Parallel Performance Comparison between Encore and OpenMP using Pedestrian Simulation

David Escher
Abstract

**Parallel Performance Comparison between Encore and OpenMP using Pedestrian Simulation**

David Escher

Parallel programming has become ubiquitous and more than a preference, it is a necessity for utilizing hardware resources (multiple cores) in order to speed-up computation. However, developing parallel programs is difficult and error-prone. In order to make development of parallel programs easier and less error-prone, new programming languages are being developed. This thesis compares Encore, a new programming language, against C++ with OpenMP for parallelizing program execution. The focus of this comparison is the respective run-times and parallel scalability, when running a pedestrian simulation program.
Contents

1 Introduction .................................................. 7

2 Background ................................................... 7
   2.1 Pedestrian simulation ........................................ 7
   2.2 OpenMP using C++ .......................................... 7
   2.3 Encore ...................................................... 7

3 Problem ....................................................... 8
   3.1 Problem specification ........................................ 9
   3.2 Reference implementation .................................... 9

4 Design ........................................................ 10
   4.1 Baseline .................................................... 10
   4.2 Optimizing the baseline ..................................... 11
   4.3 Shared Lock Matrix ......................................... 11
   4.4 Passive Agents and Active Regions ......................... 12
      4.4.1 Manager implementation ............................... 12
      4.4.2 Region implementation ............................... 13

5 Experiments and Results .................................... 15
   5.1 Parameter test ............................................. 15
   5.2 Core usage test ............................................ 17
   5.3 Performance analysis ...................................... 18
   5.4 Hardware utilization test ................................ 19

6 Related work .................................................. 20
   6.1 Survey of Active Objects ................................. 21
      6.1.1 Erlang and Akka ................................. 21
      6.1.2 Pony ........................................... 23

7 Design Limitations ........................................... 23
   7.1 Pedestrian simulation limitations ......................... 23
   7.2 Encore .................................................... 23
   7.3 Actor reference memory usage ............................. 24

8 Discussion .................................................... 24
   8.1 Object orientation and threads ............................ 24
   8.2 Lightweight threads ....................................... 25

9 Conclusions .................................................. 25

10 Future work ................................................ 26
11 Appendix

11.1 Encore Version .................................................. 29
11.2 XML instance data ............................................... 29
11.3 Benchmark Machine ............................................ 30
11.4 Tables ................................................................. 30
1 Introduction

In this thesis we will attempt to give some indication of whether or not Encore’s [4] parallel performance make it a viable alternative among other programming languages. We will measure parallel scalability and hardware utilization. As a reference for comparison, a existing C++ [24] implementation using OpenMP will be used. C++ is a language used in many high performance settings and OpenMP [10] is a technology developed to make it easier to parallelize code.

2 Background

The Encore language is currently under development, yet earlier tests on several benchmarks [26] show that Encore is on-par with Erlang and Scala in terms of speed, but have higher memory usage in some cases.

2.1 Pedestrian simulation

Pedestrian simulations are typically used for simulating people moving in spaces, and can be used for testing architectures of locations for cases like emergencies [5]. To compare Encore and C++ a rudimentary pedestrian simulation is used as a benchmark to compare different aspects of the languages performance, such as run-time, strong scalability, and hardware utilization. The requirements of this simulation are described in further detail in a later section of this paper (Section 3.2).

2.2 OpenMP using C++

OpenMP [10] is a framework for parallel programming, which mainly provides fork-and-join style parallel abstractions. In this thesis we are using the C++ [24] binding for the framework. C++ is a language that is widely used in the industry, especially in cases where performance is an important priority.

2.3 Encore

Encore is a programming language under development at Uppsala university. Its paradigm is called active-object, which is a combination of the actor model from concurrent programming and the object oriented programming paradigm. Encore has so called active-objects which is the main way to express concurrent computations in the language. It’s a type of object with its own single thread of control for manipulating its data.

The language is using lightweight [7] threads similar to those found in for example the Erlang run-time, and bind each of them to run the methods
of one object. One of the big advantages of this is that an active-objects data will only ever be handled by that thread, any other object which wants to read or write that data needs to send a message to the actor holding the object and await a reply in order to use the objects data. This in combination with the Kappa [6] type system, prevents low-level data races, and requires programmers to make explicit methods for sharing data.

As an abstraction for asynchronous method invocation, Encore uses futures, which are wrappers around possibly undelivered data. All active object methods, when called by another active-object does return a value of type future T, where T is the type of the value returned by the method. The main advantage of this is that it makes all method calls on active objects asynchronous by default, while synchronous behavior can be easily achieved.

As an example, a clock implemented as an active object in Encore might have a method called time() which returns the current time. However, if some other object calls clock ! time(), it’s translated to a remote method invocation, making the active-object clock run time() locally, and sending the result back to the local thread. This will result in the thread calling clock ! time() immediately getting a returned value of type future time, representing the fact that, at some later point, a value of type time will be delivered to them.

Encore supports some methods that operate on futures. The ones used in this work are get :: Future T → T which is a method blocking until the value T is delivered, and then returns T and await :: Future T → ∅ which is a method that suspends the current active-object method, allowing the object to run other methods. When the Future is fulfilled, the execution resumes from where await :: Future T → ∅ was called.

Encore also supports what it calls passive objects, which are very similar to objects [23] in many other object oriented languages, such as C++ and Java [2]. As Encore is in development, future versions of the language might provide additional useful constructs and might experience different behavior. However, those future versions are not considered much in this thesis. Mainly as the true implications of future language extensions are difficult to predict.

3 Problem

In order to compare Encore to C++, this thesis will use a reference implementation of pedestrian simulation implemented in C++ using OpenMP, and make implementations in Encore mimicking the reference as closely as possible. Then some comparison will be done between the implementations runtime, hardware utilization and scalability in terms of parallelism.
3.1 Problem specification

This specific pedestrian simulation have been used as an assignment program to parallelize in a course at Uppsala University, and the reference implementation is from that course. It’s an adaptation of Christian Gloor’s "pedsim" project\(^1\). From the course instructions: ”This project includes software from the PEDSIM3 simulator, a microscopic pedestrian crowd simulation system, that was originally adapted for the Low-Level Parallel Programming course 2015 at Uppsala University.”

This pedestrian simulations coordinate system is 2D dimensional and uses discrete coordinates, in a similar way to a chessboard. Agents each occupy one set of these coordinates, and only one of them can occupy a given set of coordinates at the same time, this also very similar to how chess pieces are placed on a board, there can only be one per square, and a piece is inside exactly one square. Agents can move 8 directionally, unit length, which is the same movement pattern as a king in chess, it can move only one square, but to any of the 8 squares surrounding it. Agents move in a time-sliced fashion, all agents move once (or attempts to move and decides not to), before all of them attempt to move again. They move towards instance data defined way-points and can be moved in an arbitrary order. The agents attempt to move towards their current way-point by checking first the most direct route, and then their two neighboring locations. As an example, if an agent’s way-point is directly to their right, they will first attempt to move to the right, then if that location is taken, move to either their bottom right or top right location if either of those are available. If none of these locations are available, they will not move. The case when a agent is discovering that they can not move to a location will be refereed to as a collision in this paper.

An agent is considered to have reached their way-point when they are within their radius, and will then start to move towards their next way-point. If their way-points run out, they will start over by moving towards their first way-point again.

3.2 Reference implementation

The reference implementation uses XML-files (Section 11.2) describing starting positions and way-points for agents as input data, and simulates agents for 10000 time steps, were one time step is each agent attempting to move one step.

The implementation uses "fork-and-join" [22] parallelism, moving all agents in a parallel code region and then joining before moving all agents again. A join is preformed between each time step to ensure that all movement of one time step is completed before the next time step starts. In

\(^{1}\)pedsim project website, http://pedsim.silmaril.org/ accessed 2017
the parallel code region, the simulation space is divided into regions that are moved in parallel by different threads. The regions split into smaller or merge into larger regions to try to balance out their size, this is done before each tick starts, in a serial code region. For agents moving between these region borders an atomics are used.

The main challenge of making an efficient implementation to solve this problem, is to handle agent collisions effectively. According to Orion Sky Lawlor [17], collision detection for this type of problem is possible in $O(N)$ time, even if the worst case can be $O(N^2)$, were $N$ is the number of agents per time step. This is an important fact to know when considering how to design the simulation, as any designs with a worse theoretical time complexity are very unlikely to have good performance. While this problem is different from the one in Orion Sky Lawlors paper, many of the same principles apply. One of the main differences is that Orion Sky Lawlor is trying to detect collisions, when we are trying to avoid them.

4 Design

Because of Encore’s design, the problem has to be re-modeled from the ”fork-and-join” reference implementation. This is mainly as Encore does not support operations like ”fork” and ”join” effectively, and although they can be emulated, they would most likely not be efficient. In the following sections, each presents a different way to design a pedestrian simulation in Encore.

When considering these designs, we focus on some key properties. One of these is a low number of messages sent per time step, as message send and receive are synchronization constructs, which should be avoided if possible. We also want a design that can do a lot of work in parallel, which means that we want a lot of actors with messages to process. There is an important distinction here, we want many actors to have messages to process, but we don’t want to send many messages, so messages that do more work will likely make the implementation more efficient. Lastly, we want a design with low time complexity, although that can be difficult to discern the time complexity of some of these algorithms in a meaningful way.

4.1 Baseline

As a baseline implementation in Encore, for comparison to other Encore implementations, all agents are modeled as active objects, with no shared data-structure for detecting collisions. The agents are required to send messages to all other agents, checking if the positions they want to enter are clear.

Considering the baseline implementation requires all agents to broadcast to all other agents on each step to detect collisions, the amount of messages
sent and time complexity of the implementation is $O(TN^2)$ were $N$ is the number of agents and $T$ is the number of time steps. This is because an agent before it moves needs to send one message to each of the other agent, learning about it’s position. The fact that the time complexity of each move operation is therefore $O(N)$ makes the time complexity of all move operations in a time step $O(N^2)$.

4.2 Optimizing the baseline

In order to reduce the amount of message passing required agents can keep a local list of the minimum distance between themselves and other agents, and only send messages to update that list and detect collisions when the minimum distance is small. This list contains the Manhattan distance between the agent and another agent, and is reduced by two in each time step, as it’s assuming the agents are moving towards each-other. When the distance reaches 1, a message is sent to the other agent to find it’s actual position again. As we are assuming the agents always move towards each-other, we have a guarantee that the agent pair can not collide while their Manhattan distance is greater than 1. When the distance is equal to or smaller than 1, the agent sends a message to recalculate the distance and be able to avoid potential collisions.

This increases the memory complexity of the agents (as they all need a local distance list) while it can decrease the required amount of message passing significantly. Depending on the instance data, this optimization can cause very few messages to be sent but in worst case it’s still $O(TN^2)$.

The problem with this design is that it’s memory complexity is $O(N^2)$, and that data contains a lot of active object references. This have a special interaction with the Encore run-time, as discussed later in this paper (Section 7.3).

4.3 Shared Lock Matrix

Instead of agents sending messages directly to other agents, a shared data-structure maintaining information about the agents positions can be used. In this case, a matrix of ”positions”, each representing one particular position. This implementation does potentially have very high memory complexity, if the space agents can occupy is big. The main advantage of this implementation is that each agent is guaranteed to be able to make it’s move using a constant number of messages, which gives a $O(N)$ worst case time and message complexity. Its performance is also dependent on how effectively this large matrix can be shared between agents.

This is implemented in Encore as a Matrix of active objects holding a boolean representing whether or not a position is taken. All agents, that are still active objects send messages to the ”position” agents, requesting
to move into their positions. This guarantees that no two agents are in the same position, as the "position" active objects only allows agents to move to their position if it’s not taken by another agent. It also guarantees that an agents move can be done in constant time, by sending messages to all their desired positions, giving us the $O(N)$ worst case time and message passing complexity.

This experiences the same kind of memory usage behavior as the previous implementation. Because of this, any implementation similar to this one were also considered unfeasible to implement effectively. See section 7.3 for more details.

4.4 Passive Agents and Active Regions

A different approach to this problem is to divide the space the agents traverse into regions. Each region is an active object that knows about it’s neighbors and responsible for moving agents inside it. When modeling the problem this way messages are only needed for synchronizing the regions between each move and for handling agents crossing the border between regions. It is also what is used by Orion Sky Lawlor [17] as an efficient parallel collision detection algorithm. This is the only design that have been fully implemented and used in the benchmarks, due to it being significantly faster than all other designed prototypes.

4.4.1 Manager implementation

As the regions are only allowed to move all the agents once, before moving them again, some entity is required to synchronize the regions. We call this a manager and it’s also responsible for creating all the regions, assigning them parts of the coordinate space and introducing them to their neighbors.

In order to figure out how big the 2D space is, the manager finds the extreme values of the coordinates of all agents starting positions and way-points.

When subdividing the space, it’s important to remember what run-time we are working with. As Encore is operating using light-weight threads, our main concern is to create enough regions. According to Orion Sky Lawlor [17], the best results for similar types of run-times using similar abstractions are typically somewhere between 10 and 10000 regions per core of the target machine. This is of course highly dependent on the specific run-time and problem but in general the number of active objects should be much higher than the number of cores used on the target system.

It’s also worth noting that the cost of having uneven workload between the regions is quite cheap. This is because the run-time will try to find a even load balancing between the cores using active objects as work units so having one region take more time than other is only a problem if the time
Figure 1: An example subdivision with parameter value 4, the 4 by 4 grid of rectangles all have the same size (red), except for the topmost (yellow) and rightmost row (blue) that have differing heights or widths. Only one rectangle (green) can have both a differing height and width.

Therefore, we split the 2D space into many uniform rectangles, organized in a grid pattern. In order to make the code adaptable to different architectures, it takes a parameter which is proportional to how many regions are created. In this case $N$, were $N^2$ rectangles are created. The regions width is the total width of the simulation divided with $N$ and their height the total height divided by $N$. This division is not always perfect as we are working with discrete coordinates, so in order to handle uneven subdivisions the upper and rightmost row of regions will be slightly larger or smaller than the rest, as illustrated in Figure1.

Each of these rectangles are assigned a region, and filled with the agents residing within it’s coordinates. When the simulation starts, the manager sends a message to all regions telling them to move all their agents one step and then awaits all of them doing that, at which point it send another set of messages. It’s also responsible for visualization and timing if that is used.

4.4.2 Region implementation

When considering each region, as movement is eight directional, each region needs to know about up to eight adjacent regions to handle agents, less if they are on the edge of the simulation. In addition, it needs some way of efficiently handling local agent movement.

For internal movement, each region needs some data structure to store
all the agents current positions. This data structure needs to have quick look-up and update times. The chosen data structure is a quadtree. The main advantage of using a quad-tree is that it’s very effective for storing sparse and empty regions [21].

The Regions also need a second data structure for holding the agents. As agents during any time-step might move from one region to another, quick remove and insert times are needed for this data structure.

Because agents can be moved in an arbitrary order, but only once in each time step, a quick way to iterate this data structure is also required. The choice made for this structure is a linked list.

Each region have four linked list pointers, New pointing to the first newly arrived agent, Last pointing to the lastly arrived agent this time step, Old pointing to the first of the current agents in the region and Current pointing to the first agent left to move in a region.

When one of the regions neighbors sends a request to move an agent into this region, collision detection is done by the receiving region looking up the desired position of that agent in the quad tree. If that spot in the quad tree is empty, it’s filled and the agent is added to the New linked list. This means that the agent will not be moved by this region (as it already have been moved from the neighbor), but it’s collisions will still be detected (as it’s inside the quadtree used for this).

The region will move its agents one at the time, using Current as an iterator. In case one of the agents desires to move outside of the current region, this region will figure out which of the neighbors contains the desired position of that region, and send a request for that agent to move there. The region will then await the reply, which is a language construct meaning that it will only handle incoming requests from other regions and not attempt to make any local progress until it’s request is fulfilled. Once it’s time to move another step, Old is added to the end of Last and New becomes the new Old pointer.

This design is made in part to keep the complexity of the regions low, as extra overhead from having multiple outlying requests at once will require more logic and overhead in the regions. It also guarantees that the maximum amount of request to be fulfilled in the system is lower than the number of regions, which in turn eliminates the possibility of large number of messages under some circumstances consuming large quantities of memory. One of the main concerns if regions could have more than one unfulfilled request is that it might be difficult to find an upper bound for how many unfulfilled requests can exist in the system. This in turn can lead to unpredictable memory usage, if for example some unlikely scheduling causes a lot of requests to be created but not fulfilled.

Most agents will not move over the edge of any region, and therefore their movement can be done completely locally, by simply updating the local quad tree.
5 Experiments and Results

Experiments were conducted on a 64 core "bulldozer" machine. More details about the machine can be found in Appendix (Section 11.3). In the tests, \( \text{min} \) is the smallest time measured, \( \text{max} \) is the greatest, and \( \text{geomean} \) is the geometric mean, which is the \( N \)th rot of the product of \( N \) samples, so the geomean of \( N \) samples, \( x_1, x_2 \ldots x_n \) is \( \sqrt[N]{x_1x_2\ldots x_{n-1}x_n} \). All experiments are conducted using the Passive Agents and Active Regions implementation, as earlier testing showed that it was the only viable design in terms of both speed and memory usage. The data used for creating the graphs in this section can be found in the Appendix (Section 11.4).

The two scenarios tested are using 4000 and 16000 agents with densities of 0.12 agents per area unit and 0.44 agents per area unit respectively.

5.1 Parameter test

In this experiment we vary the number of regions in order to find the optimal number of regions for this architecture. All 64 cores were used, and the results are displayed in Figure3 and Figure4.

This test quite clearly show that the optimal configuration for the Encore implementation is using only one parallel region, which is roughly equivalent to a serial implementation (alto it’s not nearly as efficient due to still having overheads from the manager). This is unexpected as the implementation is designed to be parallel and should be able to use all 64 cores with a different parameter value. This suggests that the parallelization overhead is greater than the speedup achieved by parallelization for the Encore implementation.
Figure 3: Run-times for Encore varying the number of regions, reporting the lowest run time out of 10 independent runs.

Figure 4: Run-times for Encore varying the number of regions, reporting the Geomean, minimum and maximum run time of 10 independent runs.
5.2 Core usage test

In this test, the number of cores used were changed in order to investigate what is the optimal number to use for both the reference and Encore implementation. The number of regions used by the Encore implementation is what was found to be optimal in previous tests other than parameter value one, as with only one parallel region, there is no possibility of performance increase with an increased number of cores. The parameter values are therefore 16, which translates to 16\(^2\) or 256 regions for 4000 agents, and 9\(^2\) or 81 regions for 16000 agents.

The results are displayed in Figure 6 and Figure 5. The best time achieved for 4000 agents are 10 seconds for OpenMP using one core, and 110 for Encore using 25 cores. The best time achieved for 16000 agents are 48 seconds for OpenMP using one core, and 365 for Encore using 32 cores. This gives us some more important pieces of information. First of all, it shows that the Encore implementation does benefit from an increased number of cores, at least for the tested parameter values. It also shows that the reference implementation is not benefiting from using many cores, it’s actually slows down quite significantly with an increased number of cores. This seems to indicate that the parallelization overhead outweighs the speedup from using many cores for the reference implementation.
5.3 Performance analysis

In order to investigate what were the dominant factors in the Encore implementations run-time, performance analysis was done using a tool called perf [25]. This produced a breakdown in percent of runtime used per method and function in the implementation (Figure8). This breakdown is difficult to understand without the context of seeing the code and knowing the language. Therefore Figure9 displays what the perf results really mean. The most interesting about this is the bottom entry included in the breakdown. As a Quadtree lookup is done every single time an agent attempts to move, it’s expected to be a significant portion of the total region local computations. This is also what we see in the breakdown, it’s the most time consuming Encore method (method/function written by the programmer and not the language designer). However, it’s a total of 1.7% or runtime, less than two different Garbage collection methods, and one message passing method, that all consume one order of magnitude more runtime than it does.

The amount of messages passed by the system is non-deterministic, as the simulation is non-deterministic and messages are only sent when agents are crossing the border between regions, and once per time step between the manager and the regions. This gives us a trivial upper bound for the number of messages sent to $O((N + R) * T)$ were $N$ is the number of agents, $T$ is the number of time steps and $R$ is the number of regions. However, in practice
the number of messages sent is much lower, as most agents will not move
between regions and can therefore be moved without any message passing.
This means that we can assume that quad tree lookups are done more often
than message sends. If we use this estimate for the scenario profiled in perf,
this gives us \((16000 + 81) \times 10000 = 16081 \times 10^4\), as an upper bound for how
many messages are passed during the run. We can then use the geometric runtime for this configuration measured in the core test, 402.7 seconds to
get the number of messages sent per second, 399329.52. We then divide by
the number of cores used (64) to get an upper bound for the number of
messages sent per second per core, which is 6239.5 messages per core and
second, at most. As we know from perf that message passing takes at least
20% of the total runtime, this is interesting. However, as there is nothing
to compare it to, it is difficult to say if that its good or bad.

The perf method breakdown does support the idea that the parallelization
overhead is dominating the runtime. The exact values obtained from
perf does vary depending on the scenario and parameter value, but the order
of the methods seem fairly stable between configurations, and we have yet
to find one were quadtree lockup consumes more than 5% of the runtime.

5.4 Hardware utilization test

In this test, both implementations were run and performance data was gath-
ered from a number of hardware counters using perf.
This test shows that Encore have more branch misses, and cache load misses than the reference implementation. This is expected as the Encore implementation is using linked lists and quadtrees, which tend to be less cache friendly than the arrays mostly used in the reference implementation.

As the Encore and Openmp implementations are so different, and the differences while significant are expected given the differences between the implementation. We cannot draw any conclusions about the relative hardware utilization between the languages.

6 Related work

This work is not the first to attempt to benchmark Encores performance compared to other languages, and is designed to complement the earlier work of Mikael Östlund [26]. In their paper [26], Encore is compared to Scala and Erlang over three different benchmarks.

In the Big benchmark, agents are divided into large groups and sends a lot of messages between random actors in the group. In that benchmark Encore, while faster than the other languages, does not achieve speedup for all configurations, with the 3 core configuration resulting in a slow down compared to the one core configuration. Memory tests of the benchmark show that Encore is consuming about one order of magnitude more memory.
than Scala and Erlang for all problem instances. The second benchmark Chameneos, is designed to measure fairness for shared resources, Encore has no memory issues, and performs about the same as the other two languages in terms of run time and scalability. For the fairness measurement, it seems like Encore is better than the other two languages. The last benchmark is a parallel version of the Fannkuch benchmark, designed to test parallel performance. In this test, Scala is faster than the other two, but they are all very close in terms of speedup.

The pedestrian simulation used in this thesis is derived from PEDSIM, a pedestrian simulation package designed by Christian Gloor, which was used as an assignment project in the course Low-Level Parallel Programming during the spring of 2016. Christian Gloor have published a several papers [14] [13] about pedestrian simulation. These papers largely focus on techniques for realistically simulating pedestrians, and different models that are realistic enough to be useful while performing good enough to be used.

However, these models include much more complex agent behavior, such a social forces and acceleration, which makes their techniques difficult to apply directly to the pedestrian simulation used in this thesis. Also, while parallel computation is mentioned in Christian Gloors papers, there is no data documenting its benefits and the quote ”As of now, the update is sequential; for our problems, no large differences to parallel update were observed” (page 5) [13] seems to indicate that benefits of parallelizing pedestrian simulation is not necessarily beneficial even for other pedestrian simulators.

6.1 Survey of Active Objects

Encore is one of the few active object programming languages currently in existence. In this section we will explore some of the similarities and differences between Encore and the existing techniques for parallel programming Erlang and Akka, as well as Pony, the language associated with the runtime Encore is based on.

6.1.1 Erlang and Akka

Erlang [1] is a programming language originally developed for the telecoms industry [1]. Erlang is a functional programming language [3] and supports lightweight threads [7]. Erlang threads communicate using asynchronous messages, and have an explicit receive message construct.

A common way to structure Erlang code is to have the lightweight threads acting as servers, holding some data and waiting for messages, replying to requests. This is very similar to how active objects operate. However because Erlang supports explicit messages and lightweight threads, this is by no means the only way to structure code.

Akka[15] is a framework which have a lot of overlapping design goals
with Erlang, as it’s also focusing on building fault tolerant programs using actors. Akka is a framework for the JVM (the Java Virtual Machine) it can be used in combination with either Java [2] or Scala [20], which both compile to the JVM. Depending on which language used, the code can be either object oriented, functional or both.

In order to prevent data races in both Erlang and Akka, only immutable data should be shared, however in Akka there is no way for to enforce this currently[19]. Erlang and Akka messages are simply data, delivered asynchronously to some actor.

Akka and Erlang both use an explicit receive construct for reading messages from their mailbox. One important difference is that Akka’s receive message construct is an object method, which implies that there is only one per actor. This in different from Erlang’s receive which is a flow control construct, similar to a match statement but where the data to match is from the mailbox. This is important as Erlang threads can encounter multiple receive clauses, matching different message types, were Akka actors can only have one. Akka receive patterns are also required to be exhaustive [18][15], meaning that an Akka actor needs to be able to handle any type of message sent to them. Erlang receives have no such requirement, and can therefore have messages that never gets processed. Akka builds fault tolerant systems typically by designing a hierarchy of actors, which in Erlang is called the supervisor pattern.

One important difference between Akka actors and most other Java objects is that they are not automatically garbage collected, but have to manually be destroyed when they have outlived their purpose [18][15]. This is similar to Erlang, as Erlang actors are just lightweight threads, that terminate when they are out of code to run. Therefore Erlang have no concept of actors being garbage collected, as they are not data structures. This is also different from Encore, were active objects are garbage collected when no longer needed. One effect of this is that Encore actor references are guaranteed to reference a live actor, while Erlang and Scala actor references are not.

Both Java and Scala do have type systems, and we will not go into detail of either of them in this thesis. Erlang however, does not have a type system, alto there are tools such as Dialyzer [16] which can detect the same type of problems a type system can.

Because both Java and Scala support objects, Akka actors are also Java objects, meaning that they have state and methods, however there are some big differences between Akka’s actor model with objects and the Encores active-object model.

One of the main differences is that Encore messages can not easily be handled in any order other then first in first served, as Encore messages are generated by calling methods asynchronously and therefore there is no message receive construct for specifying such things. Encore messages also
have compile-time defined types, as an active object can only ever be sent messages matching the parameters of its methods. However, Erlang and Akka actors can be sent arbitrary messages (Any object, only immutable ones by convention and any term receptively).

6.1.2 Pony

Pony is a language from which Encore have used some components. Unlike Akka and Erlang, Pony is still a language under development. It promises to be type, memory, exception safe as well as data-race and deadlock free. The pony language does seem to have a quite solid theoretical ground. However, as it is still in development it has some problems to quote their website "If your project is going to succeed or fail based on the size of community around the tools you are using, Pony is not a good choice for you. While it’s possible to write stable, high-performance applications using Pony, you will have to do a decent amount of work". Just like Encore, Pony supports embedding C libraries which does make it easier to use but it is not close to the same amount of available tools and libraries existing for the JVM or Erlang.

The main difference in focus for Pony compared to Erlang and Akka is that Pony wants to catch all errors at compile time, while the Erlang/Akka approach is to fail-fast and then recover from failure quickly and easily. This is what makes Pony a new bold idea, and only time will tell if catching all problems at compile time is a possible and useful approach to program development.

7 Design Limitations

7.1 Pedestrian simulation limitations

The pedestrian simulation specification used for this performance benchmark does not seem have any potential parallel performance improvement, at least not for any of the tested configurations. This means that the design is far from the fastest program to solve this particular problem in Encore, as a simple serial implementation, avoiding the overhead of using message passing is faster, and using less memory as it does not create any additional actors.

7.2 Encore

As Encore is in constant development, the results of this experiments might change with future updates to the language. There are also several other abstractions for parallel computation that are in development and might provide improvements to the language performance. Some of these are "hot
objects”, which are a type of multi-threaded active object which may be used to implement a parallel quadtree (or similar structure), making implementations similar to the shared lock matrix implementation viable. Another useful language extension are parallel combinators [12], which offer a different way to describe parallel computations.

Later versions of Encore will also likely feature better optimizations, and features like a standard library offering highly optimized data structures. All these are expected to yield Encore implementations faster with respect to other languages.

7.3 Actor reference memory usage

In current Encore, indirect references to active objects have a size, in the sense that, if you have for example an array of N active objects, each reference to that array increases memory usage by the program by an amount proportional to N. An indirect reference is in this case defined as, if an active object A can send a message to another active object B, A have an indirect reference to B.

This is one of the reasons the shared lock matrix and other baseline optimization’s are unfeasible to get working at any decent efficiency. The reason for this is a side effect of how the garbage collection system operates, and might be changed in future versions of Encore. This behavior have been reported to the language developers\(^2\).

8 Discussion

8.1 Object orientation and threads

One of the main differences between using OpenMP and Encore is that in OpenMP thread synchronization constructs are built in and thread communication is not available directly to the programmer. This means that the Encore programmer has to set up a sound message passing system between their active objects. This has the advantage of being very flexible. However, this also means that Encore programmers need to be able to build sound message passing systems, which can be difficult. For example, designing a message passing system where forward progress is guaranteed, and an upper bound for memory usage can be calculated might seem trivial but are really not, and without such guarantees I would not call a program a sound design.

One of the main problems encountered during the development of this benchmark, is the risk that some message of an unexpected type arrives. In particular, the region implementations worker threads have 2 different states, when it’s working on moving agents, and when it’s waiting on the next ”move” message. While in the state of moving agents, it need to be able

\(^2\)https://github.com/parapluu/encore/issues/646
to send and receive messages from other workers, as that is how transition between regions are done.

However, when debugging, it’s difficult to know if those are the only messages actors receive. If for example, one region would receive a second move message while working on the first move message, many invariants inside the region code would be violated and the program would likely crash.

The standard way to deal with this kind of issue is to not use the await construct unless any message can be handled correctly, but that is sometimes not an option, and can in many cases determent performance by not allowing some other messages to be handled. In this particular case for regions to handle cross border movement requests while waiting for their cross border movement request to be processed is required to keep the the system from potentially deadlocking by having two regions send the other a message, and not processing any messages until they got a reply.

8.2 Lightweight threads

The active objects representation in the run-time of Encore is as lightweight threads. This abstraction is really powerful, as programmers now don’t have to worry about making too many threads that often. This is one of the abstraction that make Erlang [1] the tool of choose for some applications, and now there is another language that works natively with it. One of the main reasons Erlang is using lightweight threads is that it aids in designing fault tolerant applications [1].

I think this abstraction will see a lot of use in the future, as the number of cores in CPU’s grow. Having the lightweight threads tied to objects is a new idea, but it’s quite commonly how Erlang code is structured even if it have other options.

9 Conclusions

Can Encore get comparative performance to existing technologies? Given our results there is no way to tell. As both implementations aims towards utilizing parallelism to improve performance, and neither achieves that goal. The run time values are uninteresting, as the run time is dominated by parallelization overhead. Encore is slower, but that is likely due to it having a higher parallelization overhead, which is expected as the Encore run time is less mature than OpenMP.

Some conclusion can be drawn from the tests. It seems like Encore has the potential to achieve parallel speedup, as the Encore implementations runtime lower with an increased number of cores if the simulation parameters are tweaked to allow for parallel computations. The parameter tweaking results seem to support the theory of many small work unit being optimal
for Encore, as the lowest run time (other than with 1 region) are with 81 and 256 regions respectively, which is higher than the number of cores.

The run-time test does show that Encore can utilize a wide range of cores with the same implementation. While the optimal number of cores used for the 4000 agent scenario turns out to be 25, and performance deteriorates after that point, it does not do so by much. This deterioration is likely caused by over eager work stealing, which is an indication that there is not enough work for all the cores. In the 16000 agent run-time plot we see less of this deterioration, indicating that all core can divide the work quite effectively.

When analyzing the parameter test, it’s clear that the parameter value is important for performance, however there is a fairly long range of values that have quite similar performance (13-16). That is interesting considering the differences in number of regions created by those same parameter values, as the number of regions is the parameter value squared. In the range 13-16 there is only a 1% time difference, while the difference in number of regions is 87, $13^2 = 169$, $16^2 = 256$, about 51%. This seems to indicate that Encore can use a wide range in terms of number of actors, and still be effective.

10 Future work

In order to get more insight into how well Encore compares to other languages, we would propose to make a test similar to this one were achieving parallel performance improvement is possible for the reference implementation. Further more detailed investigations of the performance of the garbage collector, and other language features, might also give some more insight into which parts of the Encore language are most efficient, and which have high overheads.

References


11 Appendix

11.1 Encore Version

Tests were conducted using encore-development, commit id:

698b8485cc69b2072a54f507dc6d5935aaaf36c1

Which is available at https://github.com/parapluu/encore.

11.2 XML instance data

The instance data provided as input to the reference implementation is formatted as can be seen below.

```xml
<welcome>
  <!-- waypoints - define before the agents! -->
  <waypoint id="w1" x="0" y="60" r="10" />
  <waypoint id="w2" x="160" y="60" r="10" />
  <agent xs="0" ys="60" n="1000" dx="40" dy="60">
    <addwaypoint id="w2" />
    <addwaypoint id="w1" />
  </agent>
  <agent xs="160" ys="60" n="1000" dx="50" dy="30">
    <addwaypoint id="w1" />
    <addwaypoint id="w2" />
  </agent>
</welcome>
```

A way-point is defined using its X and Y coordinates, and its radius R. An agent is considered to have reached a way-point when its within its radius.

The element "agent" refers to a group of agents. It’s attributes are, xs and ys defining the center of a rectangle were the agents are to be created. The attributes dx and dy are the distance from the center to the edges of
rectangle in either direction. The number \( n \) is the number of agents to create in that region.

As the instance data does not define where the agents are created, and what happens if the number of agents is greater than the number of spaces available in the defined rectangle, the call was made to standardize this. If the number of agents is greater than the number of available spaces, the program should not start a normal simulation, but print some error message. The agents are created starting with the lowest allowed X and Y coordinates, increasing X until the greatest allowed X coordinate and then incrementing Y.

11.3 Benchmark Machine

Benchmark Server "Bulldozer"
OS Linux 4.9.0-2-amd64 x86_64
Processor AMD Opteron(TM) Processor 6276
   L1 Cache 16 kB
   L2 Cache 2 MB
   L3 Cache 6 MB

11.4 Tables

This section contains the data described in Section5.

<table>
<thead>
<tr>
<th>Cores</th>
<th>16000 agents, varying number of cores, 10 independent runs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenMP</td>
</tr>
<tr>
<td></td>
<td>Max</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>140</td>
</tr>
<tr>
<td>3</td>
<td>155</td>
</tr>
<tr>
<td>4</td>
<td>196</td>
</tr>
<tr>
<td>5</td>
<td>216</td>
</tr>
<tr>
<td>6</td>
<td>210</td>
</tr>
<tr>
<td>7</td>
<td>208</td>
</tr>
<tr>
<td>8</td>
<td>207</td>
</tr>
<tr>
<td>9</td>
<td>207</td>
</tr>
<tr>
<td>10</td>
<td>209</td>
</tr>
<tr>
<td>11</td>
<td>208</td>
</tr>
<tr>
<td>12</td>
<td>209</td>
</tr>
<tr>
<td>13</td>
<td>209</td>
</tr>
<tr>
<td>14</td>
<td>211</td>
</tr>
<tr>
<td>15</td>
<td>207</td>
</tr>
<tr>
<td>16</td>
<td>209</td>
</tr>
<tr>
<td>17</td>
<td>212</td>
</tr>
</tbody>
</table>
Table 1: Data for the 16000 agent scenario, parameter value 9, translating to 81 regions. Results of 10 independent runs. All times in seconds.

<table>
<thead>
<tr>
<th>Cores</th>
<th>OpenMp (Min)</th>
<th>Encore (Min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>232</td>
<td>224.4268511196</td>
</tr>
<tr>
<td>60</td>
<td>227</td>
<td>222.3384622747</td>
</tr>
<tr>
<td>61</td>
<td>227</td>
<td>222.515329611</td>
</tr>
<tr>
<td>62</td>
<td>227</td>
<td>223.9683984253</td>
</tr>
<tr>
<td>63</td>
<td>230</td>
<td>225.7804436124</td>
</tr>
<tr>
<td>64</td>
<td>234</td>
<td>226.1346077066</td>
</tr>
</tbody>
</table>

4000 agents

<table>
<thead>
<tr>
<th>Cores</th>
<th>OpenMp (Min)</th>
<th>Encore (Min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>772</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>472</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>350</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>277</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
<td>241</td>
</tr>
<tr>
<td>6</td>
<td>49</td>
<td>221</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>197</td>
</tr>
<tr>
<td>8</td>
<td>49</td>
<td>181</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>175</td>
</tr>
<tr>
<td>10</td>
<td>49</td>
<td>167</td>
</tr>
<tr>
<td>11</td>
<td>50</td>
<td>161</td>
</tr>
<tr>
<td>12</td>
<td>51</td>
<td>152</td>
</tr>
<tr>
<td>13</td>
<td>50</td>
<td>148</td>
</tr>
<tr>
<td>14</td>
<td>51</td>
<td>146</td>
</tr>
<tr>
<td>15</td>
<td>50</td>
<td>138</td>
</tr>
<tr>
<td>16</td>
<td>50</td>
<td>136</td>
</tr>
<tr>
<td>17</td>
<td>51</td>
<td>131</td>
</tr>
<tr>
<td>18</td>
<td>51</td>
<td>126</td>
</tr>
<tr>
<td>19</td>
<td>52</td>
<td>122</td>
</tr>
<tr>
<td>20</td>
<td>51</td>
<td>119</td>
</tr>
<tr>
<td>21</td>
<td>51</td>
<td>117</td>
</tr>
<tr>
<td>22</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>23</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>24</td>
<td>52</td>
<td>111</td>
</tr>
<tr>
<td>25</td>
<td>53</td>
<td>110</td>
</tr>
<tr>
<td>26</td>
<td>51</td>
<td>110</td>
</tr>
<tr>
<td>27</td>
<td>52</td>
<td>111</td>
</tr>
<tr>
<td>28</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>29</td>
<td>51</td>
<td>111</td>
</tr>
</tbody>
</table>
Table 2: Data for the 4000 agent scenario, parameter value 16, translating to 256 regions. Results of 10 independent runs. Lowest time of 10 independent runs. All times in seconds.

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>111</td>
</tr>
<tr>
<td>31</td>
<td>111</td>
</tr>
<tr>
<td>32</td>
<td>111</td>
</tr>
<tr>
<td>33</td>
<td>112</td>
</tr>
<tr>
<td>34</td>
<td>113</td>
</tr>
<tr>
<td>35</td>
<td>114</td>
</tr>
<tr>
<td>36</td>
<td>114</td>
</tr>
<tr>
<td>37</td>
<td>116</td>
</tr>
<tr>
<td>38</td>
<td>114</td>
</tr>
<tr>
<td>39</td>
<td>116</td>
</tr>
<tr>
<td>40</td>
<td>117</td>
</tr>
<tr>
<td>41</td>
<td>118</td>
</tr>
<tr>
<td>42</td>
<td>120</td>
</tr>
<tr>
<td>43</td>
<td>120</td>
</tr>
<tr>
<td>44</td>
<td>120</td>
</tr>
<tr>
<td>45</td>
<td>121</td>
</tr>
<tr>
<td>46</td>
<td>121</td>
</tr>
<tr>
<td>47</td>
<td>122</td>
</tr>
<tr>
<td>48</td>
<td>125</td>
</tr>
<tr>
<td>49</td>
<td>124</td>
</tr>
<tr>
<td>50</td>
<td>124</td>
</tr>
<tr>
<td>51</td>
<td>125</td>
</tr>
<tr>
<td>52</td>
<td>125</td>
</tr>
<tr>
<td>53</td>
<td>126</td>
</tr>
<tr>
<td>54</td>
<td>127</td>
</tr>
<tr>
<td>55</td>
<td>126</td>
</tr>
<tr>
<td>56</td>
<td>129</td>
</tr>
<tr>
<td>57</td>
<td>128</td>
</tr>
<tr>
<td>58</td>
<td>128</td>
</tr>
<tr>
<td>59</td>
<td>130</td>
</tr>
<tr>
<td>60</td>
<td>131</td>
</tr>
<tr>
<td>61</td>
<td>131</td>
</tr>
<tr>
<td>62</td>
<td>131</td>
</tr>
<tr>
<td>63</td>
<td>133</td>
</tr>
<tr>
<td>64</td>
<td>135</td>
</tr>
</tbody>
</table>
Table 3: Data for the 4000 agent scenario. Results of 10 independent runs. Lowest time of 10 independent runs. All times in seconds.
Table 4: Data for the 16000 agent scenario, using 64 cores. Results of 10 independent runs. Sample size of 10 independent runs

<table>
<thead>
<tr>
<th>Instance</th>
<th>4000 agents, 1 core</th>
<th>16000 agents 64 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encore-Runtime</td>
<td>725</td>
<td>411</td>
</tr>
<tr>
<td>OpenMP-Runtime</td>
<td>11</td>
<td>220</td>
</tr>
<tr>
<td>encore-task-clock</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>openmp-task-clock</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>encore-branch-miss-%</td>
<td>2.45</td>
<td>1.4</td>
</tr>
<tr>
<td>openmp-branch-miss-%</td>
<td>1.45</td>
<td>0.13</td>
</tr>
<tr>
<td>Encore-LLC-load-misses</td>
<td>0.41</td>
<td>0.96</td>
</tr>
<tr>
<td>openmp-LLC-load-misses</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>encore-L1-dcache-load-misses</td>
<td>13.15</td>
<td>6.66</td>
</tr>
<tr>
<td>openmp-L1-dcache-load-misses</td>
<td>2.11</td>
<td>0.17</td>
</tr>
<tr>
<td>encore-L1-icache-load-misses</td>
<td>12.5</td>
<td>12.6</td>
</tr>
<tr>
<td>openmp-L1-icache-load-misses</td>
<td>12.52</td>
<td>12.49</td>
</tr>
</tbody>
</table>

Table 5: Data from the perf performance counters.