Interaction Automata and the ia2d Interpreter

Stéphane Gimenez and David Obwaller

Institute of Computer Science
University of Innsbruck, Austria
`stephane.gimenez@uibk.ac.at`, `david.obwaller@student.uibk.ac.at`

Abstract

We introduce interaction automata as a topological model of computation and present the conceptual plane interpreter `ia2d`. Interaction automata form a refinement of both interaction nets and cellular automata models that combine data deployment, memory management and structured computation mechanisms. Their local structure is inspired from pointer machines and allows an asynchronous spatial distribution of the computation. Our tool can be considered as a proof-of-concept piece of abstract hardware on which functional programs can be run in parallel.

1998 ACM Subject Classification F.1.2 Modes of computation

Keywords and phrases Interaction nets, computation models, parallel computation, functional programming

1 Introduction

We present a candidate computation model with a quite natural, although non-standard memory model reminiscent of cellular automata topologies which takes spatial distances and related data migration costs into consideration. Our associated tool interpreter `ia2d` is written in Haskell and is available online[1]. This interpreter takes as input a program written as an *interaction-net system*, a generic, topology-independent language. This program is then executed in parallel on a planar instance of the presented computation model, and the result of the computation is returned in the same interaction-net language, along with some detailed resource usage statistics. Operations on binary-encoded natural numbers, implementations of *merge sort* and of the *bitonic sorter*, which has a theoretical parallel time complexity of $O(\log^2(n))$, are provided as examples.

Our work is aimed at filling a gap that exists between the standard interaction-net model and real distributed memory schemes. These schemes depart from the traditional random access memory schemes in that they do not allow a uniform constant-time access to data stored remotely. Our long term objective is an abstract execution model for asynchronous computation adapted to both hardware components and network computing that automatically distributes the computation over the available computation units and incorporate latency and bandwidth management. We present here a simple and somewhat naive approach which is nevertheless partially successful.

We know that interaction nets are a target of choice to run functional programs in parallel [16, 15, 4, 2, 3], or to express concurrency such as in process algebras [18, 19]. Various implementations of interaction nets exist [20, 21, 1], some of which use a particular technique called geometry of interaction [22]. But all consider traditional computer architectures as a base model, with random access memory schemes. Our goal is to use micro and macro parallelism extensively – beyond the standard multi-threading "compromise" – to run generic

[1] The ia2d interpreter: `http://bitbucket.org/inarch/ia2d`

programs which have not been instrumented with hardware, network, or architecture-specific parallel constructions.

Towards this same goal, an experimental implementation of interaction nets on parallel hardware such as GPUs has been developed [10]. A variant of interaction nets called *hard interaction nets* has also been introduced to model asynchronous hardware components [14], but we do not know of any practical way to use this model to run standard programs. A compiler to hard interaction nets still has to be developed.

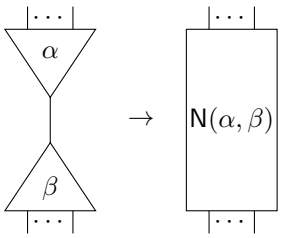
The model which we introduce is meant to incorporate spatial considerations in the most generic way. Notably, when its topological constraints are lifted, we show that it corresponds closely to the standard interaction-net model.

2 Interaction Nets

We present in this section *interaction nets* and their reduction which is based on graph-rewriting techniques. This overview should be sufficient in order to understand the programs provided as examples along with our tool. For a more detailed introduction, the reader is referred to the seminal paper by Yves Lafont [11].

An *interaction net* is a graph, with vertices called *nodes* that are labeled with *symbols*, and edges called *wires*. Each node has a *principal port* as well as a certain number of *auxiliary* ports that is fixed for a given symbol. This number of auxiliary ports is called the *arity* of the symbol. A node is typically drawn as a triangle with the principal port at the tip and the auxiliary ports on the opposite side. Node ports can be connected together with a wire. When two nodes are connected on their principal ports, we call the pair an *active pair* or a *redex*. One or both ends of a wire may also be connected to *free* ports, which do not belong to any cell. The set of free ports of a net is called its *interface*.

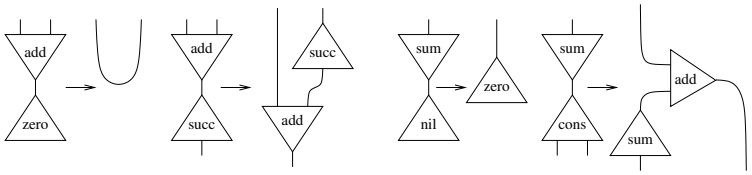
An *interaction net system* is a pair (S, R) of *symbols* and *reduction rules*. Given an interaction net built using nodes labeled with symbols S, we can rewrite the net according to the reduction rules R. A reduction rule describes how an active pair of nodes can be rewritten by removing the active pair from the graph and substituting it by a net with the same interface. Such a rule is generally represented as follows:



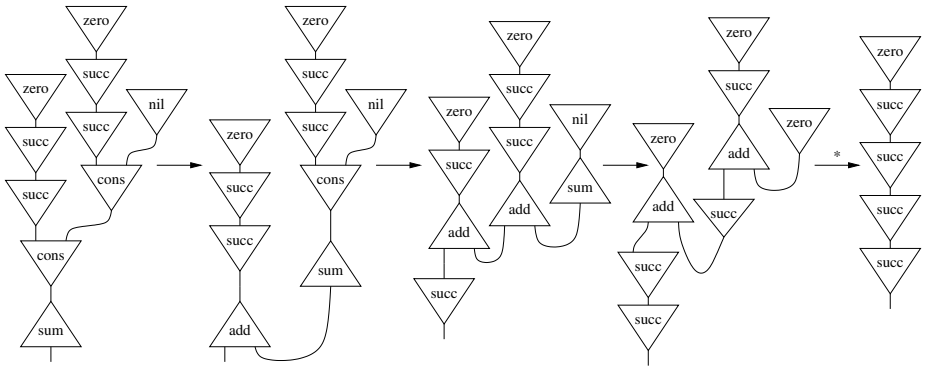
To ensure determinism only one such rule is allowed for a given pair of symbols. If $\alpha = \beta$ the right-hand side of the rule is required to be top-down symmetric. A strong confluence property, namely the diamond property, holds for the reduction of interaction nets. The order in which we choose to reduce the active pairs of the net is not important as it does not change the outcome of the reduction. Furthermore, because the reduction steps are performed locally and cannot overlap, they can be performed in parallel.

When programming with interaction nets we use labeled nodes to represent both constructors and operations. In the following example, we use the symbols zero (nullary) and succ (unary) to represent the constructors for natural numbers, cons (binary) and nil (nullary) to represent lists, and the symbols add (binary) and sum (unary) for the addition operations

on natural numbers and the sum operation on lists of naturals respectively. We can define each operation using a pair of rules as follows:



The following example computation illustrates how a given interaction net can be rewritten using the rules given above. We start with a simple net that represents a list of two natural numbers connected to a sum operation, which will compute the sum of the two numbers.



The net on the left shows the list of natural numbers $\mathsf{cons}(2, \mathsf{cons}(2, \mathsf{nil}()))$ connected to the sum operator. After two parallel reduction steps, denoted as $\rightarrow$, we obtain three redexes, namely two instances of $\mathsf{add} \bowtie \mathsf{succ}$ and $\mathsf{sum} \bowtie \mathsf{nil}$. After exhaustively applying reduction rules until no redexes are left, as denoted by $\xrightarrow{*}$, we obtain the final net, the natural number 4 represented as net.

3 Interaction Automata

3.1 Definition of the Computation Model

We start by introducing a notion of web, which defines the topology on which interaction automata will be built.

▶ **Definition 1.** A *web* is defined as a pair $W = (L, \nu)$, where L is a set of locations, called *support*, and $\nu : L \to \mathcal{P}(L)$ a map that associates a set of locations, called *neighborhood*, to every location.

Interaction agents in our model will be nodes. Their purpose is indicated by a particular symbol and they store a certain number of pointers to other locations. This number has to match with a particular *arity* that was attributed to the symbol. Locations represent positions at which two nodes are expected to interact.

▶ **Definition 2.** For a given arity-attributed set of symbols S and a set of locations L, we define *nodes* according to the syntax $n ::= \omega(l_0) \mid s(l_1, \ldots, l_k)$ where $s \in S$, k is the arity of s, and the $l_i \in L$ form a list of locations called pointers. The set of all possible nodes is written $N(S, L)$.

The ω notation can be considered as a particular "pointer-target" symbol, which contains a reference that links back to the origin of the pointer.

▶ **Definition 3.** We define the forward-reference multiset of a node as $\mathsf{f}(\omega(l_0)) = \varnothing$ and $\mathsf{f}(s(l_1, \ldots, l_k)) = \{l_1, \ldots, l_k\}$ and its backward-reference multiset as $\mathsf{b}(\omega(l_0)) = \{l_0\}$ and $\mathsf{b}(s(l_1, \ldots, l_k)) = \varnothing$. These notations are straightforwardly extended to multisets of nodes through multiset union.

▶ **Definition 4.** Given a set of symbols S, a *domain* $D \subseteq L$ and a disjoint *interface* $I \subseteq L$, we define $\Gamma(S, D, I)$ as the set of maps from locations to multisets of nodes $\mu : L \to \mathcal{M}(N(S, L))$, called *configurations*, such that the cardinal of $\mu(l)$ is respectively 2 if $l \in D$, or 1 if $l \in I$, or 0 otherwise. Additionally we require that for all $l_s, l_t \in L$ the multiplicity of l_t in $\mathsf{f}(\mu(l_s))$ matches the one of l_s in $\mathsf{b}(\mu(l_t))$.

The latter constraint ensures that every forward-reference attached to a symbol node matches with a backward-reference attached to an ω-node.

▶ **Definition 5.** A *topological configuration* of a web $W = (L, \nu)$ with symbols in S is the combination of a domain D, an interface I and a configuration $\gamma \in \Gamma(S, D, I)$ that satisfies neighborhood compatibility, i.e., $l_t \in \mathsf{f}(\gamma(l_s)) \Rightarrow l_t \in \nu(l_s)$, for all $l_s, l_t \in L$.

We will rely on an abstract reduction scheme to automatically endow any given web with a transfer function from configurations to configurations.

▶ **Definition 6.** An *abstract transition scheme* over a set of symbols S is a binary relation $\mu_l \to \mu_r$ over configurations $\mu_l \in \Gamma(S, D_l, I)$ and $\mu_r \in \Gamma(S, D_r, I)$ which is invariant upon permutations of locations in L. The sets D_l, D_r and I are arbitrary disjoint subsets of L.

D_l and D_r represent respectively the sets of freed and allocated locations during a reduction step that is defined by the abstract transition scheme. If we additionally require that D_l has cardinal 1 and that the singleton $\mu(l)$ stored at any $l \in I$ is an ω-node, the scheme is called *atomic*.

▶ **Definition 7.** An *interaction automaton* is defined as a quadruple $A = (L, \nu, S, \to)$ where $W = (L, \nu)$ is a chosen web, S is a set of symbols, and $\to$ is an abstract transition scheme over S.

▶ **Definition 8.** Given an interaction automaton A, a *parallel transition* $\gamma_l \xrightarrow{A}\!\!\!\!\!\rightarrow \gamma_r$ occurs between two topological configurations of its web, $\gamma_l \in \Gamma(S, D_l, I)$ and $\gamma_r \in \Gamma(S, D_r, I)$, if both configurations are pointwise multiset unions of configurations $\gamma_l = \bigoplus_i \mu_i^l \oplus \mu$ and $\gamma_r = \bigoplus_i \mu_i^r \oplus \mu$, such that each pair of sub-configurations satisfy the abstract transition scheme relation $\mu_i^l \to \mu_i^r$.

If the chosen abstract transition scheme is atomic, in the absence of topological constraints (unrestricted neighborhoods and an infinite number of locations), we can guarantee that the parallel transition relation, despite being non-deterministic, satisfies the diamond property up to a relocation of cells.

Bounded Interaction Automata and Interaction Grids *Bounded interaction automata* are interaction automata for which the sizes of neighborhoods are globally bounded by a constant.

Example: *Interaction grids* of dimension d and neighborhood radius m are denoted by $\mathbb{G}_m^d$. They are particular interaction automata whose sets of locations are defined as $L = \mathbb{Z}^d$ and neighborhood maps as $\nu(\vec{x}) = \{\vec{x} + \vec{u}, |\vec{u}| \leq m\}$. We chose to work with the Manhattan distance in our implementation.

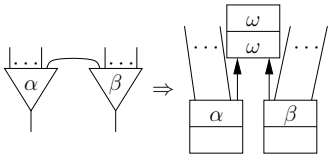
3.2 Implementation of Interaction Nets on Interaction Automata

We now show that interaction-net computation can be performed within the abstract interaction automata computation model without topological constraints.

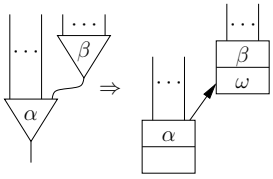
With topological constraints, nothing guarantees that allocations are always possible within the appropriate neighborhoods nor specify how the allocations should be done. They can be performed randomly, but the reduction is of course likely to block due to a local lack of space on webs with small connectivity. The allocations could also be done strategically if we rely on topological knowledge or external ways to gauge the occupancy of the web and its capacity. Our experiments however tend to show that a simple allocation strategy is enough in order to run interaction-net programs of a particular complexity class on webs with a matching connectivity.

The implementation of a given interaction-net system is quite straightforward. A glimpse at the output of our tool should be sufficient to illustrate the general principles which we sketch here.

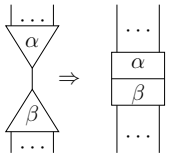
- Cells containing two ω-nodes are used to translate wires between two auxiliary ports.



- Mixed cells containing an ω-node and a symbol node are used to translate wires between an auxiliary port and a principal port.



- Cells containing two symbol nodes are used to translate wires from a principal port to another principal port.



- An interface cell containing a single ω-node is used to encode a wire that links one auxiliary port of an interaction-net node to one free port.
- An interface cell containing a single symbol node is used to encode a wire that links one principal port of an interaction-net node to one free port.
- The particular case of a wire that links two free ports is encoded by mapping two corresponding interface locations to nodes annotated with a particular *forwarder* symbol. Both nodes forward to the same additional location designated for this wire that contains two target ω-nodes.

Interaction-net node data is stored in the cell dedicated to the wiring of its principal port. Pointers are then set up such that related ports and wires reference each other.

Among the set of symbols S, additionally to the symbols of the chosen interaction-net system, as mentioned, a particular forwarder symbol (unary), which we denote by & in our

tool, can be used to lengthen the wires if necessary. Each occurrence contains a pointer to the next hop on the path to the destination cell. In particular it is used in the transition schemes associated to right-hand sides that connect two interface ports with a wire directly.

Given the above encoding for nets, reductions rules are turned to abstract transition schemes $\mu_l \to \mu_r$ easily. The left-hand side of an interaction-net rule is encoded as a configuration μ_l with any single-location domain that stores the two interaction-net nodes that are part of the redex, and any interface that contain the required number of auxiliary ports present in the redex. The right-hand side is encoded as a configuration μ_r with a domain whose cardinal corresponds to the number of wires, except for those which connect a free port to a port of an interaction-net node. The usual constraints on interaction-net rules, including the symmetry of the rules that define interaction between two identical symbols, ensure that the defined abstract transition scheme is invariant upon permutations of the locations which were arbitrarily chosen as port identifiers.

Assuming neighborhoods are "sufficiently large to always contain free cells", the interaction-net reduction can be simulated on an interaction automaton. Different topologies are expected to lead to different performances. We implemented an interpreter for the planar topology $\mathbb{G}^2_m$.

3.3 Limits

For grids $\mathbb{G}^d_m$ (and similar topologies for which the size of iterated n-neighborhoods is polynomially bounded in n), due to data propagation constraints, the number of spawned redexes after a parallel running time t is necessarily bounded by a polynomial of degree d, the dimension of the grid. We cannot hope that the asymptotic speed-up offered by the parallelization will exceed this limit.

We have not yet investigated other topologies. Some multiscale webs including limited-bandwidth but long-distance links seem sufficiently realistic and could provide exponential speed-ups in many cases.

4 Implementation

The source code of ia2d is available at `https://bitbucket.org/inarch/ia2d`. For installation and usage instructions please consult the `README.md` file included in the source repository.

4.1 General Design Principles

Our automaton implementation works on a 2-dimensional grid of cells, each of which can hold up to two nodes. Parallel transitions of the automaton are performed in a loop. Each of these parallel transitions is preceded by a migration pass and an input/output pass as described hereafter.

Migration The migration pass rearranges the cells in a way that avoids too high densities of cells, while at the same time keeping connected nodes within range of each other. As currently implemented the strategy favors migrations to locations whose neighborhoods contain the most empty cells. The final selection of a destination location among identically weighted candidates relies on randomization to avoid directional bias. Like the rest of the computation, the implemented migration strategy is local. Decisions are made by looking only at cells within a fixed radius.

Input and Output For input and output the automaton uses special nodes in and out. During the input/output pass, the in nodes lazily insert nodes from the supplied program onto the grid when their interaction with other nodes is required. Conversely, the out nodes take nodes which are part of the normal form off the grid to produce a result. Currently, the collected result is printed at once on the terminal when the automaton stops, but input and output could also be streamed in real time.

Parallel Transitions The main reduction pass rewrites cells that contain interaction-net redexes, as defined by the user, assuming enough free cells are locally available to store the right-hand sides of the reduction rules. The combination of a migration pass, an input/output pass and a parallel transition is repeated until the normal form of the input net has been entirely collected or an error such as memory exhaustion occurs.

Resource Stress The size of the grid has no effect if it is sufficiently large, but if the memory capacity is really tight the grid may become too densely populated, up to saturation. The firing of redexes that require allocations are in this case delayed until some space is made available locally by other reductions. Parallel execution which generally requires more space than sequential execution is therefore affected and gradually sequentializes to some extent. It may also occur that the memory capacity is simply too small for the computation, in which case the reduction will fail as it would on a normal computer.

Failures Reduction can stop half-way if space is unavailable or cannot be freed locally. We developed this is software in order to understand when it happens in practice, how it can be avoided, and what theoretical results are necessary. For comparison, in traditional computation, the practical answer to a shortage of resources, (i.e., an insufficient amount of memory to run a certain algorithm) is simply to "fail" rather than to try to adapt to the situation. In the more tricky case of parallel computation, where a simple amount of memory is not the only parameter, we still have to decide whether it is worthwhile to adapt.

4.2 Input Language

As input language we use a variant of the Pin language [8]. The Pin language is a flat representation of graphs. We write the rule for an active pair between nodes α and β as $\alpha(x_1, \ldots, x_n) \bowtie \beta(y_1, \ldots, y_m) \Rightarrow N$ where N is a comma-separated list of *net components*. Net components are either:

- wires $x \sim y$,
- connections to nodes $x \sim \gamma(y_1, \ldots, y_k)$,
- or active pairs $\gamma_1(x_1, \ldots, x_n) \sim \gamma_2(y_1, \ldots, y_m)$.

Moreover, any variable used in a reduction rule should occur exactly twice in this rule. For example, one rule used to define binary addition is the following:

$$\mathsf{add}(y, r) \bowtie \mathsf{succ}(x) \Rightarrow r \sim \mathsf{succ}(t), \mathsf{add}(y, t) \sim x$$

In the actual input to the program we write $\bowtie$ as `><`, $\sim$ as `~`, $\Rightarrow$ as `=>` and we separate net components with commas and rules with semicolons. The following source code defines addition on natural numbers in unary encoding and the sum over a list of natural numbers:

```
add(y,r) >< zero() => r~y;
add(y,r) >< succ(x) => r~succ(t), add(y,t)~x;
```

```
sum(r) >< nil() => r~zero();
sum(r) >< cons(x, xs) => x~add(t, r), xs~sum(t);
```

A complete source file consists of an optional header of import statements and a list of rules and input nets. By storing the above library in a file called `nat_unary.inet`, we can write the computation of the sum of the natural numbers 4 and 2 as follows:

```
import "nat_unary.inet"
x ~ succ(succ(succ(succ(zero())))),
y ~ succ(succ(zero)),
cons(x, cons(y, nil())) ~ sum(r)
```

In this net, r is the only variable which occurs only once. It is the name associated to the result of this computation.

Along with the `ia2d` source code, a number of code examples are provided in the `examples` directory. The reader should refer to these files for more involved programming examples.

Synthesized Rules Most interaction-net programs use δ nodes for duplicating and ϵ nodes for erasing parts of nets. The implementation of the automaton also makes use of forwarder nodes to connect two auxiliary ports or to extend the connections between distant nodes. The rules for δ, ϵ are synthesized and need not be provided by the programmer. For any user-defined symbol l of arity k the following rules are generated.

$$\begin{aligned} \epsilon() \bowtie l(z_1, \ldots, z_k) &\Rightarrow \epsilon() \sim z_1, \ldots, \epsilon() \sim z_k \\ \delta(x, y) \bowtie l(z_1, \ldots, z_k) &\Rightarrow x \sim l(x_1, \ldots, x_k), y \sim l(y_1, \ldots, y_k), \\ &\qquad z_1 \sim \delta(x_1, y_1), \ldots, z_k \sim \delta(x_k, y_k) \end{aligned}$$

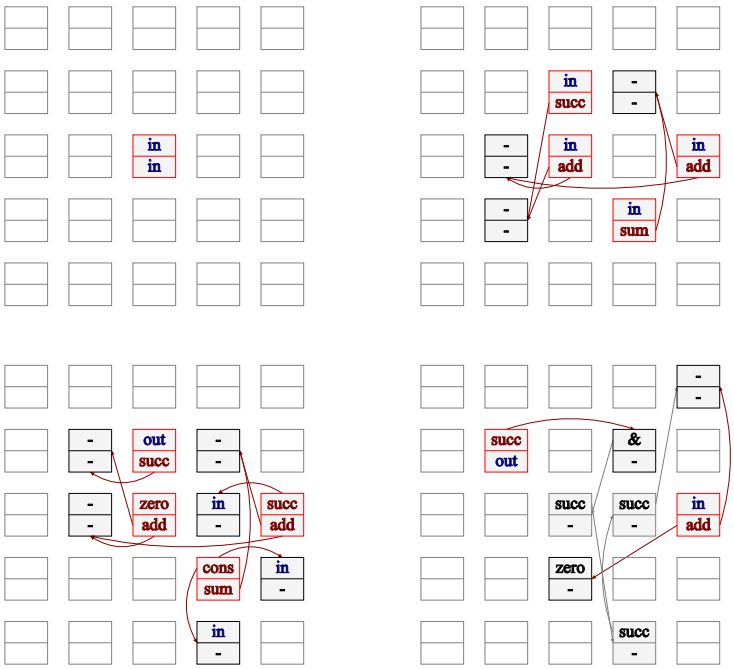
In order to duplicate cyclic data structures the rules $\epsilon() \bowtie \epsilon() \Rightarrow$ (with an empty list as right-hand side) and $\delta(x_1, y_1) \bowtie \delta(x_2, y_2) \Rightarrow x_1 \sim x_2, y_1 \sim y_2$ are provided as well.

4.3 Resource Usage Reports

When invoked on the command-line, besides the normal form of the provided net, `ia2d` outputs the number of individual transition steps and parallel reductions passes which were performed, along with more detailed reports for every reduction pass. (Please note that the output is truncated for brevity and might change in future versions.)

```
$ ia2d examples/bsort_example.inet --grid-size=16x16
SEQ STEPS: 664
PAR STEPS: 48
PASSES BREAKDOWN:
  ...
  1 x 27 fired
  1 x 29 fired
  1 x 29 fired and 1 delayed
  5 x 30 fired
  ...
DURATION:
  0.537617s
```

From the above output we can tell that `ia2d` had to perform 48 parallel reduction passes to reduce the input program. If we furthermore supply a parameter for the `--svg` flag, the intermediate states of the grid are made available as SVG files. As an illustration, we provide below four snapshots of the SVG output for the `sum.inet` program, which is available in the repository.



In the first picture, the grid is initially populated by just two in nodes which form the only redex of the input program. After three steps, as seen in the second picture, the grid contains more redexes and some input is still to be written onto the grid by the in nodes. The transition between the second and third picture is a migration pass, where existing in nodes are replaced with symbols from the input, new in nodes are added, and one different cell is migrated to a different position. The last picture shows the grid in a state where part of the output is already taken off the grid by an out node, while at the same time the final add node is still waiting for additional input data.

4.4 Results

The following table shows the number of parallel reduction passes needed to sort the leaves of a full binary tree with the *bitonic sorter* on the 2-dimensional grid model and compares it to the sequential number of steps that would be required in the standard interaction-net model.

number of leaves	sequential steps	parallel steps (average)	speed-up
2	23	10.0	2.30
4	70	15.0	4.67
8	212	22.0	9.64
16	620	31.9	19.44
32	1740	52.8	32.95

The size of the grid was chosen sufficiently large not to slow the computation. We observe a quick increase of speed-ups as long as the neighborhood radius does not influence the allocation too much. Beyond a certain input size we see that the potential quadratic increase of the speed-up is not yet met on this example with a naive migration strategy.

5 Conclusion

We introduced a parallel computation model with an entirely localized reduction on top of a memory scheme with a limited connectivity and a locally bounded storage and computation capacity. We run functional programs on this model and showed that reasonable memory management strategies can also be implemented locally.

We plan to incorporate new features such as nested pattern matching [9, 7] to our input language and support more traditional programming syntaxes.

Despite the relatively smooth executions obtained with a very simplistic memory allocation strategy, there is plenty of room for efficiency improvements that would reduce the total migration costs. Integration with complexity-analysis techniques such as [13, 6] should help to implement really accurate allocation strategies.

We only considered 2-dimensional uniform grid supports in our experimentation. In the future, we would also like to support 3 or n dimensional grids and more elaborate topologies.

A last remaining challenge is to use a minimal set of symbols and reduction rules. We know that there exist interaction-net systems with a very restricted set of symbols that are universal [12]. In particular preliminary investigations have been made concerning the usage of such minimal sets of symbols to specifically encode functional programs by means of linear logic [17, 4, 5].

Acknowledgments

This work was partially supported by FWF (Austrian Science Fund) project number P 25781-N18.

References

1 Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. The bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, 1996.

2 Horatiu Cirstea, Germain Faure, Maribel Fernandez, Ian Mackie, and François-Régis Sinot. From functional programs to interaction nets via the rewriting calculus. *Electronic Notes in Theoretical Computer Science*, 174(10):39–56, 2007.

3 Maribel Fernández, Ian Mackie, Shinya Sato, and Matthew Walker. Recursive functions with pattern matching in interaction nets. *ENTCS*, 253(4):55–71, 2009.

4 Stéphane Gimenez. *Programmer, calculer et raisonner avec les réseaux de la logique linéaire.* PhD thesis, 2009.

5 Stéphane Gimenez. Towards generic inductive constructions in systems of nets. In *13th International Workshop on Termination (WST 2013)*, page 51, 2013.

6 Stéphane Gimenez and Georg Moser. The complexity of interaction. In *POPL*, pages 243–255. ACM, 2016.

7 Abubakar Hassan, Eugen Jiresch, and Shinya Sato. An implementation of nested pattern matching in interaction nets. *arXiv preprint arXiv:1003.4562*, 2010.

8 Abubakar Hassan, Ian Mackie, and Shinya Sato. Interaction nets: programming language design and implementation. *Electronic Communications of the EASST*, 10, 2008.

9 Abubakar Hassan and Shinya Sato. Interaction nets with nested pattern matching. *Electronic Notes in Theoretical Computer Science*, 203(1):79–92, 2008.

10 Eugen Jiresch. Towards a GPU-based implementation of interaction nets. In *DCM*, pages 41–53, 2014.

11 Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 95–108. ACM, 1989.

12 Yves Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

13 Ugo Dal Lago. A short introduction to implicit computational complexity. In *Lectures on Logic and Computation – ESSLLI 2010, ESSLLI 2011*, pages 89–109, 2011.

14 Sylvain Lippi. Universal hard interaction for clockless computation. dem glücklichen schlägt keine stunde! *Fundamenta Informaticae*, 91(2):357–394, 2009.

15 Ian Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247(1-2):83–140, 2000.

16 Ian Mackie. Efficient lambda-evaluation with interaction nets. In *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169, 2004.

17 Ian Mackie and Jorge Sousa Pinto. Encoding linear logic with interaction combinators. *Information and Computation*, 176(2):153–186, 2002.

18 Damiano Mazza. Multiport interaction nets and concurrency. In *CONCUR*, pages 21–35, 2005.

19 Damiano Mazza. *Interaction Nets: Semantics and Concurrent Extensions*. PhD thesis, 2006.

20 Jorge Sousa Pinto. Sequential and concurrent abstract machines for interaction nets. In *FOSSACS*, pages 267–282. Springer-Verlag, 2000.

21 Jorge Sousa Pinto. Parallel evaluation of interaction nets with MPINE. In *RTA*, pages 353–356, 2001.

22 Jorge Sousa Pinto. Parallel implementation models for the lambda-calculus using the geometry of interaction. In *TLCA*, pages 385–399, 2001.