Proofs for Minimality of Sorting Networks by Logic Programming

L. Cruz-Filipe[1] M. Codish[2] M. Frank[2]
P. Schneider-Kamp[1]

[1]Dept. Mathematics and Computer Science, Univ. Southern Denmark (Denmark)

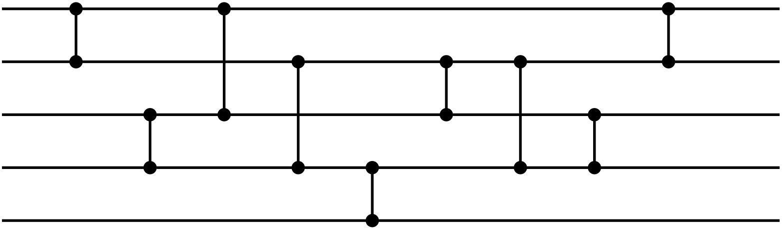
[2]Ben-Gurion University of the Negev (Israel)

CICLOPS-WLPE Workshop
July 17th, 2014

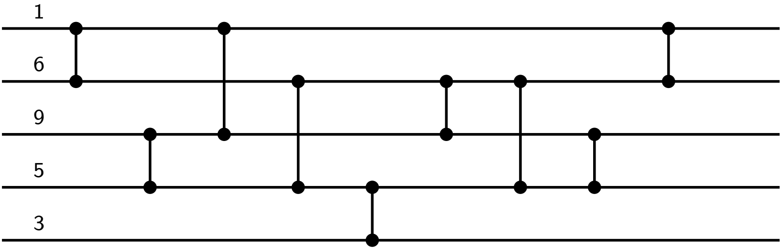
Outline

Outline

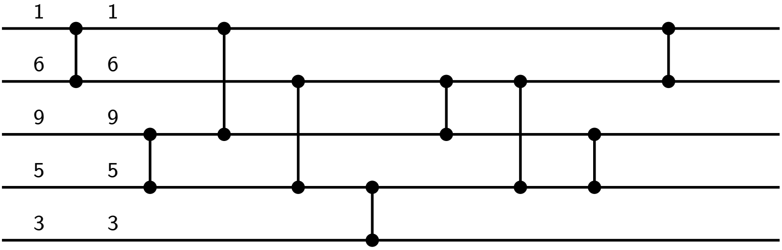
A sorting network



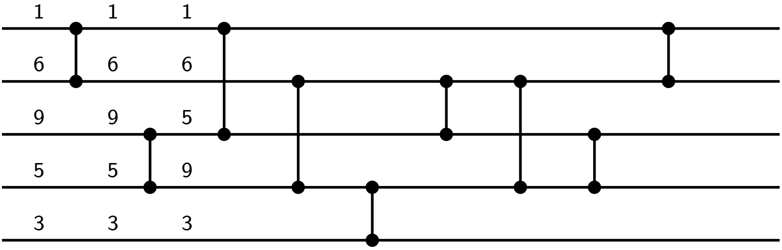
A sorting network



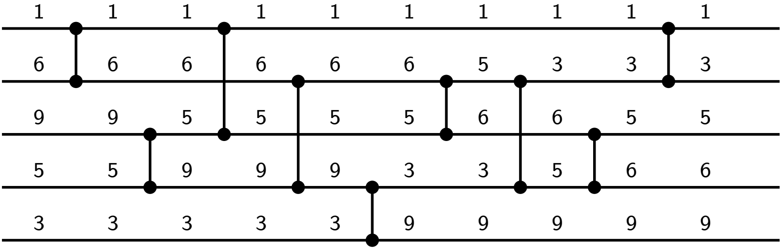
A sorting network



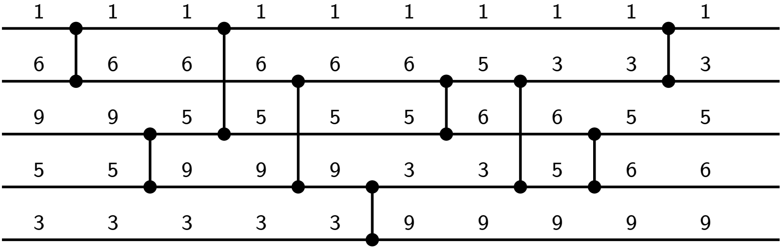
A sorting network



A sorting network



A sorting network



Size

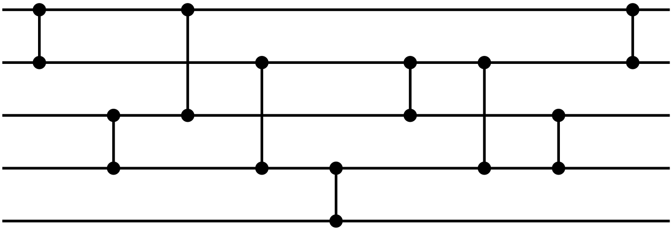
This net has 5 *channels* and 9 *comparators*.

A sorting network

Some of the comparisons may be performed in parallel:

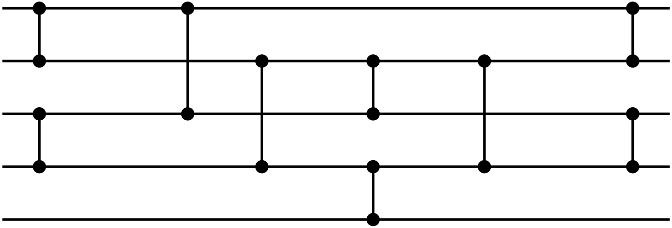
A sorting network

Some of the comparisons may be performed in parallel:



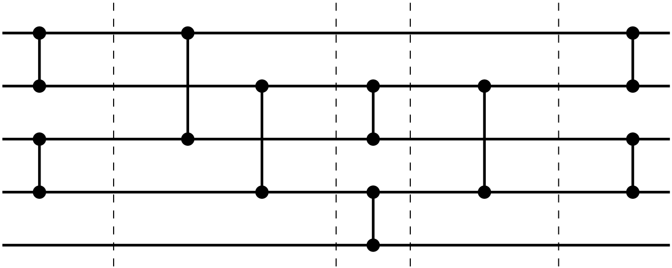
A sorting network

Some of the comparisons may be performed in parallel:



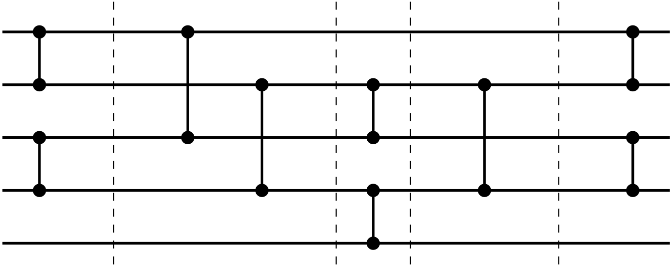
A sorting network

Some of the comparisons may be performed in parallel:



A sorting network

Some of the comparisons may be performed in parallel:

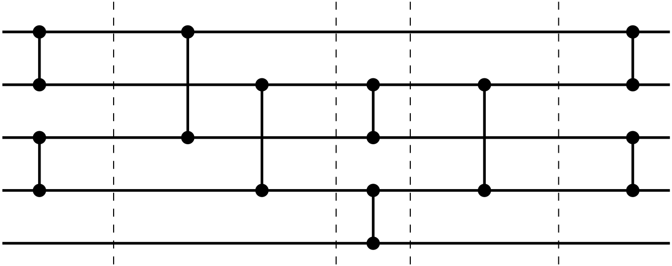


Depth

This net has 5 *layers*.

A sorting network

Some of the comparisons may be performed in parallel:



Depth

This net has 5 *layers*.

See Donald E. Knuth, *The Art of Computer Programming*, vol. 3 for more details

The optimization problems

The size problem

What is the minimal number of *comparators* on a sorting network on n channels (S_n)?

The depth problem

What is the minimal number of *layers* on a sorting network on n channels (T_n)?

The optimization problems

The size problem

What is the minimal number of *comparators* on a sorting network on n channels (S_n)?

The depth problem

What is the minimal number of *layers* on a sorting network on n channels (T_n)?

Knuth 1973

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	3	5	9	12	16	19	25 23	29 27	35 31	39 35	45 39	51 43	56 47	60 51
T_n	3	3	5	5	6	6	7 6	7 6	8 6	8 6	9 6	9 6	9 6	9 6

The optimization problems

The size problem

What is the minimal number of *comparators* on a sorting network on n channels (S_n)?

The depth problem

What is the minimal number of *layers* on a sorting network on n channels (T_n)?

Parberry 1991

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	3	5	9	12	16	19	25 23	29 27	35 31	39 35	45 39	51 43	56 47	60 51
T_n	3	3	5	5	6	6	**7**	**7**	8 **7**	8 **7**	9 **7**	9 **7**	9 **7**	9 **7**

The optimization problems

The size problem

What is the minimal number of *comparators* on a sorting network on n channels (S_n)?

The depth problem

What is the minimal number of *layers* on a sorting network on n channels (T_n)?

Bundala & Závodný 2013

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	3	5	9	12	16	19	25	29	35	39	45	51	56	60
							23	27	31	35	39	43	47	51
T_n	3	3	5	5	6	6	7	7	**8**	**8**	**9**	**9**	**9**	**9**

The optimization problems

The size problem

What is the minimal number of *comparators* on a sorting network on n channels (S_n)?

The depth problem

What is the minimal number of *layers* on a sorting network on n channels (T_n)?

Codish, Cruz-Filipe, Frank & Schneider-Kamp (CCFS) 2014

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S_n	3	5	9	12	16	19	**25**	**29**	35 **33**	39 **37**	45 **41**	51 **45**	56 **49**	60 **53**
T_n	3	3	5	5	6	6	7	7	8	8	9	9	9	9

An exponential explosion

- Parberry (1991)
  - exploration of symmetries
  - fixed first layer
  - 200 hours of computation

An exponential explosion

- Parberry (1991)
- Bundala & Závodný (2013)
  - exploration of symmetries
  - reduced set of two-layer prefixes
  - intensive SAT-solving

An exponential explosion

- Parberry (1991)
- Bundala & Závodný (2013)
- Techniques not directly applicable to the size problem

36 possibilities for each comparator when $n = 9$, so $36^{24} \approx 2.2 \times 10^{37}$ 24-comparator nets

2620 possibilities for each layer when $n = 9$, so $2620^{6} \approx 3.2 \times 10^{20}$ 6-layer networks

An exponential explosion

- Parberry (1991)
- Bundala & Závodný (2013)
- Techniques not directly applicable to the size problem
- CCFS (2014)
  - generate-and-prune
  - combine brute-force generation with optimal (?) reduction
  - compromise between time and space

Outline

Comparator networks

- A *(standard) comparator network* C on n channels is a sequence of pairs (i, j) (the comparators) such that $1 \leq i < j \leq n$.
- The *output* of C on a sequence $\vec{x}$ is denoted $C(\vec{x})$.
- The set of outputs of C is $\text{outputs}(C) = \{C(\vec{x}) \mid \vec{x} \in \{0,1\}^n\}$.
- A comparator network C is a *sorting network* if all elements of $\text{outputs}(C)$ are sorted.

Well-known results

0–1 lemma (Knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$.

Well-known results

0–1 lemma (Knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$.

"C is a sorting network on n channels" is co-NP (complete).

Well-known results

0–1 lemma (Knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$.

Output lemma (Parberry 1991)

Let C and C' be comparator networks such that outputs$(C) \subseteq$ outputs(C'). If $C'; N$ is a sorting network, then so is $C; N$.

Well-known results

0–1 lemma (Knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$.

Output lemma (Parberry 1991)

Let C and C' be comparator networks such that $\mathrm{outputs}(C) \subseteq \mathrm{outputs}(C')$. If $C'; N$ is a sorting network, then so is $C; N$.

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{C} & X & & \\ & & \cap\!\!\mid & & \\ \{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S \end{array}$$

Well-known results

0–1 lemma (Knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$.

Output lemma (Parberry 1991)

Let C and C' be comparator networks such that $\text{outputs}(C) \subseteq \text{outputs}(C')$. If $C'; N$ is a sorting network, then so is $C; N$.

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{C} & X & \xrightarrow{N} & S \\ & & \rotatebox{90}{$\supseteq$} & & \\ \{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S \end{array}$$

Permutations (Bundala & Závodný 2013)

Permuted output lemma (I)

If:

- C and C' are standard comparator networks of depth 2;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same depth.

Permutations (Bundala & Závodný 2013)

Permuted output lemma (I)

If:

- C and C' are standard comparator networks of depth 2;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same depth.

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{C} & X & & \\ & & \downarrow \pi & & \\ \{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S \end{array}$$

Permutations (Bundala & Závodný 2013)

Permuted output lemma (I)

If:

- C and C' are standard comparator networks of depth 2;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same depth.

$$
\begin{array}{ccccc}
\{0,1\}^n & \xrightarrow{C} & X & \xrightarrow{\pi^{-1}(N)} & \pi^{-1}(S) \\
 & & \downarrow \pi & & \\
\{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S
\end{array}
$$

Permutations (Bundala & Závodný 2013)

Permuted output lemma (I)

If:

- C and C' are standard comparator networks **of depth** 2;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same **depth**.

$$\begin{array}{ccccc}
\{0,1\}^n & \xrightarrow{C} & X & \xrightarrow{\pi^{-1}(N)} & \pi^{-1}(S) \\
 & & \downarrow \pi & & \\
\{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S
\end{array}$$

Permutations revisited (CCFS 2014)

Permuted output lemma (II)

If:

- C and C' are standard comparator networks **of equal size**;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same **size**.

Permutations revisited (CCFS 2014)

Permuted output lemma (II)

If:

- C and C' are standard comparator networks **of equal size**;
- π is a permutation of $1..n$ mapping outputs(C) into outputs(C');
- C' can be extended to a sorting network;

then C can also be extended to a standard sorting network of the same **size**.

We say that $C \preceq C'$ when $\pi(\text{outputs}(C)) \subseteq \text{outputs}(C')$ for some permutation π.

The algorithms (I)

Generate-and-prune

1. (Init) Set $R_0^n = \{\emptyset\}$ and $k = 0$.
2. Repeat:
   - (Generate) Extend every net in R_k^n with one comparator in every possible way. Let N_{k+1}^n be the set of all results.
   - (Prune) Keep only one element of each minimal equivalence class w.r.t. the transitive closure of $\preceq$. Let R_{k+1}^n be the resulting set.
   - Increase k.

   until $k > 1$ and $|R_k^n| = 1$.

(If C is a sorting network on n channels of size k, then $|R_k^n| = 1$.)

The algorithms (II)

Generate (Input R_k^n; output N_{k+1}^n)

- (Init) $N_{k+1}^n = \emptyset$, $C_n = \{(i,j) \mid 1 \leq i < j \leq n\}$
- for $C \in R_k^n$ and $c \in C_n$: $N_{k+1}^n = N_{k+1}^n \cup \{C;c\}$

The algorithms (II)

Generate (Input R_k^n; output N_{k+1}^n)

- (Init) $N_{k+1}^n = \emptyset$, $C_n = \{(i,j) \mid 1 \leq i < j \leq n\}$
- for $C \in R_k^n$ and $c \in C_n$: $N_{k+1}^n = N_{k+1}^n \cup \{C; c\}$

Prune (Input N_k^n; output R_k^n)

- (Init) $R_k^n = \emptyset$
- for $C \in N_k^n$ do
  - for $C' \in R_k^n$: if $(C' \preceq C)$ then mark C
  - if (not_marked(C)) then
    - for $C' \in R_k^n$: if $(C \preceq C')$ then $R_k^n = R_k^n \setminus \{C'\}$
    - $R_k^n = R_k^n \cup \{C\}$

Optimizing Generate

Redundant comparators

A comparator (i, j) is *redundant* w.r.t. C if $x_i \leq x_j$ for every $\vec{x} \in \text{outputs}(C)$.

Optimizing Generate

Redundant comparators

A comparator (i, j) is *redundant* w.r.t. C if $x_i \leq x_j$ for every $\vec{x} \in \text{outputs}(C)$.

Redundant comparators:

- do nothing;
- may not occur in minimal-size sorting networks;
- are easy to detect;
- can be avoided at generation time.

Optimizing `Generate`

Redundant comparators

A comparator (i, j) is *redundant* w.r.t. C if $x_i \leq x_j$ for every $\vec{x} \in \text{outputs}(C)$.

Redundant comparators:

- do nothing;
- may not occur in minimal-size sorting networks;
- are easy to detect;
- can be avoided at generation time.

`Generate` is much faster than `Prune`, so it pays off to do this test at generation time.

Optimizing `Prune`

The big cost in `Prune` is searching for a candidate permutation in the subsumption test.

Optimizing Prune

The big cost in Prune is searching for a candidate permutation in the subsumption test.

outputs(C_a)					outputs(C_b)				
0000	0001 0010	0011 1100	0111	1111	0000	0001	0011 0101	0111	1111

Optimizing Prune

The big cost in Prune is searching for a candidate permutation in the subsumption test.

outputs(C_a)					outputs(C_b)				
0000	0001 0010	0011 1100	0111	1111	0000	0001	0011 0101	0111	1111

A cardinality test shows that no permutation can map $\{0001, 0010\}$ into $\{0001\}$, so $C_a \not\preceq C_b$. Such a test eliminates 70% of unsuccessful subsumptions.

Optimizing Prune

The big cost in Prune is searching for a candidate permutation in the subsumption test.

outputs(C_a)					outputs(C_b)				
0000	0001 0010	0011 1100	0111	1111	0000	0001	0011 0101	0111	1111

A cardinality test shows that no permutation can map $\{0001, 0010\}$ into $\{0001\}$, so $C_a \not\preceq C_b$. Such a test eliminates 70% of unsuccessful subsumptions.

Analysis of positions containing '1' shows that no permutation can map $\{0011, 1100\}$ into $\{0011, 0101\}$, so again $C_a \not\preceq C_b$. Such a test eliminates 30% of the remaining unsuccessful subsumptions.

Optimizing Prune

The big cost in Prune is searching for a candidate permutation in the subsumption test.

outputs(C_a)					outputs(C_b)				
0000	0001	0011	0111	1111	0000	0001	0011	0111	1111
	0010	1100					0101		

A cardinality test shows that no permutation can map $\{0001, 0010\}$ into $\{0001\}$, so $C_a \not\preceq C_b$. Such a test eliminates 70% of unsuccessful subsumptions.

Analysis of positions containing '1' shows that no permutation can map $\{0011, 1100\}$ into $\{0011, 0101\}$, so again $C_a \not\preceq C_b$. Such a test eliminates 30% of the remaining unsuccessful subsumptions.

Also, this position analysis significantly restricts the search space of possible permutations.

Network representation

For efficiency, we store comparator networks with their sets of outputs:

- each output is represented as an integer
- outputs are partitioned according to the number of 1s
- each partition is annotated with its "where" sets

Network representation

For efficiency, we store comparator networks with their sets of outputs:

- each output is represented as an integer
- outputs are partitioned according to the number of 1s
- each partition is annotated with its "where" sets

outputs(C_a)	
0000	
0001	0010
0011	1100
0111	
1111	

$$
\begin{aligned}
\langle C_a, \langle & \langle\{0\}, \{1,2,3,4\}, \emptyset\}\rangle, \\
& \langle\{8,4\}, \{1,2,3,4\}, \{3,4\}\}\rangle, \\
& \langle\{12,3\}, \{1,2,3,4\}, \{1,2,3,4\}\}\rangle, \\
& \langle\{14\}, \{1\}, \{2,3,4\}\}\rangle, \\
& \langle\{15\}, \emptyset, \{1,2,3,4\}\}\rangle\rangle\rangle
\end{aligned}
$$

Network representation

For efficiency, we store comparator networks with their sets of outputs:

- each output is represented as an integer
- outputs are partitioned according to the number of 1s
- each partition is annotated with its "where" sets

outputs(C_a)	
0000	
0001	0010
0011	1100
0111	
1111	

$$
\begin{aligned}
\langle C_a, \langle & \langle\{0\}, \{1,2,3,4\}, \emptyset\}\rangle, \\
& \langle\{8,4\}, \{1,2,3,4\}, \{3,4\}\}\rangle, \\
& \langle\{12,3\}, \{1,2,3,4\}, \{1,2,3,4\}\}\rangle, \\
& \langle\{14\}, \{1\}, \{2,3,4\}\}\rangle, \\
& \langle\{15\}, \emptyset, \{1,2,3,4\}\}\rangle\rangle\rangle
\end{aligned}
$$

This data is computed at generation time, so that it will be readily available every time it is needed for a subsumption test.

What do we need to trust?

What do we need to trust?

```
%% iterates over partitioned set of outputs
carriesInto(_,[],_).
carriesInto(P,[t(P1,_,_)|Part1],[t(P2,_,_)|Part2]) :-
        mapsInto(P,P1,P2),
        carriesInto(P,Part1,Part2).

%% iterates over outputs
mapsInto(_,[],_).
mapsInto(P,[X|P1],P2) :- permuted(P,X,Y), member(Y,P2),
                         mapsInto(P,P1,P2).

%% applies permutation
permuted(P,N,M) :- permuted(P,0,N,0,M).

permuted([],_,_,M,M).
permuted([_|P],I,N,K,M) :- position(N,I,0), !,
                           I1 is I+1, permuted(P,I1,N,K,M).
permuted([J|P],I,N,K,M) :- K1 is K+2**J, I1 is I+1,
                           permuted(P,I1,N,K1,M).
```

Some numerology

R_k^n	3	4	5	6	7	8
1	1	1	1	1	1	1
2	2	3	3	3	3	3
3	**1**	4	6	7	7	7
4		2	11	17	19	20
5		**1**	10	36	51	57
6			7	53	141	189
7			6	53	325	648
8			4	44	564	2,088
9			**1**	23	678	5,703
10				8	510	11,669
11				4	280	16,095
12				**1**	106	13,305
13					33	6,675
14					11	2,216
15					6	503
16					**1**	77
17						18
18						9
19						**1**

Outline

1. Sorting Networks in a Nutshell
2. The Generate-and-Prune Approach
3. Parallelization
4. Conclusions & Future Work

Parallelization (I)

With all these optimizations in place, the known values for S_n ($n \leq 8$) could be checked in under one day.

- $n = 6$: two seconds
- $n = 7$: two minutes
- $n = 8$: several hours

Parallelization (I)

With all these optimizations in place, the known values for S_n ($n \leq 8$) could be checked in under one day.

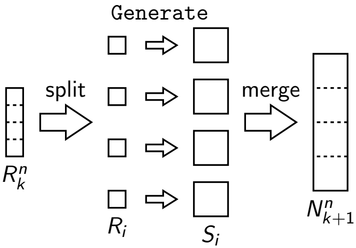
- $n = 6$: two seconds
- $n = 7$: two minutes
- $n = 8$: several hours

A rough estimate of the computation time for $n = 9$ yielded 10–20 years. With a 288-thread cluster available, the precise computation of S_9 became feasible for the first time.

Parallelization (II)

Parallel-Generate
(Input R_k^n; output N_{k+1}^n)

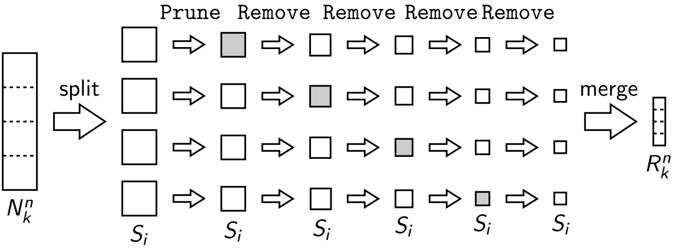
- split R_k^n into sets $R_1, \ldots, R_p$
- $\texttt{for}||^p\ i \in \{1, \ldots, p\}$ do
  - $S_i = \texttt{Generate}(R_i)$
- $N_{k+1}^n = \biguplus_{1 \leq i \leq p} S_i$;



Parallelization (III)

Parallel-Prune (Input N_k^n; output R_k^n)

- split N_k^n into sets $S_1, \ldots, S_p$
- $\texttt{for}||^p\ i \in \{1, \ldots, p\}$: $S_i = \texttt{Prune}(S_i)$
- $\texttt{for}\ j \in \{1, \ldots, p\}$ do
  - $\texttt{for}||^{p-1}\ i \neq j$: $S_i = \texttt{Remove}(S_i, S_j)$
- $R_k^n = \biguplus_{1 \leq i \leq p} S_i$;



Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions – empty files for synchronization

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions – empty files for synchronization
  - distributed memory

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions – empty files for synchronization
  - distributed memory – read and write from shared file system

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions – empty files for synchronization
  - distributed memory – read and write from shared file system
  - limited disk space

Master-Slave Parallelization - The Hard Way

- for reasonable number of threads p
  - parallel runtime dominated by computation
  - overhead of master-slave parallelization tolerable
- robustness over sophistication
  - round-robin strategy to distribute goals
  - busy waiting (aided by some sleeps)
  - goals distributed through shared file system
  - no instantiation of variables
  - all predicates must succeed
- practical challenges
  - race conditions – empty files for synchronization
  - distributed memory – read and write from shared file system
  - limited disk space – use `zlib` for transparent (de-)compression

Master-Slave Parallelization - The Slave

```
%% launch multiple clients with given thread id range
slave(FirstThread, LastThread) :-
    findall(client(I),between(FirstThread,LastThread,I),Goals),
    NumThreads is LastThread - FirstThread + 1,
    concurrent(NumThreads, Goals, []).

%% client with thread id I
client(I) :-
    goal_files(I, GI, RI), exists_file(RI), !,
    see(GI), read(Goal), seen,
    (Goal = halt -> true; Goal),
    delete_file(RI), delete_file(GI),
    (Goal = halt -> true ; client(I)).

client(I) :- sleep(1), client(I).

%% helper
goal_files(I, GI, RI) :-
    name('goal', G), name('.', D), name(I,II), name('.ready', R),
    append([G, D, II], GIName),
    append(GIName, R, RIName),
    name(A,AName), name(B, BName).
```

Master-Slave Parallelization - The Master

```
%% distributed simplified variant of SWI-Prolog's concurrent/3
parallel(Procs, Goals) :- distribute(1, Procs, Goals).

distribute(_, Procs, []) :- !, wait(Procs).

distribute(I, Procs, Goals) :-
    goal_file_exists(I), !,
    I1 is I mod Procs + 1, distribute(I1, Procs, Goals).

distribute(I, Procs, [Goal | Goals]) :-
    goal_files(I, GI, RI),
    tell(GI), write_goal(Goal), told, tell(RI), told,
    distribute(I, Procs, Goals).

wait(0) :- !.
wait(I) :- goal_file_exists(I), !, sleep(1), wait(I).
wait(I) :- I1 is I-1, wait(I1).

%% helpers
write_goal(G) : - write_term(G, [quoted(true)]), writeln('.'),

goal_file_exists(I) :- goal_files(I, GI, _), exists_file(GI).
```

Wish List

- better support for distributed computing
  - Erlang / Akka style?
  - MPI support?

Wish List

- better support for distributed computing
  - Erlang / Akka style?
  - MPI support?
- better supoort for memory-intensive computations
  - file-backed data structures
  - distributed memory data structures

Wish List

- better support for distributed computing
  - Erlang / Akka style?
  - MPI support?
- better supoort for memory-intensive computations
  - file-backed data structures
  - distributed memory data structures
- more transparent compression suport
  - built-in `zlib`-equivalents of `see/1`, `seen/0`, `tell/1`, `told/0`

Independent Java Verifier

- independent implementation of generate-and-prune
  - stupidly generate all comparator networks by nested for-loops
  - instead of search, use log file for pruning
  - check subsumptions in log before use
  - ensure acyclicity of reasoning
  - 205 lines of Java code

Independent Java Verifier

- independent implementation of generate-and-prune
  - stupidly generate all comparator networks by nested for-loops
  - instead of search, use log file for pruning
  - check subsumptions in log before use
  - ensure acyclicity of reasoning
  - 205 lines of Java code
- all 123,599,036 subsumptions for $n = 9$ verified
- generate-and-prune without search for $n = 9$ in 6 hours

Independent Java Verifier

- independent implementation of generate-and-prune
  - stupidly generate all comparator networks by nested for-loops
  - instead of search, use log file for pruning
  - check subsumptions in log before use
  - ensure acyclicity of reasoning
  - 205 lines of Java code
- all 123,599,036 subsumptions for $n = 9$ verified
- generate-and-prune without search for $n = 9$ in 6 hours
- verifier & logs available at:
  `http://imada.sdu.dk/~petersk/sn/`

Outline

Results & Future work

- Exact values of S_9 and S_{10}
- Log-file that can be independently verified

- Technique may be adapted to settle higher values which are still unknown
- Algorithms may be useful for *finding* smaller-than-currently-known networks
- Further theoretical results may help proving optimality of best known upper bounds

Thank you!