Parallel evaluation of interaction nets: some observations and examples
(Work-in-progress)

Ian Mackie and Shinya Sato

Abstract. Interaction nets are a particular kind of graph rewriting system that have many properties that make them useful for capturing sharing and parallelism. There have been a number of research efforts towards implementing interaction nets in parallel, and these have focused on the implementation technologies. In this paper we investigate a related question: when is an interaction net system suitable for parallel evaluation? We observe that some nets are cannot benefit from parallelism (they are sequential) and some have the potential to be evaluated in a highly parallel way. This first investigation aims to highlight a number of issues, by presenting experimental evidence for a number of case studies. We hope this can be used to help pave the way to a wider use of this technology for parallel evaluation.

1 Introduction

Interaction nets are a model of computation based on a restricted form of graph rewriting: the rewrite rules must be between two nodes on the left-hand side, be local (not change any part of graph other than the two nodes), and there must be at most one rule for each pair of nodes. These constraints have no impact on the expressive power of interaction nets (they are Turing complete), but they offer a very useful feature: they are confluent by construction. Taken with the locality constraint they lend themselves to parallel evaluation: all rewrite rules that can apply can be rewritten in one parallel step.

The question that we propose in this paper is: when is a particular interaction net system well suited for parallel evaluation. More precisely, are some interaction nets "more parallel" than others? A question that naturally follows from this is can we transform a net so that it is more suited for parallel evaluation. Once we have understood this, we can also ask the reverse question: can a net be made sequential? The purpose of this paper is to make a start to investigate these questions, and we begin with an empirical study of interaction systems to identify when they are suitable for parallel evaluation or not.

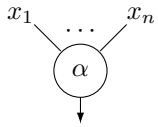
We take a number of typical examples (some common ones from the literature together with some new ones we made up for this paper) to see if they benefit from parallel evaluation. In addition, we make some observations about how programs can be transformed so that parallelism is more useful. Using these examples, we give some heuristics for getting more parallelism out of an interaction net system.

Related work. There have been a number of studies for the parallel implementation of interaction nets: Pinto [6] and Jiresch [3] are two examples. In these works it is the implementation of a given net that has been the focus. Here we are interested in knowing if a net is well suited for parallel evaluation or not.

Structure. In the next section we recall the definition of interaction nets, and describe the notion of parallel evaluation that we are interested in. Through examples we motivate the ideas behind this work. In Section 3 we give a few small case studies to show how parallelism can have a significant impact on the evaluation of a net. In Section 4 we give a short discussion and conclude in Section 5.

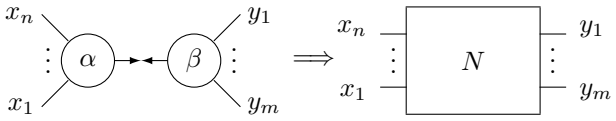
2 Background and Motivation

In the graphical rewriting system of interaction nets [4], we have a set Σ of *symbols*, which are names of the nodes in our diagrams. Each symbol has an arity ar that determines the number of *auxiliary ports* that the node has. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n + 1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.



Nodes are drawn variably as circles, triangles or squares. A *net* built on Σ is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.

Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction nets analogue of a redex. A rule $((\alpha, \beta) \Longrightarrow N)$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



The most powerful property of this system is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length (in fact they are permutations). This has practical consequences: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer. It is the latter feature that we develop in this paper.

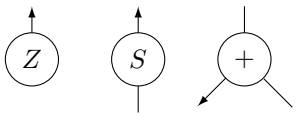
We define some notions of nets and evaluation. A net is called *sequential* if there is at most one active pair that can be reduced at each step. We say that a

net is evaluated sequentially if one active pair is reduced at each step. For our notion of parallel evaluation, we require that all active pairs in a net are reduced simultaneously, and then any redexes that were created are evaluated at the next step. We do not bound the number of active pairs that can be reduced in parallel. We remark that the number of parallel steps will always be less than or equal to the number of sequential steps (for a sequential net, the number of steps is the same for sequential and parallel evaluation).

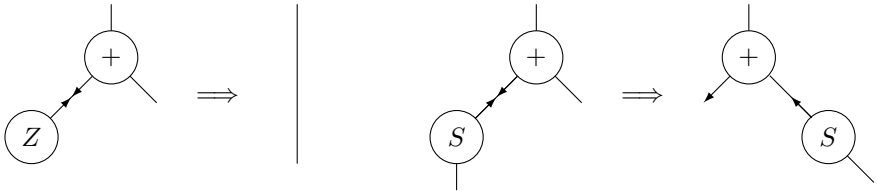
As an example, consider unary numbers with addition. We represent the following term rewriting system

```
add(Z,y)    = y
add(S(x),y) = add(x,S(y))
```

as a system of nets with agents Z, S, $+$:



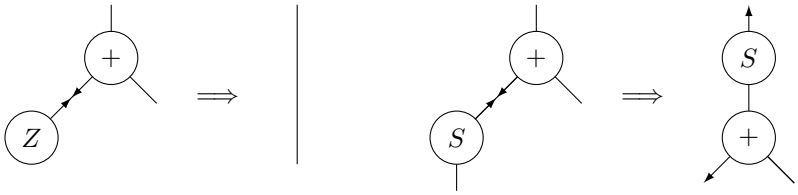
together with two rewrite rules:



We observe that addition of two numbers is sequential: at any time there is just one active pair, and reducing this active pair creates one more active pair, and so on. In terms of cost, reducing `add(n,m)` requires $n + 1$ interactions. If we consider the net corresponding to the term $add(add(m, n), p)$, then the system is sequential, and the costs are now $2m+n+2$. Using associativity of addition, the situation changes significantly. The net corresponding to $add(m, add(n, p))$ has sequential cost $m+1+n+1 = m+n+2$, and parallel cost $max(m+1, n+1)$. This is significantly more efficient sequentially, and moreover is able to benefit from parallel evaluation. The example becomes even more interesting if we change the system to an alternative version of addition:

```
add(Z,y)    = y
add(S(x),y) = S(add(x,y))
```

The two interaction rules are now:



Unlike the previous system, the term $add(add(m,n),p)$ already has scope for parallelism. The sequential cost is now $2m+n+2$ and the parallel cost is $m+n+2$. But again, if we use associativity then we can do even better and achieve sequential cost $m+n+2$ and parallel cost $max(m+1,n+1)$ for the term $add(m,add(n,p))$.

These examples illustrate that some nets are sequential; some nets can use properties of the system (in this case associativity of addition) to get better sequential and parallel behaviours; and some systems can have modified rules that are more efficient, and also more appropriate to exploit parallelism. The next section gives examples where there is scope for parallelism in nets.

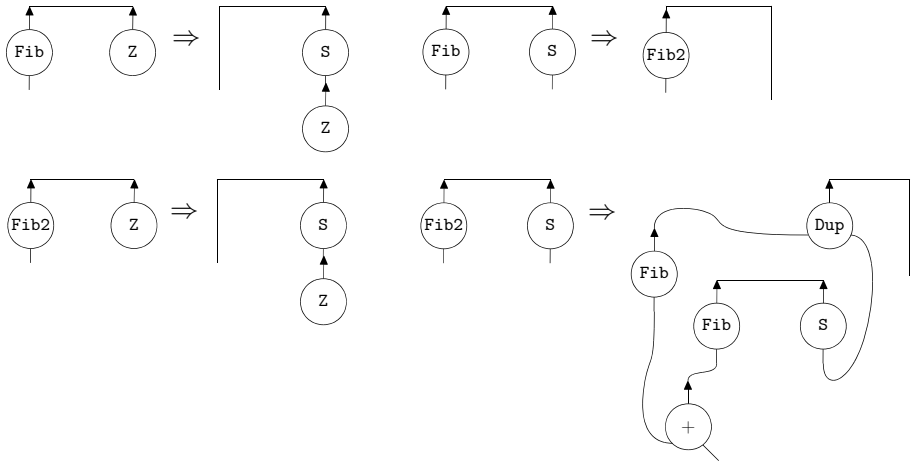
3 Case studies

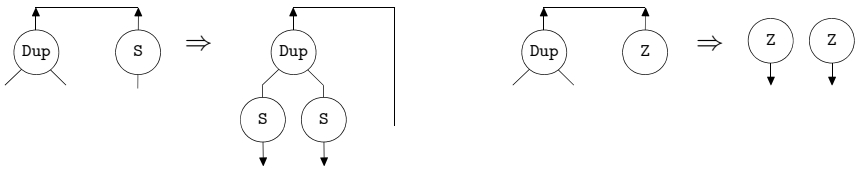
The previous arithmetic example demonstrates that some systems are more useful than others for parallel evaluation. In this section we give some empirical case studies for a number of different systems to show that when a suitable system can be found, the parallel evaluation gives significantly better results than sequential evaluation.

Fibonacci. The Fibonacci function is a good example where many recursive calls generate a lot of possibilities for parallel evaluation. We build the interaction net system that corresponds to the term rewriting system:

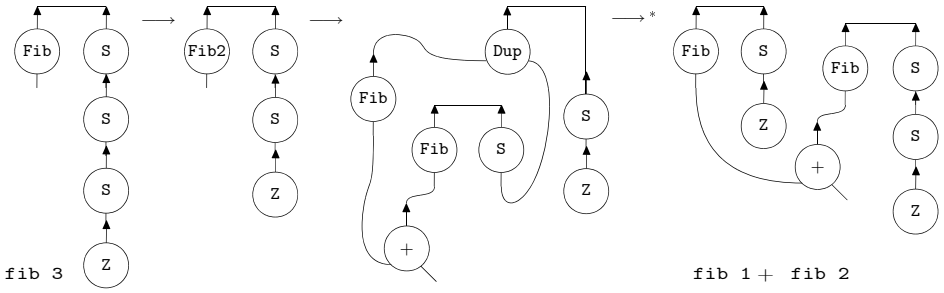
```
fib 0 = fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Using a direct encoding of this system together with addition defined previously, we can obtain an interaction system:



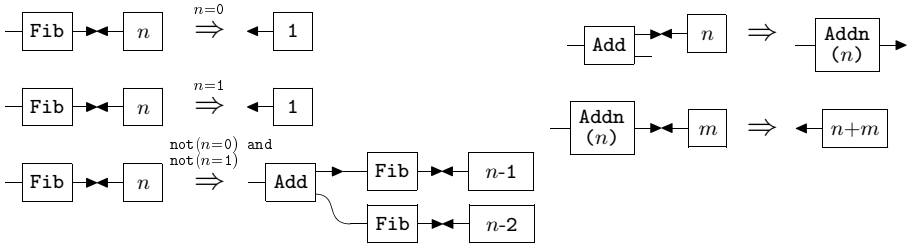


The following is an example of rewriting:



With respect to the two versions of the addition operation introduced in Section 2, we call the former a *batch* operation, which returns the computational result after finishing processing all of the given data, and the latter a *streaming* operation, which computes one (or a small number of) elements of the given data and returns partial parts of the computational result immediately. The graphs in Figure 1 show the number of interactions in each version, where we plot sequential steps against parallel steps to indicate the rate of growth of each one. Both graphs demonstrate that the sequential computation is exponential, while the parallel one is quadratic. We remark that, in the parallel execution, the numbers of steps with the streaming operation are less than a half of the numbers with the batch operation. This result is illustrated in the third graph in the figure.

By allowing attributes as labels of agents, we can include integer numbers in agents. In addition, we can use conditional rewritings, preserving the one-step confluence, when these conditions on attributes are disjoint. In this case, the system of the Fibonacci function is written as follows:



There is very little difference between the load balances of `fib` $(\mathtt{n}-\mathtt{1})$ and `fib` $(\mathtt{n}-\mathtt{2})$, and thus this system gives the following graph, demonstrating that the growth rate for parallel computation is linear, while the sequential rate is exponential:

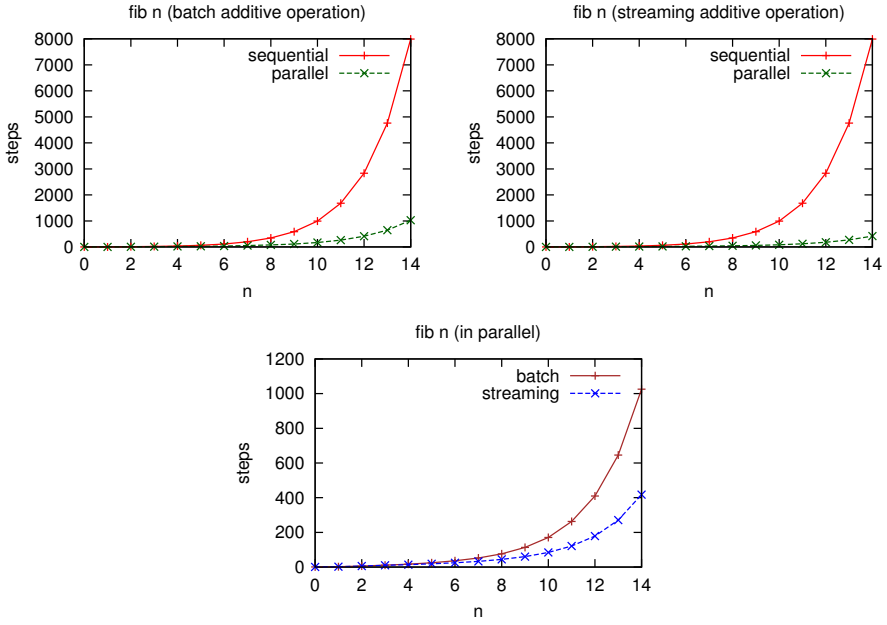
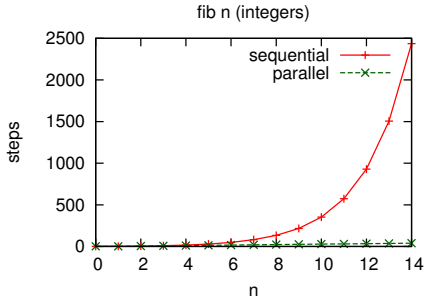
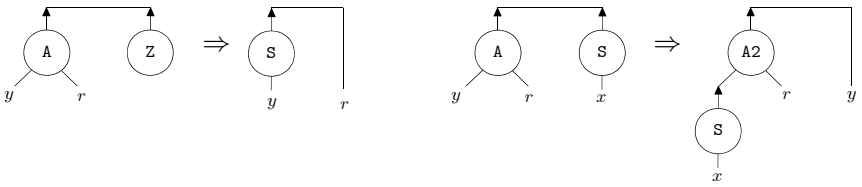
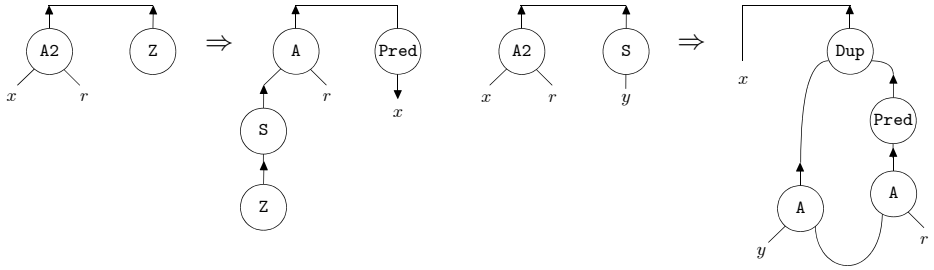


Fig. 1. Comparing batch and streaming operations

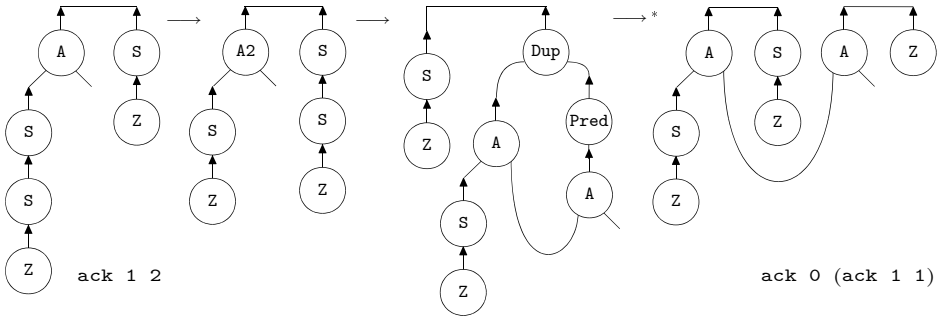


Ackermann. The Ackermann function is defined by three cases: `ack 0 n = n+1`, `ack m 0 = ack (m-1) 1`, and `ack m n = ack (m-1) (ack m (n-1))`. We can build the interaction net system on the unary natural numbers that corresponds to the term rewriting system as follows:





where the agent `Dup` duplicates `S` and `Z` agents. The following is an example of rewriting:



When we use numbers as attributes, the system can be written as:

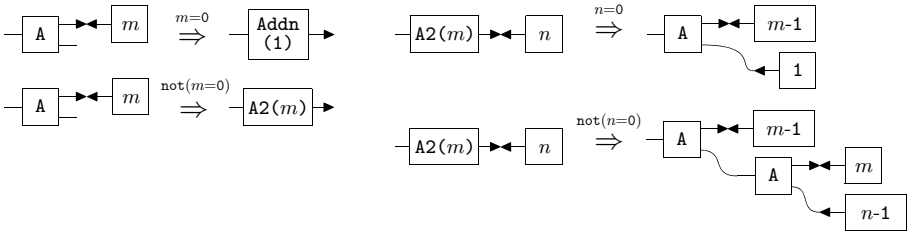
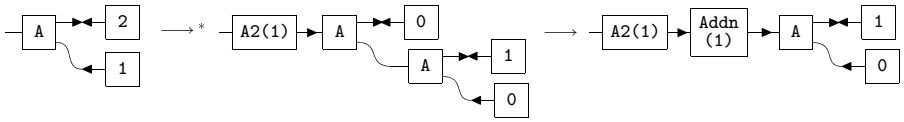


Figure 2 shows the number of interactions in the cases of (a) unary natural numbers and (b) integer numbers, where we plot sequential steps against parallel steps to indicate the rate of growth of each one. Unfortunately, in Figure 2 (b), there is no significant difference in the sequential and the parallel execution, and thus there is no possibility of the improvement by the parallel execution. This is because the `Addn` agent works as the batch operation, thus it waits for part of the result. For instance, after the last step in the following the computation step `ack 2 1`, the `Addn`(`1`) agent, which is the result of `ack 0` (`ack 1 0`), waits the computational result of `ack 1 0`. However, the computation of `A2` should proceed because the result of the `Addn`(`1`) will be more than 0.



(a) unary natural numbers (b) integers

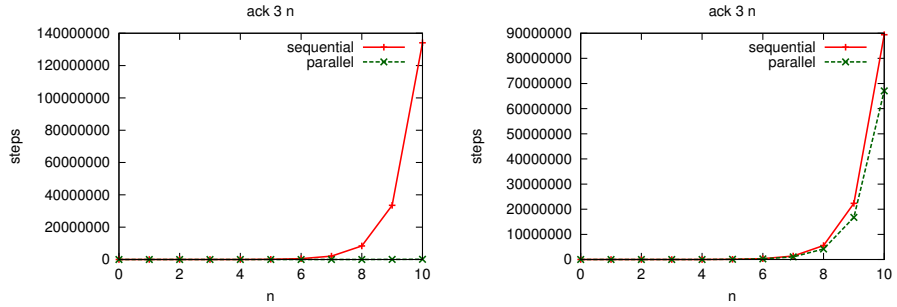
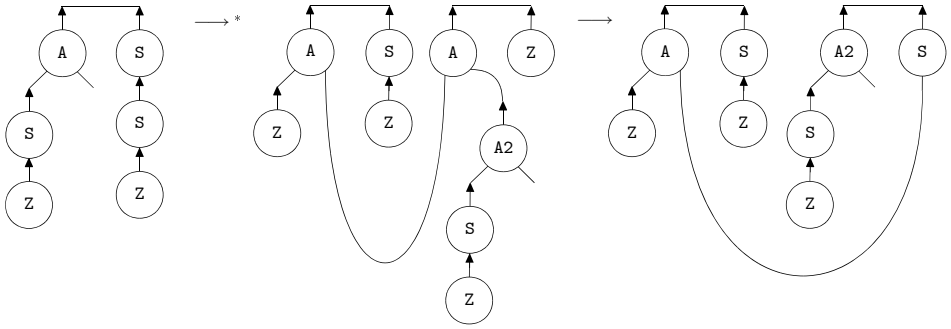
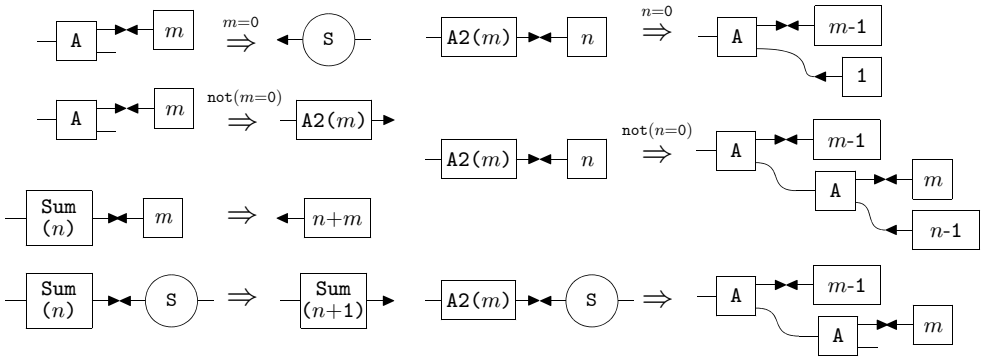


Fig. 2. Benchmarks of the execution of Ackermann function in sequential and parallel

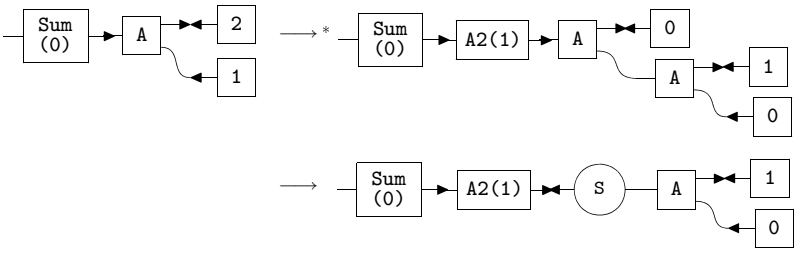
On the other hand, in the case of the computation on unary natural numbers, the `A2` interacts with the streaming result of `ack 0` (`ack 1 0`):



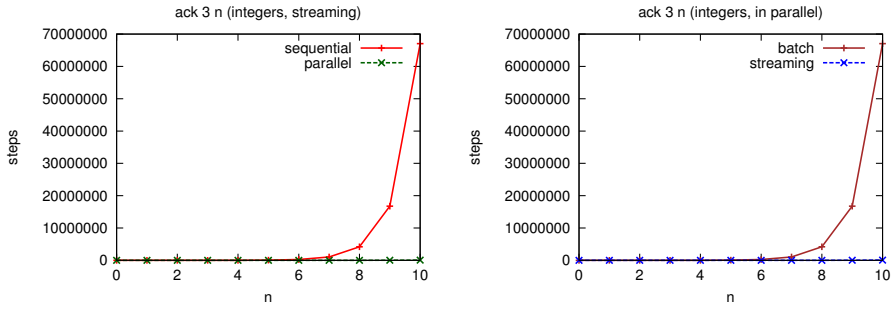
Here, borrowing the `S` agent to denote numbers greater than 0, we change the rules, especially in the case of `ack 0 n`, `Addn` into `S` as follows:



Thanks to the introduction of the `S` agent, `A2` can be processed without waiting for the result of `ack 1 0`. This therefore gives a streaming operation:



In addition, the benchmark graph shows that the improved system is more efficient and more appropriate to exploit parallelism:

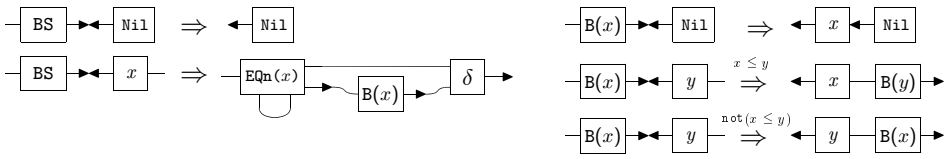


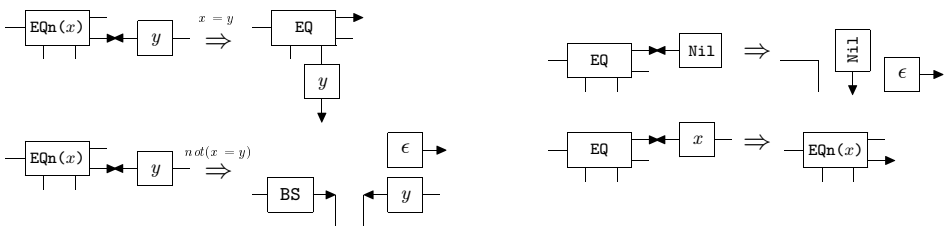
To summarise this section, a system can exploit parallelism by changing some batch operations into streaming ones. We leave as future work the criteria to determine when this transformation can benefit from parallelism.

Sorting. Bubble sort is a very simple sorting algorithm that can benefit from parallel evaluation in interaction nets. One version of this algorithm, written in Standard ML [5], is as follows:

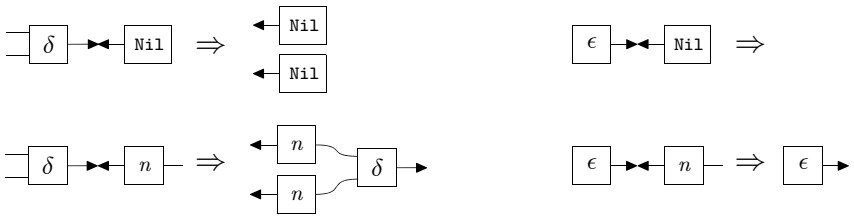
```
fun bsortsub (x::x2::xs) =
  if x > x2 then x2::(bsortsub (x::xs)) else x::(bsortsub(x2::xs))
| bsortsub x = x

fun bsort t =
  let val s = bsortsub t
   in if t=s then s else bsort s
  end;
```

Using a direct encoding of this program, we obtain the interaction system:

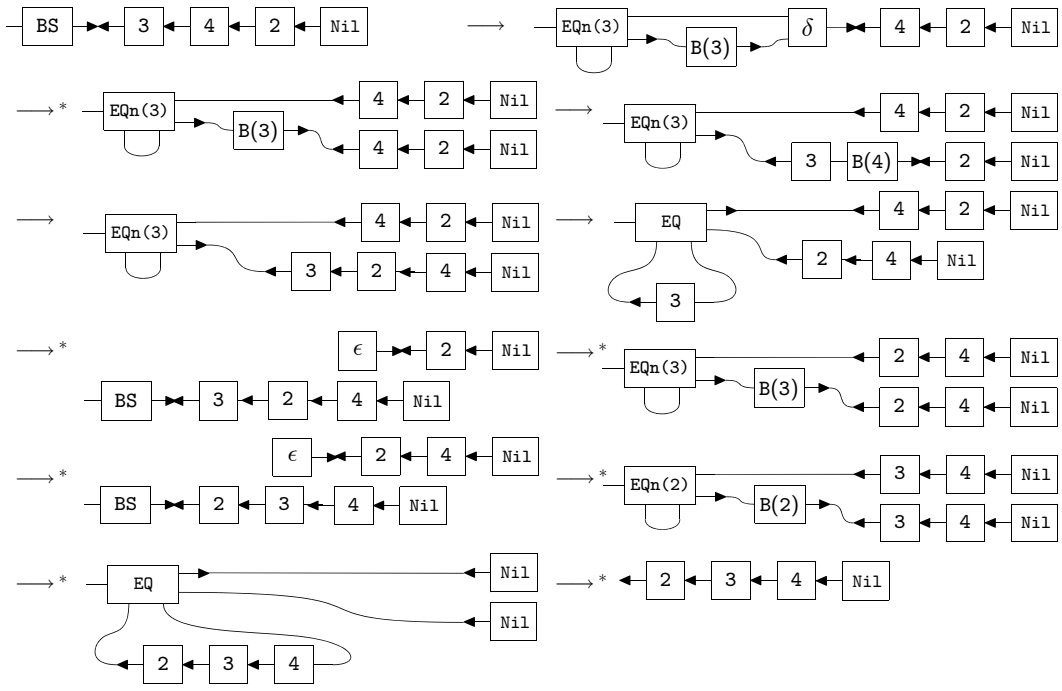




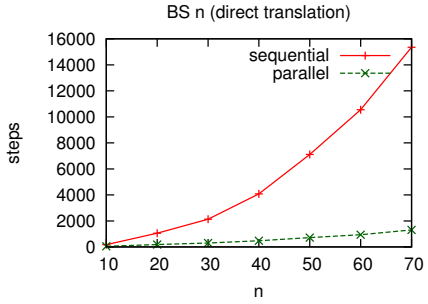
where the δ and ϵ agents are defined as a duplicator and an eraser:



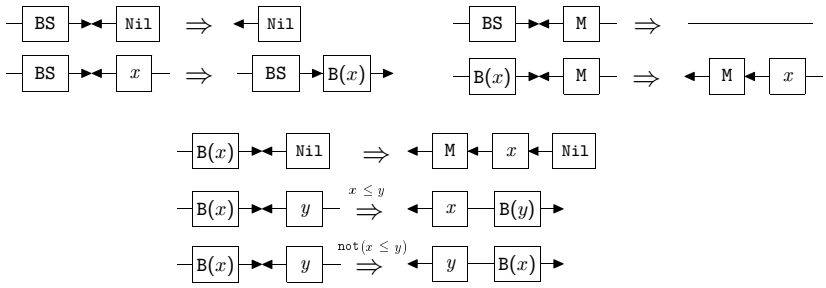
For instance, a list $[3, 4, 2]$ is sorted as follows:



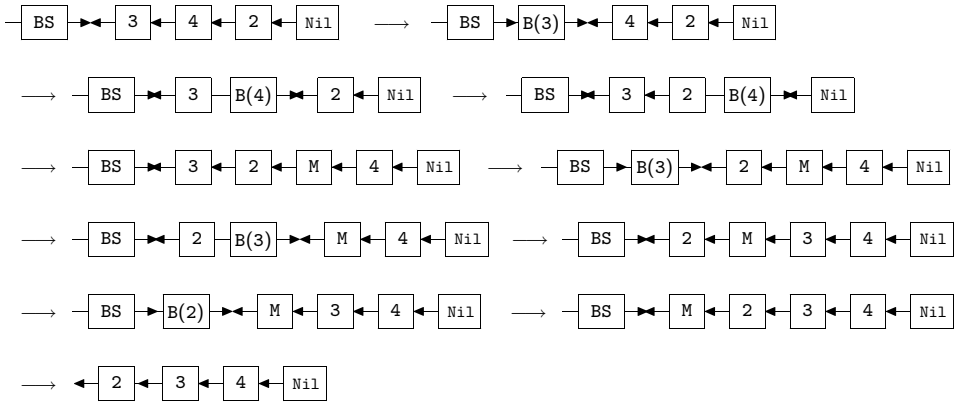
This system shows that parallel bubble sorting is linear, whereas sequential evaluation is quadratic, as indicated in the graph below.



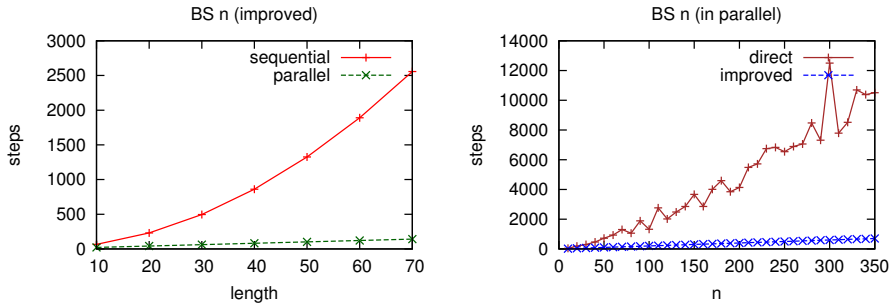
However, it contains the equality test operation by `EQ` and `EQn` to check whether the sorted list is the same as the given list. In comparison to the typical functional programming languages, interaction nets require copying and erasing of lists for the test that can cause inefficient computation. Moreover, the sorting process is applied to the sorted list by `B` again and again. Taking into account that the `B` moves the maximum number in the given unsorted list into the head of the sorted list, we can obtain a more efficient system:



For instance, a list $[3, 4, 2]$ is sorted as follows:



The system reduces the number of computational steps significantly, and gives the best expected behaviour as follows:



Summary/discussion. All these examples show the scope for harnessing parallelism from an empirical study: some systems do not benefit, whereas others allow quadratic computations be executed in linear parallel complexity. However, these results give a flavour of the potential, and do not necessarily mean that they can be implemented like this in practice.

4 Discussion

In this section we examine the potential of parallelism illustrated by the graphs in Section 3, by using a multi-threaded parallel interpreter of interaction nets, called *Inpla*, implemented with gcc 4.6.3 and the Posix-thread library.

We compare the execution time of Inpla with other evaluators and interpreters. The programs were run on a Linux PC (2.4GHz, Core i7, 16GB) and the execution time was measured using the UNIX `time` command as the average of five executions.

First, in executions of the pure interaction nets, we take INET [1] and amineLight [2] and compare Inpla with those by using programs – Fibonacci function (streaming additive operation) and Ackermann function. Table 1 shows execution time in seconds among interaction nets evaluators. We see that Inpla runs faster than INET since Inpla is a refined version of amineLight, which is the fastest interaction nets evaluator [2]. In the table the subscript of Inpla gives the number of threads in the thread pool, for instance Inpla_2 means that it was executed by using two threads. Generally, since Core i7 processor has four cores, it tends to reach the peak with four execution threads.

Next, we compare Inpla with Standard ML of New Jersey (SML v110.74) [5] and Python (2.7.3) [7] in the extended framework of interaction nets which includes integer numbers and lists. SML is a functional programming language and it has the eager evaluation strategy that is similar to the execution method in interaction nets. Python is a widely-used interpreter, and thus the comparison with Python gives a good indication on efficiency. Here we benchmark the Fibonacci function and the streaming operation versions of Ackermann and the improved version of Bubble Sort algorithm for randomly generated list elements. Table 2 shows that SML computes those arithmetic functions fastest. Inpla uses

agents to represent the functions and integer numbers, and those agents are consumed and reproduced repeatedly during computation. Thus the execution time becomes slower eventually, compared to the execution in SML that performs computation by function calls and managing stacked arguments. In comparison with Python, Inpla computes those functions faster. The sort algorithm is a special case in that interaction nets are efficient to implement these algorithms.

	INET	amLight	Inpla	Inpla_1	Inpla_2	Inpla_3	Inpla_4	Inpla_5
`fib 29`	2.31	2.05	1.29	1.31	1.00	0.93	0.90	0.92
`fib 30`	3.82	3.40	1.74	1.74	1.24	1.13	1.08	1.12
`ack 3 10`	18.26	11.40	4.24	4.42	2.33	1.66	1.36	1.44
`ack 3 11`	66.79	46.30	17.53	18.13	9.47	6.67	5.86	5.83

Table 1. The execution time in seconds on interaction nets evaluators

	SML	Python	Inpla	Inpla_1	Inpla_2	Inpla_3	Inpla_4	Inpla_5
`fib 34`	0.12	2.09	1.67	1.50	0.80	0.70	0.68	0.82
`fib 38`	0.66	16.32	11.39	10.22	5.68	4.47	4.40	4.75
`ack 3 6`	0.03	0.05	0.02	0.03	0.02	0.02	0.02	0.02
`ack 3 9`	0.06	-[1]	0.69	0.72	0.38	0.27	0.24	0.24
`BS 10000`	1.64	6.71	2.11	2.25	1.17	0.87	0.76	0.68
`BS 20000`	8.38	30.35	8.38	8.93	4.57	3.64	2.98	2.49

[1] RuntimeError: maximum recursion depth exceeded

Table 2. The execution time in seconds on interpreters

Next we analyse the results of the parallel execution in Inpla by using graphs in Section 3, which show the trends of steps in parallel execution on the assumption of the unbounded resources. We may write "`parallel`(n)" in the following graphs to make explicit that Inpla_n is used for the experiment.

Fibonacci function. Figure 3 shows the execution time of each program for Fibonacci function by using Inpla. We see that each the sequential execution is exponential as shown in the graphs on the assumption of the unbounded resources. The increase rate of execution time in the parallel execution by Inpla gradually becomes close to, according to increasing the number of threads, the trends of the parallel computation in the graphs on the assumption.

We note that, in the computation of unary natural numbers, the execution of the streaming version is slower than the batch version as shown in the graph on the left side in Figure 4. The graph on the right side shows the ratio of steps in the streaming version to steps in the batch version on the assumption of the

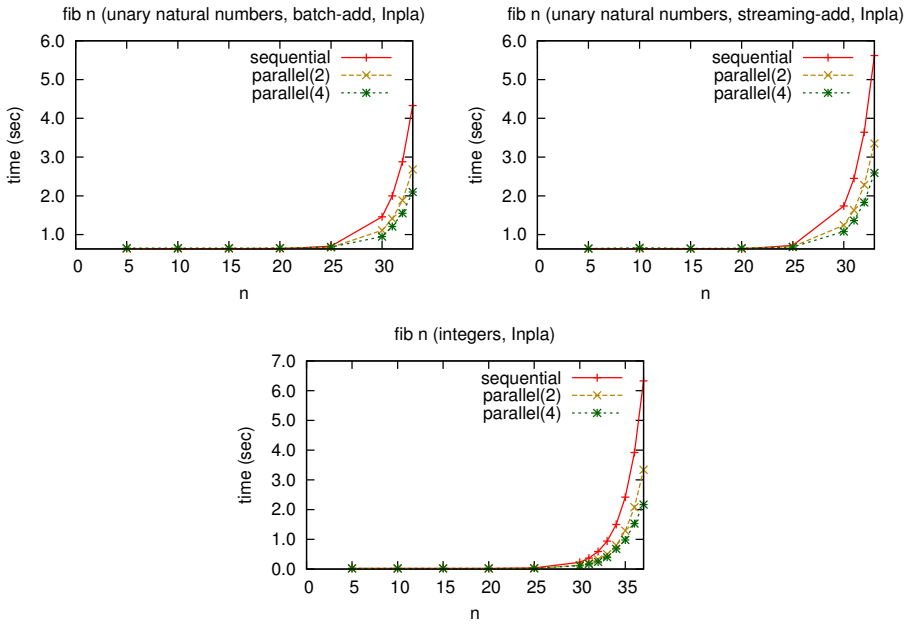


Fig. 3. The execution time of Fibonacci function by Inpla

unbounded resources. The ratio becomes around 0.4 according to increasing n in `ack 3` n. This means that there is a limited benefit of the parallelism, even if we assume unbounded resources. In the real computation, the cost of parallel execution more affects the execution time in comparison to the benefit of the parallelism, and thus the streaming version becomes slower.

Ackermann function. Figure 5 shows the execution time of each program for Ackermann function by using Inpla. We see that, except for the batch operation version, the parallel computation follows well the trends on the assumption of the unbounded resources. On the other hand, the parallel execution of the batch operation version takes quite a long time compared to the streaming version. This is because, in the unbounded resources, not only that there is no significant difference in sequential and parallel execution, but also that there is a cost of parallel execution such as scheduling of threads execution uselessly. These are some of the reasons why the parallel execution does not always have good performance, but are improved in the streaming version.

Bubble sort. Figure 6 shows the execution time of the two programs for Bubble sort using Inpla. As anticipated by the graphs on the assumption of the unbounded resources, we see that the improved version performs best as expected.

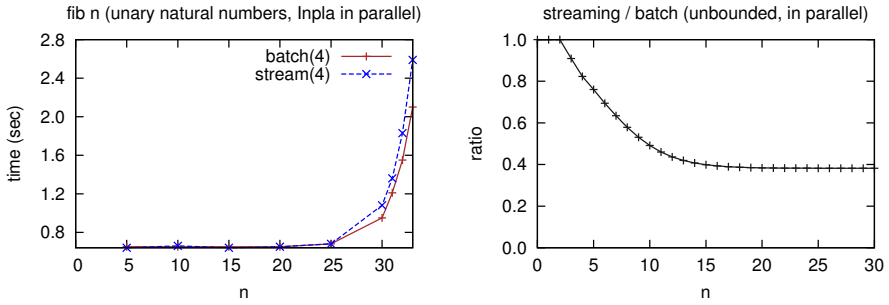


Fig. 4. Comparison between the batch and the streaming addition in parallel execution by Inpla

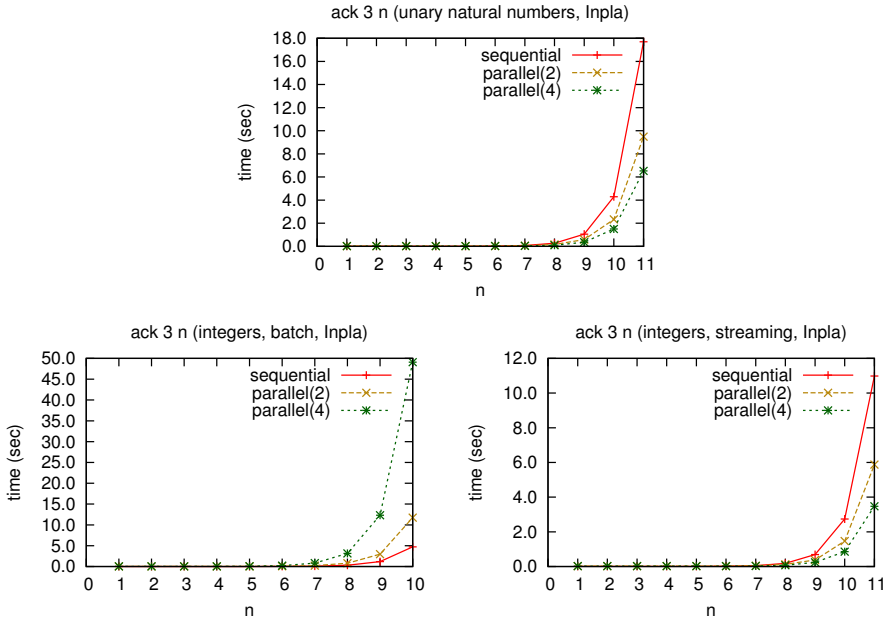


Fig. 5. The execution time of Ackermann function by Inpla

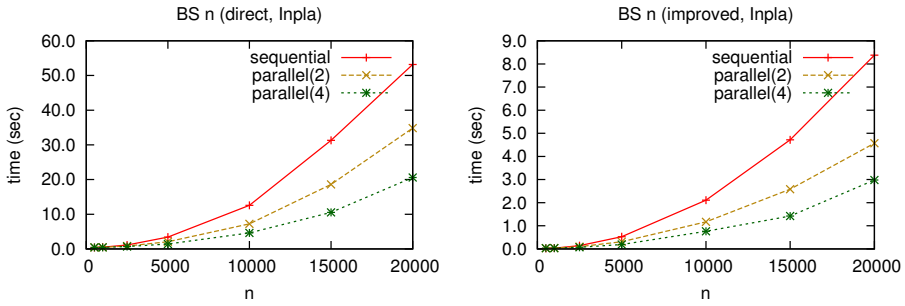


Fig. 6. The execution time of Bubble sort by Inpla

5 Conclusion

Although discussed for many years, we believe that parallel implementations of interaction nets is still a very new area and much needs to be done. In this work we have assumed unbounded resources in terms of the number of processing elements available. This is a reasonable assumption with GPU when many thousands of processing elements are available. We analysed the execution result of the multi-threaded execution by using the investigation result on the assumption, and also showed that, on the one hand, these perform as the best expected, and on the other hand, some of execution results take something away from the investigation results due to an overhead of using parallel technologies as anticipated by the investigation. We hope the ideas in this paper may help in moving this work forward.

References

1. A. Hassan, I. Mackie, and S. Sato. Compilation of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009.
2. A. Hassan, I. Mackie, and S. Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 2010.
3. E. Jiresch. Towards a gpu-based implementation of interaction nets. In B. Löwe and G. Winskel, editors, *DCM*, volume 143 of *EPTCS*, pages 41–53, 2014.
4. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
5. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
6. J. S. Pinto. Sequential and Concurrent Abstract Machines for Interaction Nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 1784 in Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, 2000.
7. G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.