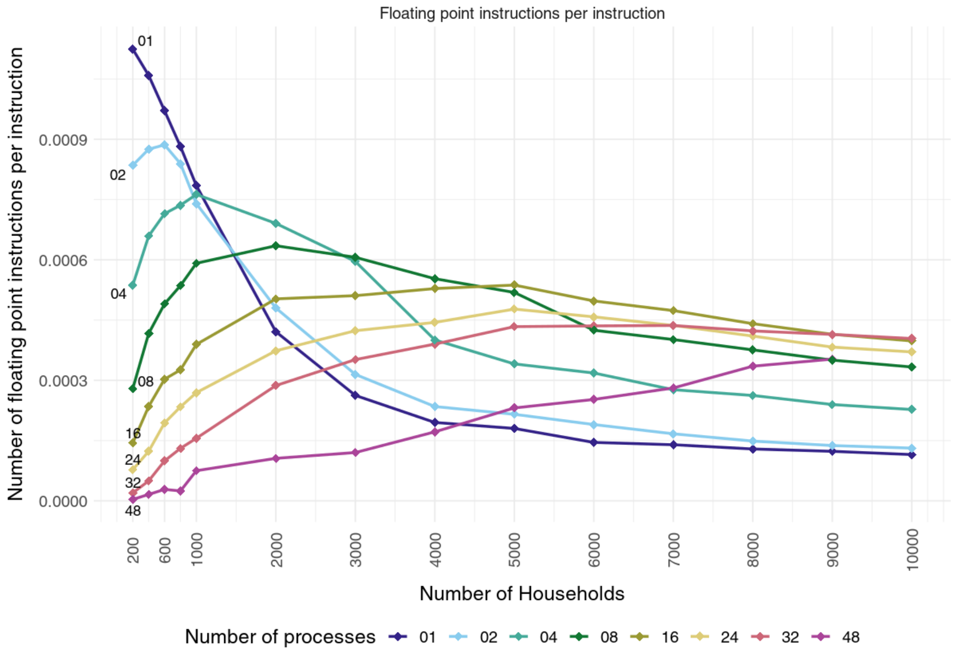

Figure 4.20: Floating point intensity

A household agent for example, the type of agent with the highest occurrence, consists of 21 variables (mostly doubles) and three arrays, each again five doubles. So, in sum, the struct of household agent contains 46 variables. But only a fraction of these variables are really needed for the execution of the transition function. E.g., when the household looks for work, it accesses three agent memory variables (ID, Employment status and current employer (if any)) and two environment variables (total number of firms in the simulation and the number of allowed firm interactions); when the household calculates its taxes, it accesses eight variables such as current income, received dividends and so forth. Most transition functions use only a very small subset of the agents' variables.

So, by constantly requiring to load the whole agent structure into memory, the program moves an enormous amount of data through the caches and is not able to keep the CPU busy[17].

Now, this might also explain why we see such a high occurrence of superlinear speedups. Since the application is bound the amount of data it can move through the caches, more processes can split the burden *if* they have access to their own hardware ressources. On Lewis, this is – at least partly – the case[18]: Each core has its own L1 and L2 cache and shares the L3 cache with six other cores. These six cores have their own non-uniform memory access (NUMA). The parallelized application is able to utilize this additional bandwidth and can, therefore, achieve the very high speedups we saw in the previous chapter.

This would also explain the patterns we observed at the cache misses (figures 4.5, 4.9 and 4.14): when the number of processes is low, the number of agents and thus also the amount of necessary data movement per process is high. This leads subsequently to a higher rate of cache misses when using a low number of processes.

Furthermore, when we studied the results of activated hyperthreading in figure 4.16, we saw that additional processes did not accelerate the simulation. Since these additional processes do not have access to any additional hardware ressources, we see that more processes do *not* yield further decreases in computation time. So the results for hyperthreading can also be explained by the inefficient memory design.

[17]This would also explain why we see the falling MFLOPS/s in figure 4.8. As the problem size increases, the amount of data that needs to be moved through the caches grows and less work gets done.

[18]It is obviously true for the VSC, where each compute node has its own memory, caches and processors.

Chapter 5

Conclusion

We wanted to test the performance of an agent-based macroeconomic model, implemented with the FLAME framework, on three different machines: a commodity hardware laptop, a multicore machine and the Vienna Scientific Cluster. The sobering result of this research is that the performance of the model (e.g. measured in MFLOPS/s) is rather disappointing.

On the multicore machine called Lewis, 48 processes achieve 18.2 MFLOPS/s where 400 GFLOPS/s would be possible. The results on the VSC in a similar range. On the other hand, we also see very high speedups, even super-linear speedups are quite common.

The reason for both results – the bad performance and the high speedups – could lie in the fact that the whole agent instance is loaded to execute a transition function. In the tested economic simulation, an agent tends to be rather big in terms of the number of variables in its memory. This leads to an excessive amount of data being loaded and stored as a transition function *rarely* needs accesss to *all* variables. During the execution of the simulation, the machine is almost solely occupied with transferring data in and out of the caches. This leads to the observed poor performance.

When using additional process running on their own hardware, we see super-linear speedups as the data transfer is split on different processes and hardware.

What does this mean for FLAME? Even though the L in FLAME stands for Large-scale, FLAME seems ill-prepared for handling large-scale simulation when the agents themselves hold a large memory. The design idea of FLAME is very appealing and implementing an ABM-SFC model is relatively easy but unfortunately it fails to deliver in practice. There is currently a redesign of FLAME, FLAME II, under way. The issue of excessive data transfer will also be alleviated[1]: To ensure a more efficient (in terms of concurrency) scheduling of the transition functions, FLAME II will require the modeller to indicate the memory variables a transition function needs. This would allow a scheduler to create a graph of memory dependencies and execute the transition functions as dependencies are met. Additionally, this should allow for less data transfer, as the framework then exactly knows which transition functions will need which memory addresses.

[1]See Chin et al. (2012) for a discussion of the planned improvements.

Appendix A

Detailed results

A.1 Additional results on laptop

A.1.1 Performance comparison with optimized compiler flags

A comparison of the results with and without hardware specific compiler flags on the laptop is presented below.

We can see that there are differences between the two compilation configurations (`-O3` and `-O3 -march=native -ffast-math -funroll-loops`). The compilation with the hardware-specific flags fares slightly better. The differences look small, but this is also due to the fact that the y-axis is on a log scale.

A.1.2 Measured instructions

Figure A.1.2 gives the measured "retired instructions" on the laptop. We can see that the results are proportional to the measured execution times that were given in figure 4.4.

A.1.3 Performance metrics based on measured instructions

As already mentioned in section 4.3.1.2, the results for redundancy and utilization are highly dubious, as the underlying measure for the completed work, retired instructions, is an imperfect replacement for floating point operations. Thus we see values for redundancy smaller than one for problem

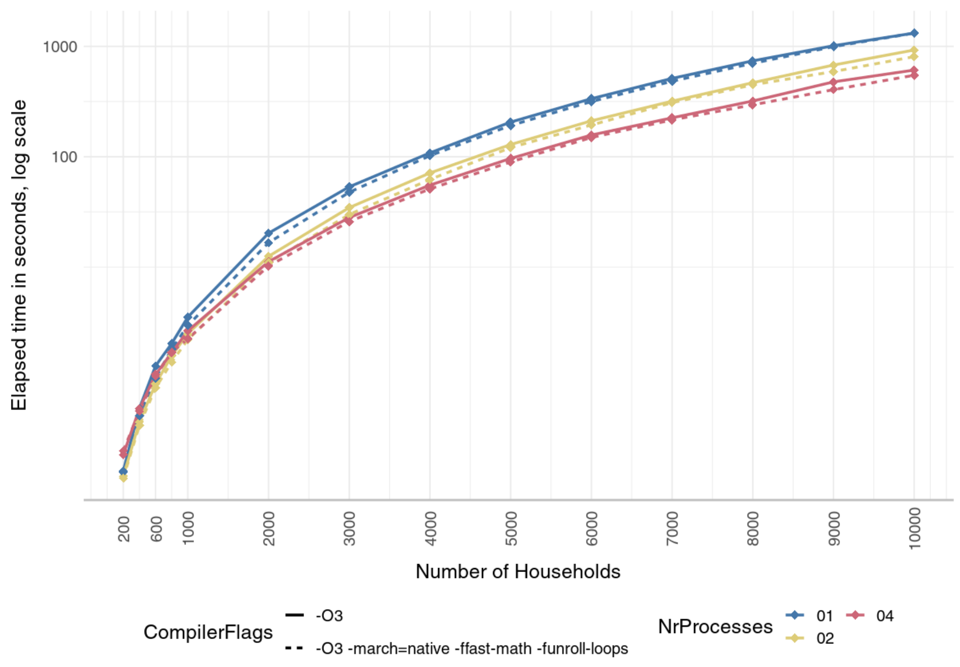

Figure A.1.1: Elapsed time comparison on laptop, different compiler optimizations

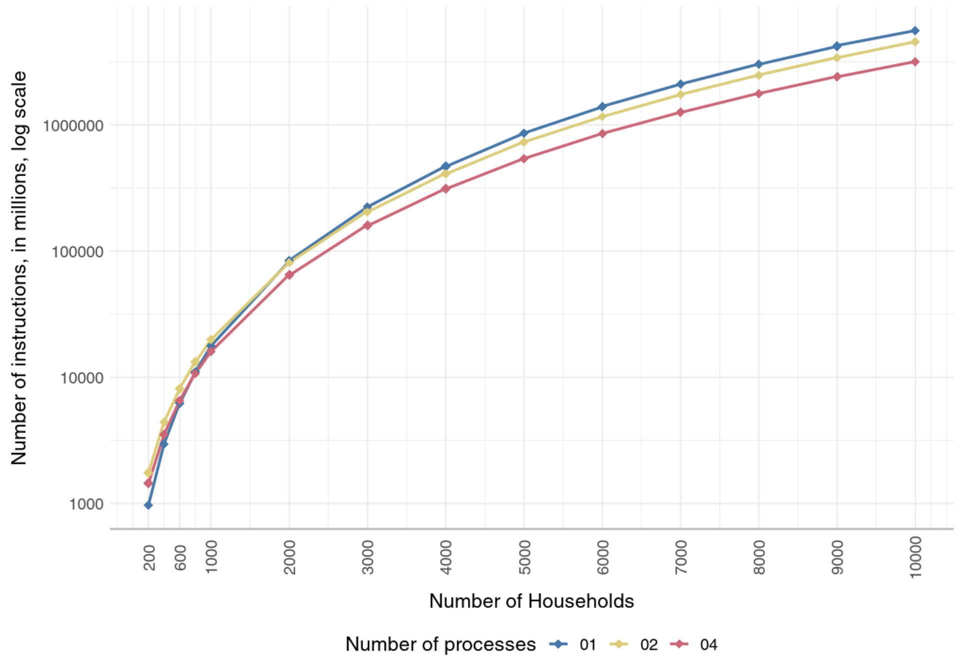

Figure A.1.2: Retired instructions on the laptop

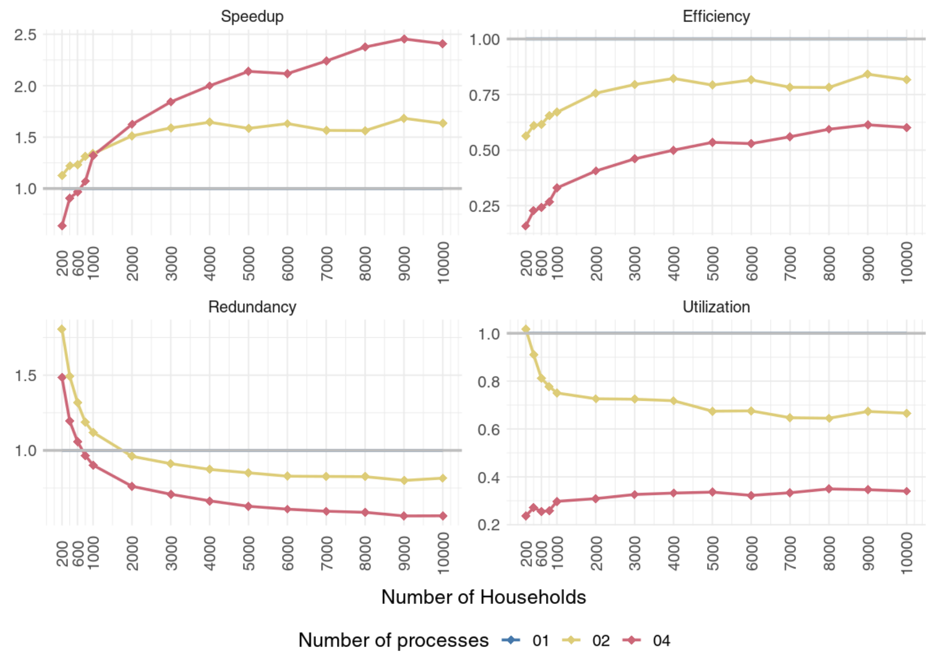

Figure A.1.3: Performance metrics on laptop

sizes bigger than 2000 households, while we would expect the opposite: values larger than one indicating the additional work that has to be done for the parallelization of the simulation.

A.2 Additional results on VSC

A.2.1 Performance metrics, all compilers

Figure A.2.4 compares the performance metrics for all three compiler/MPI library combinations.

We see that all configurations achieve positive speedup compared to the reference of one compute node. The speedups with OpenMPI 3 and Intel-MPI tend to be higher than the achieved speedups from OpenMPI 2. This is of course due to the relatively worse performance of OpenMPI 3 and Intel-MPI with one compute node and the relatively better performance with two and more compute nodes.

With OpenMPI 2 we do not see any super-linear speedups (i.e., efficiencies larger than one), while OpenMPI 3 and Intel-MPI show super-linear speedup when using 32 or 64 cores. The achieved efficiencies at 45000 households are 1.25 for OpenMPI 3 and 1.25 for Intel-MPI.

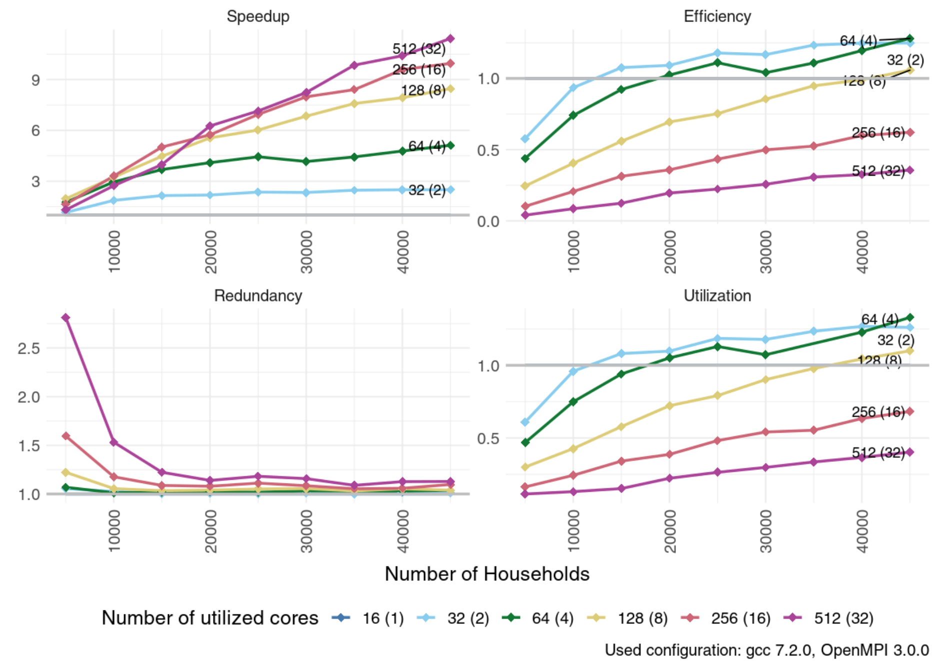

Figure A.2.4: Performance metrics on VSC

A.2.2 Load imbalance, all compilers

The FLAME code that was compiled with gcc 7.2.0 and OpenMPI 3 shows the highest levels of imbalance. It is interesting to note that OpenMPI 2 showed relatively high values of MFLOPS/s (compared to the other two versions, shown in figure 4.13), but exhibits here smaller values of load imbalance.

A.3 Flags activated by gcc and `-march=native`

A.3.1 On the laptop

```
/usr/lib/gcc/x86_64-pc-linux-gnu/8.2.0/cc1 -v help-dummy -march=haswell
-mmmx -mno-3dnow -msse -msse2 -msse3 -mssse3 -mno-sse4a -mcx16 -msahf
-mmovbe -maes -mno-sha -mpclmul -mpopcnt -mabm -mno-lwp -mfma -mno-fma4
-mno-xop -mbmi -mno-sgx -mbmi2 -mno-pconfig -mno-wbnoinvd -mno-tbm -mavx
-mavx2 -msse4.2 -msse4.1 -mlzcnt -mno-rtm -mno-hle -mrdrnd -mf16c
-mfsgsbase -mno-rdseed -mno-prfchw -mno-adx -mfxsr -mxsave -mxsaveopt
-mno-avx512f -mno-avx512er -mno-avx512cd -mno-avx512pf -mno-prefetchwt1
-mno-clflushopt -mno-xsavec -mno-xsaves -mno-avx512dq -mno-avx512bw
-mno-avx512vl -mno-avx512ifma -mno-avx512vbmi -mno-avx5124fmaps
```

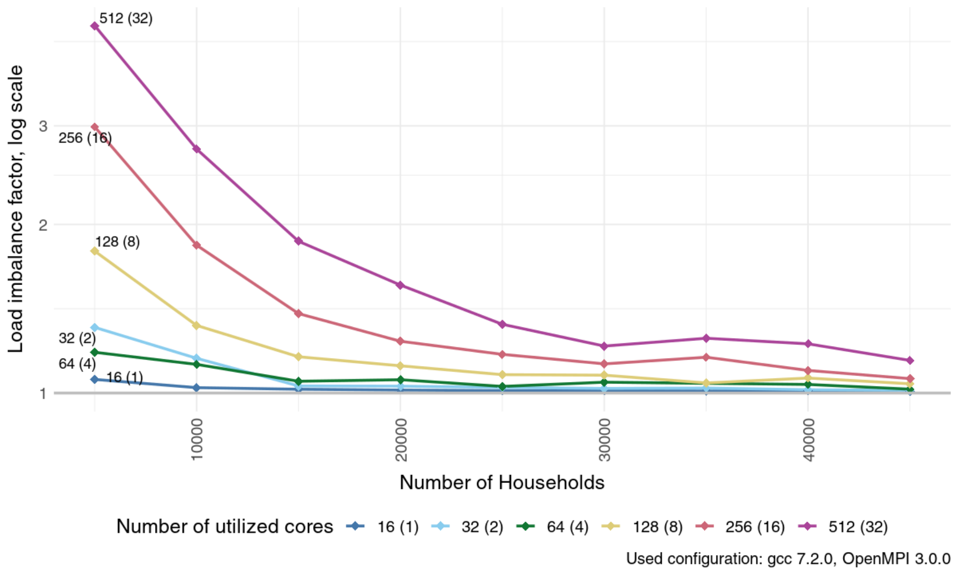

Figure A.2.5: Load imbalance

```
-mno-avx5124vnniw -mno-clwb -mno-mwaitx -mno-clzero -mno-pku -mno-rdpid
-mno-gfni -mno-shstk -mno-avx512vbmi2 -mno-avx512vnni -mno-vaes
-mno-vpclmulqdq -mno-avx512bitalg -mno-movdiri -mno-movdir64b
--param l1-cache-size=32 --param l1-cache-line-size=64
--param l2-cache-size=3072 -mtune=haswell -dumpbase help-dummy
-auxbase help-dummy -version --help=target
```

A.3.2 On Lewis

```
/usr/lib/gcc/x86_64-linux-gnu/5/cc1 -v -imultiarch x86_64-linux-gnu
help-dummy -march=amdfam10 -mmmx -m3dnow -msse -msse2 -msse3
-mno-ssse3 -msse4a -mcx16 -msahf -mno-movbe -mno-aes -mno-sha
-mno-pclmul -mpopcnt -mabm -mno-lwp -mno-fma -mno-fma4 -mno-xop
-mno-bmi -mno-bmi2 -mno-tbm -mno-avx -mno-avx2 -mno-sse4.2
-mno-sse4.1 -mlzcnt -mno-rtm -mno-hle -mno-rdrnd -mno-f16c
-mno-fsgsbase -mno-rdseed -mprfchw -mno-adx -mfxsr -mno-xsave
-mno-xsaveopt -mno-avx512f -mno-avx512er -mno-avx512cd -mno-avx512pf
-mno-prefetchwt1 -mno-clflushopt -mno-xsavec -mno-xsaves
-mno-avx512dq -mno-avx512bw -mno-avx512vl -mno-avx512ifma
-mno-avx512vbmi -mno-clwb -mno-pcommit -mno-mwaitx
--param l1-cache-size=64 --param l1-cache-line-size=64
--param l2-cache-size=512 -mtune=amdfam10 -dumpbase help-dummy
-auxbase help-dummy -version --help=target -fstack-protector-strong
-Wformat -Wformat-security
```

A.3.3 On the VSC

```
cc1 -v help-dummy -march=ivybridge -mmmx -mno-3dnow -msse -msse2
-msse3 -mssse3 -mno-sse4a -mcx16 -msahf -mno-movbe -maes -mno-sha
-mpclmul -mpopcnt -mno-abm -mno-lwp -mno-fma -mno-fma4 -mno-xop
-mno-bmi -mno-sgx -mno-bmi2 -mno-tbm -mavx -mno-avx2 -msse4.2
-msse4.1 -mno-lzcnt -mno-rtm -mno-hle -mrdrnd -mf16c -mfsgsbase
-mno-rdseed -mno-prfchw -mno-adx -mfxsr -mxsave -mxsaveopt
-mno-avx512f -mno-avx512er -mno-avx512cd -mno-avx512pf
-mno-prefetchwt1 -mno-clflushopt -mno-xsavec -mno-xsaves
-mno-avx512dq -mno-avx512bw -mno-avx512vl -mno-avx512ifma
-mno-avx512vbmi -mno-avx5124fmaps -mno-avx5124vnniw -mno-clwb
-mno-mwaitx -mno-clzero -mno-pku -mno-rdpid --param l1-cache-size=32
--param l1-cache-line-size=64 --param l2-cache-size=20480
-mtune=ivybridge -dumpbase help-dummy -auxbase help-dummy
-version --help=target
```

References

Abar, S. et al., 2017. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24, pp.13–33. Available at: http://www.sciencedirect.com/science/article/pii/S1574013716301198.

Antelmi, A. et al., 2015. On evaluating graph partitioning algorithms for distributed agent based models on networks. In *Euro-par 2015: Parallel processing workshops*. Cham: Springer International Publishing, pp. 367–378.

Bandini, S., Manzoni, S. & Vizzari, G., 2009. Agent based modeling and simulation: An informatics perspective. *Journal of Artificial Societies and Social Simulation*, 12(4), p.4.

Bersini, H., 2012. UML for ABM. *Journal of Artificial Societies and Social Simulation*, 15(1), p.9.

Caiani, A., Catullo, E. & Gallegati, M., 2017. The effects of fiscal targets in a monetary union: A multi-country agent-based stock flow consistent model. *Industrial and Corporate Change*.

Caiani, A. et al., 2016. Agent based-stock flow consistent macroeconomics: Towards a benchmark model. *Journal of Economic Dynamics and Control*, 69, pp.375–408.

Caiani, A., Russo, A. & Gallegati, M., 2017. Does inequality hamper innovation and growth? An AB-SFC analysis. *Journal of Evolutionary Economics*, pp.1–52.

Carothers, C. et al., 2017. Computational challenges in modeling and simulation. In *Research challenges in modeling and simulation for engineering complex systems*. Springer, pp. 45–74.

Chin, L. et al., 2012. FLAME-II: A redesign of the flexible large-scale agent-based modelling environment. *STFC Research publications*. Available at: http://purl.org/net/epubs/work/64112.

Coakley, S. et al., 2012. Exploitation of high performance computing in the flame agent-based simulation framework. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference*. IEEE, pp. 538–545.

Coakley, S., Smallwood, R. & Holcombe, M., 2006. Using X-machines as a formal basis for describing agents in agent-based modelling. *Simulation Series*, 38(2), p.33.

Colander, D. et al., 2009. The financial crisis and the systemic failure of the economics profession. *Critical Review*, 21(2-3), pp.249–267.

Collier, N. & North, M., 2013. Parallel agent-based simulation with Repast for High Performance Computing. *Simulation*, 89(10), pp.1215–1235.

Collier, N., Ozik, J. & Macal, C.M., 2015. Large-scale agent-based modeling with repast HPC: A case study in parallelizing an agent-based model. In *European conference on parallel processing*. Springer, pp. 454–465.

Deissenberg, C., Van Der Hoog, S. & Dawid, H., 2008. EURACE: A massively parallel agent-based model of the European economy. *Applied Mathematics and Computation*, 204(2), pp.541–552.

Economist, 2010. Agents of change. *The Economist.* Available at: https://www.economist.com/node/16636121.

Epstein, J.M., 2009. Modelling to contain pandemics. *Nature*, 460(7256), p.687.

Epstein, J.M. & Axtell, R., 1996. *Growing artificial societies: Social science from the bottom up*, Brookings Institution Press.

Farmer, J.D. & Foley, D., 2009. The economy needs agent-based modelling. *Nature*, 460(7256), p.685.

Flynn, M.J., 1972. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9), pp.948–960.

Fujimoto, R.M., 2016. Research challenges in parallel and distributed simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(4), p.22.

Gerst, M.D. et al., 2013. Agent-based modeling of climate policy: An introduction to the engage multi-level model framework. *Environmental modelling & software*, 44, pp.62–75.

Godley, W. & Lavoie, M., 2007. *Monetary economics: An integrated approach to credit, money, income, production and wealth*, Springer.

Gualdi, S. et al., 2015. Tipping points in macroeconomic agent-based models. *Journal of Economic Dynamics and Control*, 50, pp.29–61.

Haldane, A.G. & Turrell, A.E., 2018. An interdisciplinary model for macroeconomics. *Oxford Review of Economic Policy*, 34(1-2), pp.219–251. Available at: http://dx.doi.org/10.1093/oxrep/grx051.

Heywood, P., Richmond, P. & Maddock, S., 2015. Road network simulation using FLAME GPU. In *European conference on parallel processing*. Springer, pp. 430–441.

Kiran, M. et al., 2008. Porting of the software platform to parallel computer. *Project EURACE Deliverable D8.4*. Available at: http://www.wiwi.uni-bielefeld.de/lehrbereiche/vwl/etace/Eurace_Unibi/Eurace_Deliverables.

Kiran, M. et al., 2010. FLAME: Simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1*. International Foundation for Autonomous Agents; Multiagent Systems, pp. 1633–1636.

Konur, S. et al., 2015. Agent-based high-performance simulation of biological systems on the GPU. In *High performance computing and communications (HPCC), 2015 IEEE 7th international symposium on cyberspace safety and security (CSS), 2015 IEEE 12th international conference on embedded software and systems (ICESS), 2015 IEEE 17th international conference*. IEEE, pp. 84–89.

Korinek, A., 2015. Thoughts on DSGE macroeconomics. *A Just Society: Festschrift in Honor of Joseph Stiglitz*.

Lengnick, M., 2013. Agent-based macroeconomics: A baseline model. *Journal of Economic Behavior & Organization*, 86, pp.102–120.

Lysenko, M. & D'Souza, R.M., 2008. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4), p.10.

Makarov, V.L., Bakhtizin, A.R. & Bakhtizina, N.V., 2014. Application of supercomputer technologies in agent-based models. In *5th. World congress on social simulation.* p. 96.

Massaioli, F., Castiglione, F. & Bernaschi, M., 2005. OpenMP parallelization of agent-based models. *Parallel Computing*, 31(10-12), pp.1066–1081.

Márquez, C., César, E. & Sorribes, J., 2013. A load balancing schema for agent-based spmd applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).* The Steering Committee of The World Congress in Computer Science, Computer Engineering; Applied Computing (WorldComp), p. 12.

Márquez, C., César, E. & Sorribes, J., 2015. Graph-based automatic dynamic load balancing for HPC agent-based simulations. In *Euro-par 2015: Parallel processing workshops.* Cham: Springer International Publishing, pp. 405–416.

Nature, 2009. A model approach. *Nature*, 460, p.667 EP. Available at: http://dx.doi.org/10.1038/460667a.

Parker, J. & Epstein, J.M., 2011. A distributed platform for global-scale agent-based models of disease transmission. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1), p.2.

Parry, H.R. & Bithell, M., 2012. Large scale agent-based modelling: A review and guidelines for model scaling. In *Agent-based models of geographical systems.* Springer, pp. 271–308.

Reed, D.A. et al., 2005. *Computational science: Ensuring America's competitiveness*, President's Information Technology Advisory Committee Arlington.

Richmond, P., Coakley, S. & Romano, D., 2009. Cellular level agent based modelling on the graphics processing unit. In *High Performance Computational Systems Biology, 2009. HIBI'09. International Workshop.* IEEE, pp. 43–50.

Richmond, P. et al., 2010. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3), pp.334–347.

Rousset, A. et al., 2014. A survey on parallel and distributed multi-agent systems. In *European Conference on Parallel Processing.* Springer, pp. 371–382.

Schasfoort, J. et al., 2017. Monetary policy transmission in a macroeconomic agent-based model. *SOM Research Reports*, 17010.

Smets, F. & Wouters, R., 2003. An estimated dynamic stochastic general equilibrium model of the Euro area. *Journal of the European Economic Association*, 1(5), pp.1123–1175.

Tang, W. & Bennett, D.A., 2011. Parallel agent-based modeling of spatial opinion diffusion accelerated using graphics processing units. *Ecological modelling*, 222(19), pp.3605–3615.

Terpstra, D. et al., 2010. Collecting performance data with PAPI-C. In M. S. Müller et al., eds. *Tools for High Performance Computing 2009.* Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 157–173.

Tesfatsion, L. & Judd, K., 2006. *Handbook of Computational Economics, Vol. 2: Agent-Based Computational Economics*, Iowa State University, Department of Economics.

Tisue, S. & Wilensky, U., 2004. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*. Boston, MA, pp. 16–21.

Wang, F.X., Liu, S.Z. & Deng, T.S., 2014. Agent-based mood spread diffusion model for GPU. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*. pp. 1056–1059.