A Comparison of Languages and Frameworks for the Parallelization of a Simple Agent Model

Stefan McCabe[1], Dale Brearcliffe[1], Peter Froncek[1], Marta Hansen[1], Vince Kane[1], Davoud Taghawi-Nejad[2], and Robert L. Axtell[1]

[1] Computational Social Science Program, Department of Computational and Data Sciences
George Mason University, Fairfax, Virginia 22030 USA
{smccabe2, dbrearcl, pfroncek, mhansen6, vkane2, rax222}@gmu.edu

[2] Center for Metropolitan Studies
University of São Paulo, São Paulo, Brazil
davoud@taghawi-nejad.de

Abstract. The zero-intelligence trader model is a well-known agent-based representation of supply and demand relations for the study of the emergence of market clearing. While conventional implementations of it employ a variety of agent activation schemes, essentially all are single-threaded. We have parallelized this model using a variety of languages and technologies, including pthreads in C, C++11 threads, OpenMP running under C, Clojure, Erlang, Go, Haskell, Java, Python, and Scala. For each, speed-up as a function of the number of threads is studied on a high-performance computing platform. Most of these software systems demonstrate significant performance gains over single-threaded implementations and are capable of taking advantage of a large number of hardware cores. The advantages and disadvantages of each approach are described and compared. Some of these languages are largely functional, some feature persistent data, and some have relatively poor absolute performance. We discuss the utility of them for multi-agent systems generally, including ease of programming and likely performance improvements on multi-core hardware.

Keywords. Parallel agent execution; zero-intelligence trader model; pthreads; C++11 threads; OpenMP; Clojure; Erlang; Go; Haskell; parallel Python; Scala.

1 Parallel Execution of Agent Models

With the proliferation of multi-core hardware and the rise of massive multi-agent systems (MAS) [1] and large-scale agent-based models (ABMs) [2], it is natural to ask how agent software can be made to efficiently run in parallel. While parameter sweeps of conventional MAS codes are 'embarassingly parallel,' efficiently executed on conventional hardware by placing each model instance on its own thread, it is also the case that parallelization of single model instances is desirable when individual runs are computationally expensive. However, conventional software technologies for parallelization, many of them born in the natural sciences, may not be appropriate for MAS. Specifically, parallelization schemes that focus on distributing nested loops

MABS

over multiple threads, processes or cores may not be either an effective or natural way to speed-up agent systems. Our codes often feature densely interacting populations, and message passing between threads/processes/cores/machines may be an order of magnitude slower than placing sub-populations of agents most likely to interact with one another on the same thread or process. For these reasons, specific high-performance computing (HPC) architectures, like vector supercomputers and clouds, may be inappropriate for large-scale agent computing. Indeed, our experience is that storing a large agent population on a single machine in a big, 'flat' memory space often produces higher performance for agent systems than more conventional HPC.

The societal metaphor for agent computing—to wit, each agent autonomous and therefore acting as its own process—means that agent models *should be* naturally parallelizable. Once we recognize that human societies are the result of myriad local, decentralized actions and interactions of large numbers of people, we should be able to faithfully represent this. Partitioning a population of agents into sub-populations so that each can run on its own thread/process/core for some amount of time—effectively the fork/join model of parallel computation—is a very accessible way to convert a serial/single-threaded agent model into parallel code.

An immediate problem facing researchers interested in making their agent models run in parallel is that there is, today, no dominant technology for implementing fork/join or related styles of parallelism, no single best hardware platform or software engineering practice. Rather, there is a large number of competing technologies, such as Cilk, CO2P3S, Eden, FastFlow, HDC, MPI, OpenMP, P3L, PAS, Skandium, SkePU/SkeTo, and Threading Building Blocks [3] on the software side, and hardware involving graphics processing units (GPUs) and coprocessors (e.g., Intel's Xeon Phi). This is a kind of alphabet soup of technologies, many of which have not been systematically applied to agent systems. Navigating this wide range takes one from low-level system (e.g., shaders, pthreads) to higher level constructs like dispatch queues and software libraries/frameworks for specific types of hardware, such as CUDA and FLAME [4, 5] for GPUs.

What most of these technologies have in common is the notion of breaking up program execution streams running on one thread into smaller pieces that can run in parallel on multiple threads—potentially hundreds or thousands of threads. If each thread can run largely autonomously, without having to interact with or wait on results from other threads, then large speedups can be achieved in principle. In writing agent codes to execute in parallel we are, it seems to us, returning our models to the more realistic pattern of concurrent, largely asynchronous execution that characterizes real human societies. When you get up in the morning you do not have to wait to drink your coffee until someone else, in a possibly distant part of the world, has had their breakfast. Or when you come to a stop sign you may be obliged to wait for the car that has arrived at the opposite corner, but you are not *physically forced* to do so—as an autonomous agent you can either comply with local social and legal norms or not, and in any case your decision processes are not running *sequentially* with the other drivers, rather they are running in parallel, mostly asynchronously. That is, the real social world is very far from the single-threaded execution model that the vast majority of MAS models rely on, and that conventional agent modeling software frameworks

(e.g., NetLogo, Repast and Mason) employ. Surely the main rationale for coding an ABM on a single-thread is *convenience*, not realism. Our main goal here is to explore the process of moving from one to many threads in hopes of uncovering 'good' ways of doing so, fully expecting that some ways will be harder to code, some will have higher performance, and so on.

Here we will not exhaustively characterize all the software systems available for parallelizing agent models, for this would surely be a formidable task. Rather, we will exercise some relatively well-known languages and parallel frameworks in the context of a particular agent model and study the nature of the speedup that is possible on a high-performance, multiple core workstation.

2 Zero-Intelligence Traders

The zero-intelligence (ZI) trader model derives from the attempts of Gode and Sunder [6, 7] to reproduce the behavior of individual buyers and sellers in a financial asset market. They began their efforts by coding quite complicated agents but eventually realized that relatively simple, almost minimal specifications could do as good a job reproducing their data as more complex behavioral models. Specifically, models in which buyers paid less than some value ceiling while sellers tried to cover their costs, with haggling between these two limits, not only explained much of what they observed, it also produced reasonable market performance overall. Indeed, so near to peak performance were their ZI markets that they suggested it was the market *institutions* that were mainly responsible for the overall high performances of such traders, not the ratiocination of the individual traders themselves, since this was so minimal.

Subsequently, a variety of authors showed that the simplest ZI traders displayed pathologies that needed to be corrected in order to explain other kinds of market behavior, thus giving rise to 'zero-intelligence plus' (ZIP) traders [8, 9] and related notions. However, these enhanced ZI traders also have relatively simple behavior. As our goal here is primarily to explore the relative performance of distinct software environments running on common hardware, there is no need for agents more sophisticated than the original ZI traders. Pseudo-code for this model is:

- **INSTANTIATE and INITIALIZE BUYER, SELLER, DATA and THREAD objects;**
- **Assign sub-populations of BUYERS and SELLERS to THREADS;**
- **FORK all THREADS;**
- **FOR each THREAD, REPEAT:**
  - **Randomly activate 1 BUYER agent + 1 SELLER agent:**
    - **BUYER proposes a BID price;**
    - **SELLER proposes an ASK price;**
    - **IF (BID > ASK) THEN**
      - **Pick EXECUTION price between BID and ASK;**
      - **INCREMENT BUYER holdings;**
      - **DECREMENT SELLER holdings;**
      - **Collect DATA on the trade;**
  - **INCREMENT the attempted number of trades;**
  - **END when maximum trade attempts exceeded;**
- **JOIN all THREADS;**
- **Collect final DATA;**

Clearly, there will be dense interactions between buyer and seller agents intra-thread, while communication between threads is minimal. The sub-populations can be periodically reformed during execution but in what follows this proved inessential. In the next section we shall work with several model instances, of different sizes, and vary the number of threads for each, studying the amount of speedup achieved.

3 Codes for Parallel ZI Traders

We have written code for ZI Traders over three orders of magnitude of model sizes:

1. 10K buyers and 10K sellers who attempt up to 1 million trades;
2. 100K buyers and 100K sellers who try to trade 10 million times;
3. 1 million buyers and 1 million sellers, with 100 million attempted trades.

All model runs were conducted on a Microway workstation having 256 GB of RAM, dual Intel Xeon processors (E5-2687W with 8 cores/processor for a total of 16 hardware cores, each core having a 20.48 MB cache) running at 3.10 GHz under Fedora Linux. The workstation is also outfitted with an NVIDIA GTX 980 although this was not used in the present study. This hardware proved sufficiently capable that for mature languages (e.g., C) even the largest model instance—#3 above—executed in less than a second of wall time when all cores were utilized.

3.1 C and Pthreads

An existing single-threaded ZI Trader model was parallelized by importing the POSIX thread (pthread) library and forking the main execution stream of the code, joining everything together at the end before collecting statistics. This code was the fastest we produced, unsurprisingly, given the somewhat low level nature of pthreads. Figure 1 shows speedups as a function of the number of threads utilized, for each of the model sizes—the dotted line is 10K buyers and sellers, the dashed line is 100K agents of each type, and the solid line is 1 million. (Plotted values are means over 10 repetitions for each parameter setting. Standard deviations are small and not shown.)

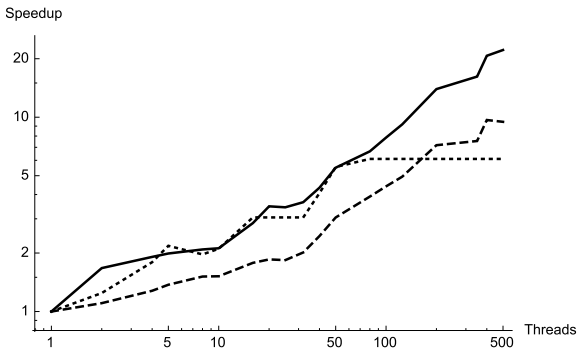

Fig. 1: Speedup in C using pthreads: 10K buyers and 10K sellers (dots); 100K each type (dashed); 1 million each (solid)

There are several things to note about this figure. First, invoking additional threads always led to greater performance (less execution time than the single threaded model), although beyond about 50 threads the small model's performance plateaus. The maximum speedup is below 10x for the small model, about 10x for the medium model, and about 20x for the large model. Note also that at least for the largest problem, and perhaps for the medium one, 500 threads may fall below the maximum speedup, i.e., the speedup curve has not reached its maximum in the figure. Indeed, further experimentation with this code has revealed that the maximum speedup seems to occur for somewhat more than 2000 threads, after which performance drops off significantly. Finally, note that in the case of the largest problem (solid line) the speedup seems to be superlinear, in the sense that there is more speedup than one might reasonably expect on a machine with 16 hardware cores. While more diagnostic work is necessary to pinpoint the exact nature of this superlinearity, we believe that it results from a large number of threads producing smaller agent populations/thread, and therefore the progressive ability of threads having smaller memory footprints to benefit from the level 2 and 3 caches. In essence, and this will have to be checked, the smaller sub-populations produced by larger numbers of threads run more quickly and easily (i.e., without being interrupted) and this makes the whole code terminate sooner.

3.2 C++11 Threads

Threads are part of the C++11 standard, meaning multithreaded code written in this language is more portable than pthreads. The *thread* class abstractions are somewhat higher level in C++11 in comparison to pthreads, making them simpler to use. Specifically, it was easier to code this version of ZI Traders than the pthreaded one. While C++11 threads may be implemented in pthreads on some compilers, we found important performance differences from previous subsection. Figure 2 plots speedups for this code as a function of the number of cores, for each of the three problems.

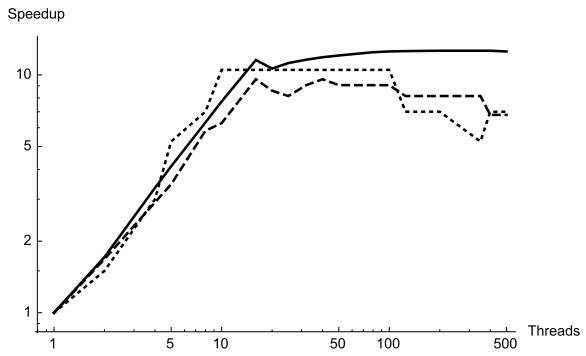

Fig. 2: Speedup using C++11 threads: 10K (dots); 100K (dashes); 1 million (solid)

This first thing to notice is that the speedups are about the same for all three problems. Second, each code is sped up about 10x, a little more for the larger model, a little less for the smaller model. Third, each model hits its peak performance between 10 and 15

threads, which is quite close to the number of hardware cores. Lastly, note that there is no superlinear speedup here.

3.3 OpenMP under C

OpenMP [10] and MPI [11] are older frameworks, initially designed for parallelization across processors or machines, primarily of numerical codes. For example, executing nested loops and related linear algebra type applications are efficiently parallelized in OpenMP. For our agent models we used the parallel FOR pragmas which, starting from the single threaded version, made this implementation the simplest to write. Running the resulting codes we obtained the speedups shown in figure 3.

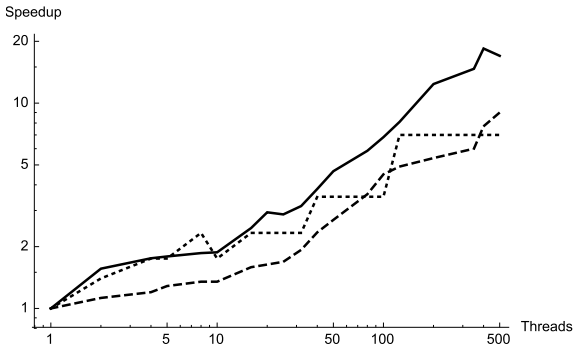

Fig. 3: Speedup using OpenMP in C: 10K (dots); 100K (dashes); 1 million (solid)

This code turned out to be about as fast as the pthreaded code (see section 4 below for direct comparison). Note the similarity of figure 3 to figure 1 and its dissimilarity to figure 2. The optimal number of threads for OpenMP is greater than 500 for the two larger problems, similarly to the pthreaded code. Speedup for the largest model also appears to be superliner (i.e., greater than the 16x level one might expect from the hardware used), and we suspect this is again due to the role of the CPU caches.

3.4 Java Threads

The ZI Java code has been parallelized using native Java threading [12] in the fork/join pattern employed previously. These threads are easier to program than pthreads, closer to C11 threads, but not as simple as OpenMP. In writing this code we created two distinct versions of it with different kinds of locks. The speedups achieved are shown in figure 4, with one code corresponding to the darker lines and the other represented by the lighter ones. Note that for the small and medium size problems there is essentially no difference in speedup between the codes. However, for the large problem there is a sizeable difference between the implementations. The optimal number of threads is somewhere in the 10-15 range for most of these Java codes, but is substantially higher (>100) for the implementation solving the largest problem instance.

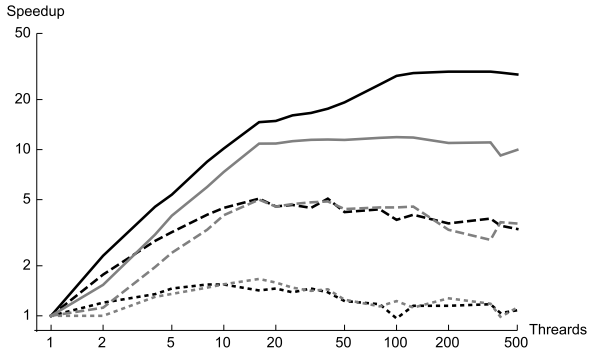

Fig. 4: Speedup using Java threads: 10K (dots); 100K (dashes); 1 million (solid)

As we shall see in the next section, the absolute performance of these Java codes is inferior to the C and OpenMP, versions, but can outperform C++ for large problems.

3.5 Clojure

Clojure is a dialect of Lisp that was created by Rich Hickey and released in 2007. A new version of Clojure has been released each year from 2009 through 2015. Hickey wrote Clojure because he wanted a Lisp for functional programming (FP) that ran on the Java Virtual Machine (JVM). He also wanted concurrency and data immutability functions that were not present in existing dialects of Lisp.

Clojure runs as a compiled language on the JVM (Java 5 or later) using a just-in-time process.[1] The JVM is an abstraction of the hardware layer enabling Clojure to function wherever the JVM platform exists. Interoperability with Java is provided, permitting Clojure to make use of Java libraries [13]. Concurrency was a major design goal for the language. The immutability of Clojure's core data structures facilitate sharing of them between threads without the overhead of manually implementing locks. State can change and Clojure provides several mechanisms for sharing these changes between threads. Lazy evaluation permits a function to be defined but not evaluated until its value is a computational necessity. There are implementations of Clojure that target platforms other than the JVM. Two that are directly supported by Clojure are ClojureCLR for Microsoft's.Net framework and ClojureScript for JavaScript.

Any cell based ABM, or one in which the agents can be represented in an array or vector, should be functionally compatible with Clojure. The ZI model agents can be represented in a vector of vectors. Because our target is a functional language it is important to refactor the pthreaded C code into idiomatic Clojure rather than porting it line by line. This requires an in-depth understanding of the C code so that the *same functionality* is recreated in Clojure. In coding ZI Traders in Clojure it was found that

[1] Clojure can also be pre-compiled to an executable.

writing each function from the inside out was easier than creating the code from the outside in. The resultant code is aesthetically pleasing to view.

Figure 5 shows the results of running the ZI trader Clojure code on the small, medium and large problems, with different numbers of threads. The results show the maximum benefit occurred at 2 threads.

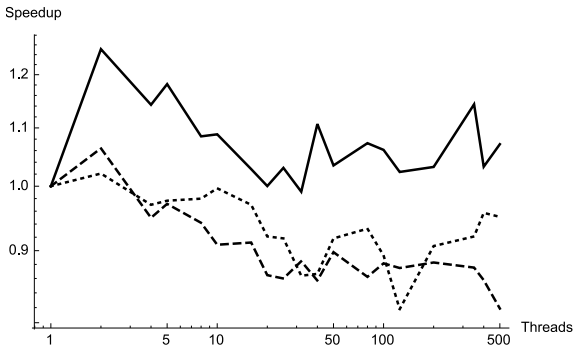

Fig. 5: Speedup in Clojure: 10K (dots); 100K (dashes); 1 million (solid)

While Clojure achieves some speedup initially the shape of the lines in this graph are qualitatively unlike what has come before. As the number of threads increases the speedup declines.

3.6 Go

This language is a compiled, statically-typed, garbage-collected language with C-style syntax that emphasizes simplicity. It has built-in concurrency features inspired by Hoare's [14] CSP language. Its first major release, Go 1.0, was in 2012; the most recent version is Go 1.6.1. It was developed at Google by Robert Griesemer, Rob Pike and Ken Thompson.

Concurrency is implemented in a straightforward fashion using *goroutines* (coroutines) and channels. Goroutines represent a concurrent process, analogous to a thread but higher-level. Tang [15] reports very little overhead in using goroutines for a parallel integration task, with each goroutine taking approximately 10 microseconds to launch. Channels are first-class objects used for message-passing between goroutines, allowing parallel programs to be written without dealing with the traditional pitfalls associated with shared memory (mutexes, etc.).

Go seems to be very well-suited to the development of certain types of agent-based models. It performs reasonably well relative to C and can be faster than basic Python. One potential frustration, its static typing, is less of a problem in ABM development, where model parameters should already be well-specified. Its approach to object-oriented programming also seems well-suited to ABMs; implicit interfaces and methods on `structs` are convenient for developing simple agents, where type hierarchy is typically unnecessary. Its utility is currently limited by poor library support. There are some basic statistical libraries but very few libraries for visualization, limiting its use

to economic models or other cases where visualization is deemphasized. ZI Trader code in Go has been created and typical speedups are shown in figure 8.

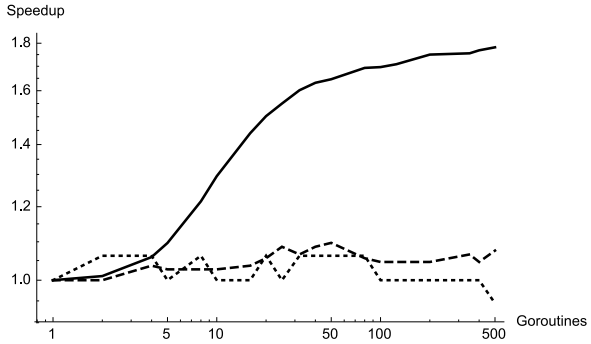

Fig. 6: Speedup in Go: 10K (dots); 100K (dashes); 1 million (solid)

For the smaller problem instances Go achieves little speed up, but for the largest problem it produces about 2x speedup and does this with a large number of goroutines.

3.7 Scala

This language, whose name is an acronym for "Scalable Language," combines object-oriented and FP with a strong static type system. It was designed by Martin Odersky in 2003 as an improvement to Javascript, and as such was built with the primary intent of addressing common criticisms of Java. Like Clojure, Scala uses the JVM as its execution platform and possesses language interoperability with Java, meaning that Java libraries, frameworks, and tools can all be used in Scala and vice versa. Besides simply addressing criticism of Java, Scala is intended to be "future-proof" in that it allows for concurrent and synchronous processing, parallel utilization of multiple cores, and distributed processing in the cloud. Its functional nature also allows it to perform well with multi-threaded code.

Scala is different from other programming languages in that it is remarkably easy to use, especially for those who have programmed in Java before. Scala is concise in syntax, resulting in code that is significantly shorter than its Java counterpart, and relatively safe with the majority of errors being caught at compile-time. It is also a programming language that strives for execution speed, especially since the collections operations were recently adapted for parallel execution on multi-cores. Scala has mutable and immutable collections, meaning there are guarantees that certain collections cannot be changed. All of this contributes to Scala being a prime programming language for creating models for parallel execution.

The similarity to Java made Scala very easy to learn and straightforward when writing the ZI Traders model with multi-threaded processes. However, Scala does not have a return type, which was occasionally confusing and requires close attentiveness to the model's functionality. The language interoperability with Java was also very convenient as it allowed for the use of Java's libraries without having to 're-invent the

wheel' in Scala. Further, although Scala is intended as an immutable language, it is flexible and can infer mutable code when necessary. Running the three problems in Scala generated the results depicted in figure 7.

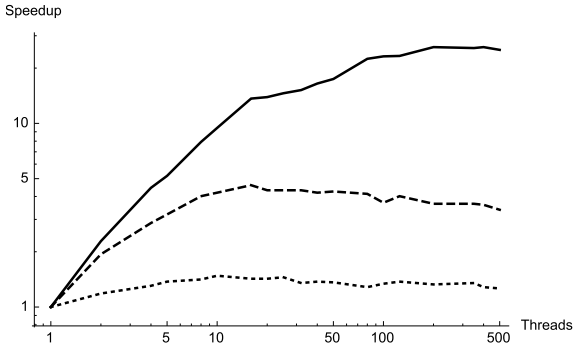

Fig. 7: Speedup in Scala: 10K (dots); 100K (dashes); 1 million (solid)

Good speedups were achieved for all but the smallest problem. Note the similarity of this figure to the Java results shown in figure 4.

3.8 Erlang

This language is a functional, dynamically-typed, garbage-collected programming language. It is compiled into bytecode and runs on a virtual machine. Erlang was originally developed by the Swedish telecommunications operator Ericsson for creating distributed, fault-tolerant, highly-available applications. Erlang is known for its error handling and focus on concurrency. The language uses the actor model to execute multiple processes concurrently. Individual processes do not share information apart from explicit message passing. This removes the need for using explicit locks for distributed processes. Erlang also possesses the ability to execute processes in parallel and provides its own libraries for managing threads.

While the language was built for concurrency, i.e., resolving simultaneous requests in a non-conflicting manner, it was not built for parallel processing in the beginning. The parallel processing capability was added to Erlang only recently [16]. Thus, Erlang is very useful for applications that manage communication between clients and servers. On the other hand, as with many functional languages, Erlang is not ideal for number processing and large-scale arithmetic operations. This has implications for developing agent-based models on the Erlang platform. It should be expected that agent-based models written solely in Erlang code will be much slower than their counterparts in other, mostly object-oriented languages, which is the case for our Erlang ZI Traders implementation. However, the parallel-processing power and the speedups seen when using multiple threads could potentially be harnessed and used in part for large ABMs running on computer clusters, where the interface for passing data between the networked components would be written in Erlang. In its single-threaded instantiations Erlang is relatively inefficient (more on this in section 4) and

so we have only run it for the first two (smaller) problems. Because it is so slow there is opportunity for large speedups as code is distributed on multiple cores. Figure 7 shows that such speedups are realizable in practice as the number of threads increases.

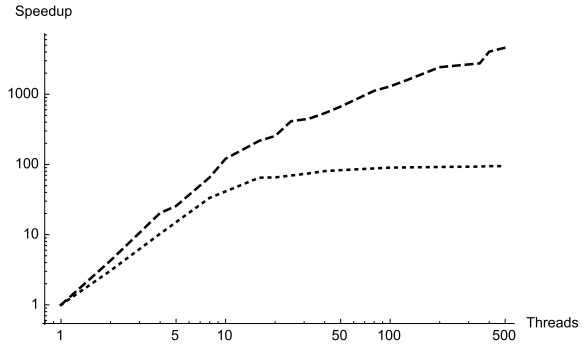

Fig. 8: Speedup in Erlang: 10K (dots); 100K (dashes)

This plot shows the most dramatic speedups we have seen, more than a thousand-fold decrease in execution time as all cores are utilized and the number of threads becomes large. This is truly superlinear speedup. While this enormous performance gain is mostly due to the single-threaded code running very slowly, the sheer magnitude of this effect invites further investigation. More work needs to be done to understand the origins of the superlinearity on display here.

3.9 Python

Python is a rapidly growing language, combining some of the best features of object-oriented languages with ease of use and reasonable performance. Unfortunately, its main implementations, CPython and pypy, do not allow more than one thread to run at the same time, and so we should not expect speedup for multiple threads. We have multithreaded the ZI Trader model in Python and figure 9 illustrates the results.

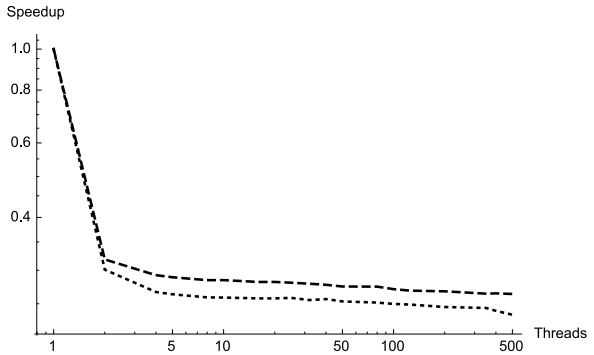

Fig. 9: Speedup using Parallel Python: 10K (dots); 100K (dashes)

Clearly, putting more threads to work under Python does *not* produce performance gains. The reason for this is Python's Global Interpreter Lock (GIL). The GIL is implementation specific and it is not a requirement of the Python language as such. Indeed, it is possible to release the GIL and execute several threads simultaneously, as when Numpy's C modules and certain I/O bound routines run at the same time, permitting simultaneous computation and user interaction. There are currently a variety of approaches to parallelizing Python and we expect to see developments in this direction in the future.

3.10 Haskell

This is an FP language. Haskell, and FP languages generally, are an extension of the lambda-calculus to specify computable functions. FP languages began with Lisp in the 1950s and since then their number has grown rapidly. In 1987, a team of researchers, concerned about this proliferation, and the obstacles it placed in advancing the use of FP as a paradigm, formed a committee to "crystallize" the myriad FP languages then in use and generate a common solution. The first version of Haskell was released in April 1990 (see [17] for a detailed history of the language and its capabilities). Haskell has the following key features:

- *Lazy evaluation*: In "lazy" languages, functions are called only when they are needed, not when they are referenced. The laziness property dictates many other features of the language, such as "purity".
- *Pureness*: Since demand-driven evaluation order interferes with reliable input and output, interactions with "the real world" are restricted, and programmers must clearly and consciously segregate the algorithmic specification of their application into code that performs input and output and the [pure] portions performing reliable and repeatable data transformations. This means that "pure" Haskell code always produces the same output on a given input.
- *Data typing and type classing*: Haskell's data typing is possibly one of the strongest among widely-used languages. Haskell also implements the polymorphic feature of type classing—a class of types share a common set of allowable operations.

Concurrency (multiple threads, for dealing with different I/O streams for example) and parallelism (performing the same operation in parallel) are provided through two separate extensions in Haskell. Agent-based modelers are more likely to need the parallelism extension for performance improvement. A Haskell code may be easily transformed into parallel operation by importing the parallel library and wrapping the `par` and `pseq` functions around the data transformations to be performed in parallel. Command-line directives to the compiler may also dictate various threading schemes.

For a variety of technical reasons we were unable to get a version of ZI Traders in Haskell up and running on the hardware used to generate all the figures above. Rather, our implementation was run on a single Intel Core i7-2600K processor having 4 cores. Typical speedups for this code are shown in figure 9 for all the three problem sizes: 10 thousand, 100 thousand, and 1 million each buyers and sellers.

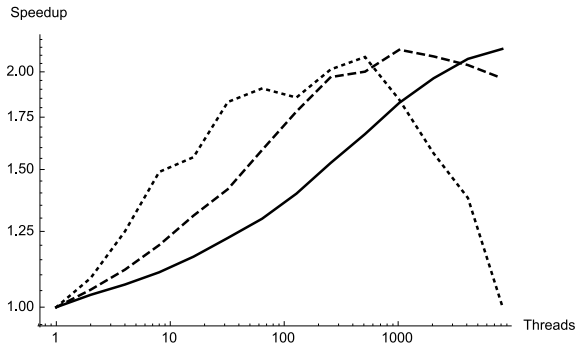

Fig. 10: Speedup in Haskell: 10K (dots); 100K (dashes); 1 million (solid)

Note that a modest amount of speedup in achieved, and at least for the large problem, this occurs for more than 1000 threads.

4 Comparisons

So far we have studied the speedups achieved as a function of the number of threads, with each of figures 1-10 having a different range on the vertical axis. Here we directly compare these speedups. Figure 11 plots them for the large problem instance of ZI Traders—a million buyers and a million sellers. (We have excluded Erlang, Python and Haskell because we do not have results on this problem instance for the first two, and since incommensurate hardware was used for the third).

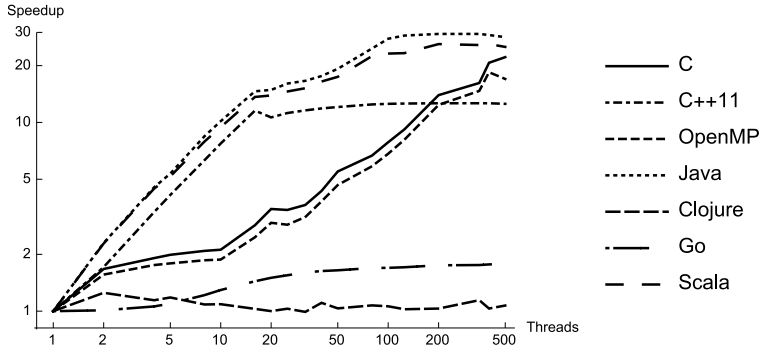

Fig. 11: Comparison of speedups achieved

Note the very similar speedups of C and OpenMP, on the one hand, and of Java and Scala on the other. Go and Clojure speed up the least.

Ultimately, perhaps the most important reason to parallelize an ABM is absolute execution speed. For each language and problem we determined the number of threads that yielded the least run time and have plotted these in figure 12. In the previous 11 figures *larger* values of the ordinate have indicated better performance, but

here *lower* values are better. Note that there is more than 2 orders of magnitude difference in performance between the different software systems tested. Note also that some five of them—C, OpenMP, C++, Go and Clojure—display approximately the same (exponential) increase in runtime as problem size increases (exponentially), while Java and Scala also increase but at lower exponential rate. Indeed, to solve a problem 100x larger, Java and Scala only require 5x more time. Python and Erlang are shown for only the first two problems; they are among the slowest.

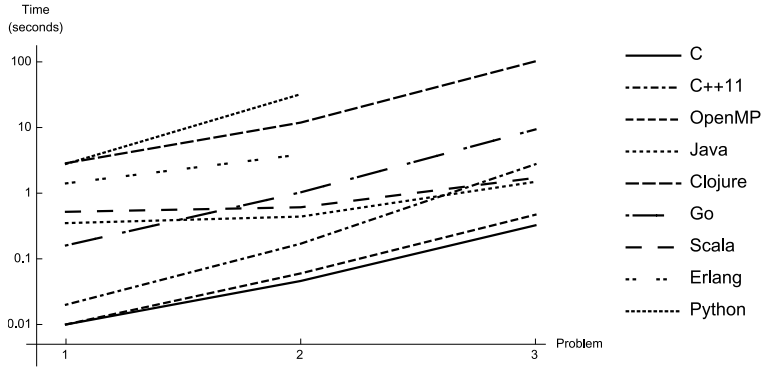

Fig. 12: Absolute performance comparison of execution time across problems and languages

It is also informative to look at the memory requirements of the various codes. This is done in figure 13 for the second problem (100K agents) where the vertical axis is in units of KB. Note the vast difference in memory requirements, from a few MB (C) to more than a GB (Clojure)! There is little growth in memory needs with the number of threads. Comparing figures 12 and 13, memory usage and execution time are directly related, with smaller memory footprints implying higher performance.

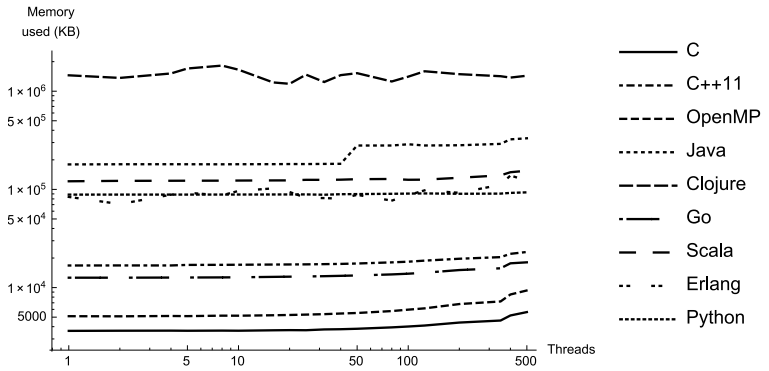

Figure 13: Memory requirements for the second problem (100K buyers and 100K sellers)

Overall, it is remarkable that nearly a 1000 fold difference in execution times and memory usage results from distinct software systems solving the same problem! Clearly, some of the compilers and runtimes are more optimized than others.

5 Summary and Conclusions

In this paper we have built nearly a dozen distinct implementations of the well-known zero-intelligence trader model, each in a language or software framework that in some way facilitates parallel execution of sub-populations of agents. Within these sub-populations of buyers and sellers the interactions are potentially quite dense, while between sub-populations they are minimal. The resulting codes show differential speed-up as a function of the number of threads. Typically, for each language/framework there is some optimum number of threads for execution of a model of a particular size. Sometimes this optimal level is clearly related to hardware considerations, when the best number of threads is approximately the number of cores present. Sometimes the optimal number is intimately connected to other aspects of the hardware, as when the creation of many very small sub-populations facilitates loading them entirely onto caches so that they can be executed with minimal swapping in and out of RAM. This is almost certainly what is going on here when the number of threads giving best performance is in the thousands or more. Finally, there are also situations where the optimal number of threads is very hard to understand from first principles, where the interaction of software and hardware is sufficiently complex that intuition for why the optimal level is what is observed is opaque.

A wide variety of absolute speedups have been observed, this variability no doubt due in part to relative inexperience programming some of the newer and less well understood languages. But the slow performance of some of the compiled codes is also likely due to their relative immaturity, the lack of optimized compliers, the fluid state of the newer languages, and so on. Many of these have room for significant improvement that the C and C++ languages almost certainly do not. As to which will have the greatest performance in the long run, it is very difficult to forecast. While parallel C/C++ will usually run fast, the challenge will be to create correct parallel codes in these languages. Other languages offer somewhat higher-level abstractions and may make writing parallel code easier but at a price of slower execution. In many cases this 'cost' may be acceptable, as a correct but slow code will eventually give the right answer whereas an incorrect but fast one never will.

In the end, the main value of the explorations described here may have less to do with the actual speedup numbers on display or even the codebases themselves, and more to do with the rather subjective judgments as to which languages are easier to use, which are more expressive, which are more stable, and which have room for performance improvement. For as of the time we are writing (approaching mid 2016), it is far from clear which software systems will provide the best tradeoffs for future efforts to parallelize agent models.

Acknowledgements

For constructive comments we are grateful to seminar participants in CSS 898 in Spring 2016 at George Mason University.

References

1. Ishida, T., L. Gasser, and H. Nakashima, eds. *Massively Multi-Agent Systems I*. Lecture Notes on Artificial Intelligence, J.G. Carbonell and J. Siekmann, eds., vol. 3446, Springer-Verlag: Berlin (2005).
2. Axtell, R.L., "120 Million Agents Self-Organize into 6 Million Firms: A Model of the U.S. Private Sector." In *Proceedings of the 15th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS2016)*, J. Thangarah, et al., eds. (2016).
3. Reinders, J., *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reily: Sebastopol, California (2007).
4. Kiran, M., et al., "FLAME: Simulating Large Populations of Agents on Parallel Hardware Architectures." In *Proceedings of the 9th International Conference on Autonomous Agent and Multiagent Systems (AAMAS 2010)*, W. van der Hoek, et al., eds. (2010).
5. Richmond, P., S. Coakley, and D.M. Romano, "A High Performance Agent Based Modelling Fremework on Graphics Card Hardware with CUDA." In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systes (AAMAS 2009)*, K.S. Decker, et al., eds. 1125-1126 (2009).
6. Gode, D.K. and S. Sunder, "What Makes Markets Allocationally Efficient?" *Quarterly Journal of Economics* **112** (2), 603-630 (1997).
7. Gode, D.K. and S. Sunder, "Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality." *Journal of Political Economy* **CI**, 119-137 (1993).
8. Cliff, D. and J. Bruten, "Minimal-Intelligence Agents for Bargaining Behaivors in Market-Based Environments." Hewlett-Packard Labs: Bristol, UK (1997).
9. Cliff, D. and J. Bruten, "Less Than Human: Simple Adaptive Trading Agents for CDA Markets." Hewlett-Packard Laboratories: Bristol, UK (1997).
10. Chapman, B., G. Jost, and R. van der Pas, *Parallel Programming in OpenMP*. MIT Press: Cambridge, Mass. (2007).
11. Quinn, M.J., *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill: New York, N.Y. (2003).
12. Oaks, S. and H. Wong, *Java Threads*. 3rd ed. O'Reilly Media: Sebastapol, Calif. (2004).
13. Emerick, C., B. Carper, and C. Grand, *Clojure Programming*. O'Reilly Media, Inc.: Sebastopol, California (2012).
14. Hoare, C.A.R., "Communicating sequential processes." *Communications of the ACM* **21** (8), 666-677 (1978).
15. Tang, P., "Multi-core parallel programming in Go." In *Proceedings of the First Int'l. Conference on Advanced Computing and Communications*, 64-69 (2010).
16. Armstrong, J., "History of Erlang." In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*, (2007).
17. Hudak, P., et al., "A history of Haskell: being lazy with class." In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*, 12-1 - 12-55 (2007).