Virtual Machine and Integrated Developer Environment for Sleptsov Net Computing

Dmitry A Zaitsev, Tatiana R Shmeleva, Qing Zhang, Hongfei Zhao

▶ **To cite this version:**

Dmitry A Zaitsev, Tatiana R Shmeleva, Qing Zhang, Hongfei Zhao. Virtual Machine and Integrated Developer Environment for Sleptsov Net Computing. Parallel Processing Letters, 2023, 33 (03), 10.1142/s0129626423500068 . hal-04121781

HAL Id: hal-04121781
https://hal.science/hal-04121781

Submitted on 8 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Processing Letters
© World Scientific Publishing Company

Virtual Machine and Integrated Developer Environment for Sleptsov Net Computing *

Dmitry A. Zaitsev
Tatiana R. Shmeleva
I3S CNRS, Université Côte d'Azur, 28 Avenue de Valrose, Nice, 06103, France

Qing Zhang
Hongfei Zhao
XIDIAN University, 266 Xinglong Section of Xifeng Road, Xi'an, Shaanxi, 710126, China

Received (received date)
Revised (revised date)

ABSTRACT

Modern computing is a path of violations and transformations coming from an intrinsically concurrent application domain into a sequence of instructions and then back to concurrency with OpenMP, MPI and CUDA/OpenCL. Why we create so many difficulties? Sleptsov Net Computing (SNC) maps a task into an appropriate computing structure implemented as a re-configurable multidimensional sparse matrix of computing memory. It has entirely graphical mass parallel language for concurrent programming and a framework of techniques for concurrent program verification to develop reliable software. Estimated efficiency of SNC is higher than 50% compared to actual less that 1% efficiency of the most powerful supercomputers. It yields hyper-performance capable of efficient control of hyper-sonic objects, colliders, thermonuclear reaction. This paper presents an open source prototype VM and IDE for SNC with a view on upcoming hardware implementation of the corresponding computer.

Keywords: Sleptsov net computing; concurrent programming; efficient computing; computing memory; virtual machine; integrated developer environment.

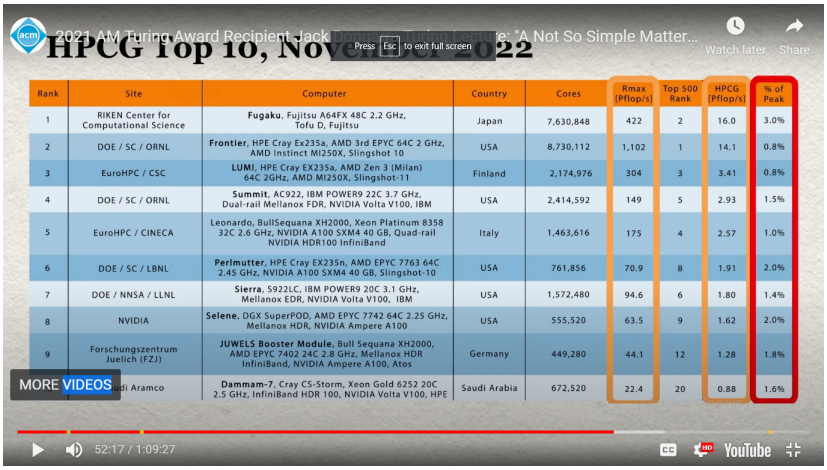
1. Introduction

ACM Tech News [1] digest of editorial paper [2], following recent update [3] on Sleptsov net computing (SNC) [4], presents SNC as an approach that resolves modern supercomputing problems [5].

*Partially supported by Fulbright Program, USA in 2017, Université Côte d'Azur, France in 2022, and Programme PAUSE, Collège de France in 2022-2023.

Jack Dongarra, in his ACM A.M. Turing Award lecture [5] revealed rather shocking information. For a few decades LINPACK [6, 7] – one of the first of his well-known packages, – has been applied for benchmarks of supercomputers, reflected on Top500 [8] web-site, that influenced world trends of computer hardware design and strategic plans of leading international companies which produce supercomputers. LINPACK benchmark represent solving a dense system of linear equations.

During the final part of his lecture, Jack Dongarra reveals results of recent tests on real-life applications, concisely expressed in a novel High Performance Conjugate Gradients (HPCG) Benchmark [9], which uses a mixture of tasks over sparse data. Corrected with HPGC column, table of the top ten most powerful supercomputers in the world (Fig. 1) looks rather drastically. Actual performance of the most powerful in the world USA computer Frontier is 0.8% of its performance measured with LINPACK that positions it on the second place after Japanese computer Fugaku [10] that is the leader with 3.0%.



Rank	Site	Computer	Country	Cores	Rmax [Pflop/s]	Top 500 Rank	HPCG [Pflop/s]	% of Peak
1	RIKEN Center for Computational Science	**Fugaku**, Fujitsu A64FX 48C 2.2 GHz, Tofu D, Fujitsu	Japan	7,630,848	422	2	16.0	3.0%
2	DOE / SC / ORNL	**Frontier**, HPE Cray Ex235a, AMD 3rd EPYC 64C 2 GHz, AMD Instinct MI250X, Slingshot 10	USA	8,730,112	1,102	1	14.1	0.8%
3	EuroHPC / CSC	**LUMI**, HPE Cray EX235a, AMD Zen 3 (Milan) 64C 2GHz, AMD MI250X, Slingshot-11	Finland	2,174,976	304	3	3.41	0.8%
4	DOE / SC / ORNL	**Summit**, AC922, IBM POWER9 22C 3.7 GHz, Dual-rail Mellanox FDR, NVIDIA Volta V100, IBM	USA	2,414,592	149	5	2.93	1.5%
5	EuroHPC / CINECA	Leonardo, BullSequana XH2000, Xeon Platinum 8358 32C 2.6 GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 InfiniBand	Italy	1,463,616	175	4	2.57	1.0%
6	DOE / SC / LBNL	**Perlmutter**, HPE Cray EX235n, AMD EPYC 7763 64C 2.45 GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	USA	761,856	70.9	8	1.91	2.0%
7	DOE / NNSA / LLNL	**Sierra**, S922LC, IBM POWER9 20C 3.1 GHz, Mellanox EDR, NVIDIA Volta V100, IBM	USA	1,572,480	94.6	6	1.80	1.4%
8	NVIDIA	**Selene**, DGX SuperPOD, AMD EPYC 7742 64C 2.25 GHz, Mellanox HDR, NVIDIA Ampere A100	USA	555,520	63.5	9	1.62	2.0%
9	Forschungszentrum Juelich (FZJ)	**JUWELS Booster Module**, Bull Sequana XH2000, AMD EPYC 7402 24C 2.8 GHz, Mellanox HDR InfiniBand, NVIDIA Ampere A100, Atos	Germany	449,280	44.1	12	1.28	1.8%
10	di Aramco	**Dammam-7**, Cray CS-Storm, Xeon Gold 6252 20C 2.5 GHz, InfiniBand HDR 100, NVIDIA Volta V100, HPE	Saudi Arabia	672,520	22.4	20	0.88	1.6%

Fig. 1. Jack Dongarra top10 table supplied with HPCG results

Fugaku leadership is explained by using Tofu D Interconnect representing 6D torus with big quantity of alternative paths, having short length, between nodes. Indeed, real-life mixture of tasks has rather irregular pattern of intensive communications that amplifies negative influence of well-known processor-memory bottleneck. It is traditionally mended by multilevel cash which is getting well filled in with repeatedly used data on dense matrix operations for LINPACK test.

Discussing advantages of certain computers architecture on HPCG test, we deviate from the primary conclusion induced by the fact that even for the best computer Fugaku, it is about thirty times less than its peak performance specified, that means novel paradigms of concurrent computing are in utmost demand.

Petri nets with multichannel transitions, introduced [11, 12] for practical man-

ufacture control and management [13] at Topaz and Motorsich plants of Ukraine, represent a concept, later on, successfully applied in theory of computations to speed-up spiking neuron systems [14] and DNA computing [15], and called an exhaustive use of rule. Multiply firing of a transition at a step (in a certain number of copies) resulted in exponential speed-up when composing small universal Petri nets [16] that was basic motivation to call the corresponding class of place-transition nets a Sleptsov net (SN) [4, 17, 18] after an outstanding scientist Anatoly Sleptsov from Ukraine who hinted the idea.

Indeed, for many years place-transition nets were applied for concurrent programming loaded with constructs of an algorithmic language. Besides, place-transition notation was adopted in UML [19] for activity diagrams. Though Petri nets are slow, especially when computing arithmetic functions because of their incremental character of computations similar to counter automata.

Sleptsov net computing resolves modern supercomputing problems [1, 2]. A Sleptsov net yields fast computations that opens prospects of its application as a uniform graphical language of concurrent programming. It means computations are completely specified by Sleptsov net graphical program, textual constructs are applied as comments only. Implementation of Sleptsov net machine in the form of computing memory removes processor-memory bottleneck of traditional architecture resulting in hyper-performance. It allows us to start with fine granulation of concurrent processes at the level of separate operations. It implies a novel style of concurrent programming, when drawing a program, we do not introduce sequential dependencies which are not intrinsic to the application domain. Moreover, for reliable software design, a framework of early developed formal techniques for concurrent programs verification is available.

The present paper describes an experimental implementation of Sleptsov net computing (SNC) in the form of virtual machine (VM) and integrated developer environment (IDE) based on graphical editor of place-transition nets of system Tina [20] and dedicated ad-hoc tool-set developed in University Cote d'Azur (France) and XIDIAN University (China) as an open-source software available on GitHub [21, 22, 23, 24]; the paper is accomplished with analysis of benchmarks based on tasks of LINPACK and HPCG tests. Hardware implementation of SNP promises break-though in efficiency of modern HPC.

2. An overview of Sleptsov net computing paradigm

Sleptsov net computing paradigm [3, 4, 17, 25] is based on completely graphical language of concurrent programming, with fine granulation of concurrent processes, and computing memory implementation in the form of a big, adjustable to the task structure, sparse matrix. For solving tasks with a certain spatial structure, multidimensional matrices of computing elements are applied.

The foremost peculiarity of a Sleptsov net is firing a transition at a step in a certain number of instances (multiple firing), while a Petri net fires a single transi-

tion in a single instance, and Salwicki net fires the maximal set of firable transitions in a single instance each [3]. That is why an SN runs exponentially faster than a Petri net [4] opening prospects of using it as a uniform language for concurrent programming. Inherently, an SN transition implements division-subtraction on each its incoming arc and multiplication-addition on each its outgoing arc that makes SN rather powerful computer as well. An example of SN adding two numbers in three time cycles is shown in Fig. 2, its state space represented in Fig. 3.

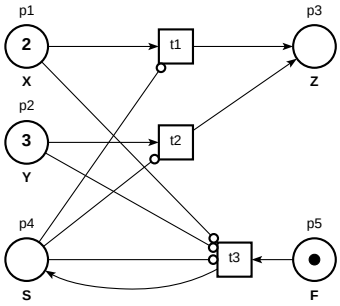


Fig. 2. An LSN for computing z=x+y

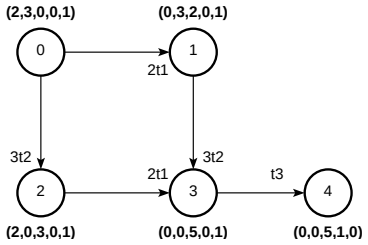


Fig. 3. Behavior of LSN shown in Fig.2

SNC allows us using data flow or control flow approaches or both in mixed control programs. The basic motivation, when drawing a program, is to not restrict intrinsic parallelism of an application domain starting from the level of separate arithmetic and logic operations implementation. Usual textual program enforces application of sequential constructs while, when drawing a program, it is easier to preserve concurrency, especially with a Sleptsov net where each transition is controlled locally. All the actions are primordially concurrent and the net just ensures certain limitations on this mass parallelism.

For convenience of drawing programs, we supply an SN with inhibitor and priority arcs though, according to the recent results [3], a strong Sleptsov net is Turing-complete and we can use conventional arcs only. As distinct from traditional technique of programming in place-transition nets, where a control flow is represented by passage of a control token through the control flow subnet [26, 27], in SN programs, we use a reversed control flow represented by passage of an empty marking (a "hole") through the control flow subnet. Because of it, we use inhibitor arcs, which possess a symbolic infinite firing multiplicity at zero marking, to control firing of data processing transitions. Thus, firing multiplicity of control does not restrict multiplicity ensured by data. For each traditional control flow pattern such as branching, loop, parallel execution, we offer corresponding SN control-flow patterns to make programming in SNs easier at the beginning.

To avoid using textual programming languages, we start from a set of SNs which implement basic arithmetic, logic, and comparison operations. For hierarchical de-

sign of graphical programs, we use substitution of a transition by a subnet. SNs with transition substitution are called high-level SNs. They are compiled and linked into a single plain low-lewel SNs without substitution of transition which are considered as an analog og machine language for SNC paradigm.

Though a place of SN stores a nonnegative integer number, we can represent integer, rational, and floating numbers by a cortege of places, subnets for corresponding operations have been presented in [28]. As for the arrays and matrices, multidimensional as well, the most efficient implementation uses explicitly generated matrix computing structures, similar to shown in an example for solving Laplace equation in a single step for each iteration. Based on this technique, deep learning implementation on SNs has been presented [29]. For variable-length structures encoding, we can apply technique of encoding arrays and matrices described in [18].

As for SN machine, we offer its implementation in the form of computing memory. Its main function consists in fast firing transitions. Combining Sleptsov strategy of maximal firing multiplicity with Salwicki strategy of maximal parallelism, we fire a maximal multiset [3] of transitions at a step. It results in mass parallel computations with fine granulation (addition, subtraction, multiplication, division within a transition) in case a Sleptsov net is perfectly mapped into the corresponding hardware structure of computing memory. Computing memory implementation results in a nanosecond time cycle that, combined with mass parallelism, opens prospects for control of very fast processes, such as holding plasma in controlled thermonuclear reaction and control of hypersonic flying objects.

An additional benefit of the SN language is direct applicability of a wide range of formal techniques [30, 31, 32] for verification of concurrent programs that is an utmost significant aspect of reliable software design.

3. Overall organization of SNC VM and IDE

In appreciation of a project for full-scale thorough implementation of SNC paradigm, including dedicated hardware design, we present an experimental implementation, that further develops [33, 34], based on graphical editor of modeling system Tina [20] and a set of additional tools that includes: a converter of net files (*NDRtoSN*) [21]; compiler-linker (CL) of SN programs (*HSNtoLSN*) [24]; SN virtual machines implemented on multicore CPU [22] and GPU [23] (*SN-VM* and *SN-VM-GPU*, respectively). The general scheme of SNC IDE organization is represented in Fig. 4.

We draw SN programs in graphical editor *nd* of system Tina [20], low-level programs – directly, high-level programs (which use substitution of a transition by a subnet) – supplied with specification of transition substitution that is stored in *label* attribute of a transition with special syntax. Thus, initially we store SN programs as Tina files having graphical format (NDR). For fast data processing, two dedicated file formats for high-level (HSN) and low-level (LSN) Sleptsov nets have been developed, the corresponding converter *NDRtoSN* [21] from NDR format provided.

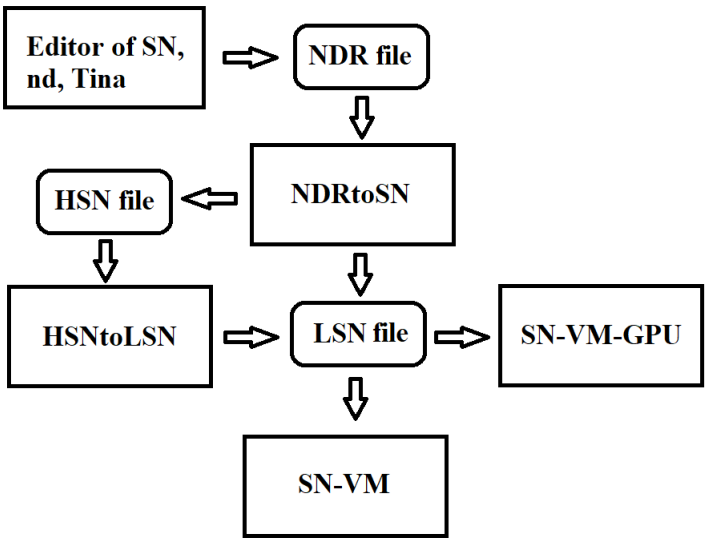


Fig. 4. SN IDE diagram.

High-level SN programs are converted by the compiler-linker of SN programs *HSNtoLSN* [24] into low-level SN programs which can be directly executed by an SN virtual machine implemented either on multicore CPU – *SN-VM* [22] or on GPU – *SN-VM-GPU* [23]. *SN-VM* serves, at first, for debugging SN programs offering step-by-step mode of execution with printing auxiliary information, though performance of *SV-VM* in automatic mode is rather impressive because of employing multicore facilities of CPU. *SN-VM-GPU* can approach theoretical performance of a hypothetical dedicated SN hardware machine using mass parallel matrices of threads provided by GPU. The corresponding benchmarks are described in Section 7.

The software is written in C language and compiled on Windows platform with *Dev-C* compiler and on Linux platform – with *gcc* compiler; GPU accelerator is implemented using NVIDIA CUDA [35].

Substitution of a transition for HSN implies mapping of contact places. In a simple case it can be implemented as merging (union) of the corresponding places of high-level and low-level nets. Though the basic operations have been implemented in a data-flow style consuming source data. Thus, in the general case, we need to copy data into input places of subnet and, after its completion, move result from output places of subnet. To briefly specify this kind of connection, dashed arcs were introduced in notation of SN programs [4, 17]. When transforming an HSN into LSN, they are expanded with insertion of the corresponding subnets COPY, CLEAN, and MOVE.

Let us consider examples of LSN and HSN which we will use later on for ex-

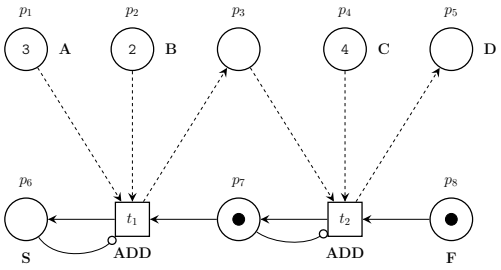


Fig. 5. HSN to compute $d := a + b + c, a := 3, b := 2, c := 4$, using LSN Add (Fig. 2).

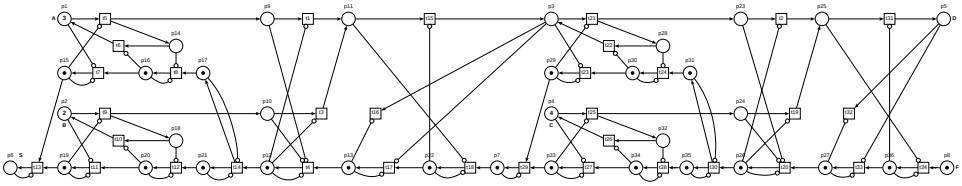


Fig. 6. Graph of final net (LSN).

plaining basic concepts of SN IDE work. LSN in Fig. 2 implements addition in three steps. HSN in Fig. 5 computes an arithmetic expression $z = a + b + c$ using two addition subnets. The HSN is transformed [4] into the corresponding LSN shown in Fig. 6 via insertion of two subnets ADD and expansion of dashed arcs via insertion of subnets COPY, CLEAN, and MOVE (studied in detail in Section 6). Note that here we use an explicit control flow and inline substitution of subnets [17]. As an alternative approach, data float concept can be implemented.

4. Formats of files and data converters

Since we use graphical editor *nd* of system Tina as a tool to integrate components of SNC VM and IDE, we deal, at first, with graphical specification of a place-transition net of system Tina, the corresponding file format is called NDR [20]. For simple and fast processing of SN programs, two file formats have been developed, the corresponding converters implemented: Low-level SN (LSN) that represents a kind of machine language; High-level SN (HSN) that represents an SN program with transition substitution for hierarchical modular program design. Besides, a raw matrix format MSN is provided for compatibility reasons.

4.1. *Low-level SN – LSN and MSN*

When transitions in a Sleptsov net are not substituted by some subnets, that is, there is no transition substitution, we call such a net the low-level Sleptsov net (LSN), LSN file [22] specifies the corresponding net structure and its initial marking.

Fig. 2 is a Low-level SN for computing $z = x + y$, where $x = 2$, $y = 3$. The input place *Start* (S) and the output place *Finish* (F) are a pair of dedicated control flow (CF) places. The place *Start* (S) indicates the beginning of a net's work and place *Finish* (F) indicates the completion of a net's work. The reachability graph of the SN, shown in Fig. 2, is represented in Fig. 3.

A low-level SN runs on the SN Virtual Machine (VM), and a universal SN [18] is a prototype of the SN Virtual Machine. We take the initial marking, the number of places, the number of transitions, and the arcs of an LSN as input data for SN VM. When the net halts, we obtain output data, which represent the result of an LSN run, in the form of the final marking. Sometimes a trace, represented by the number and multiplicity of transitions firing at each step (firing sequence), is of some interest, especially for debugging purposes. For convenience of programming in SNs, we use also priority arcs which allow us to represent some peculiarities of computing process in compact form, for instance in [36], issues of adding priority arcs to guess the path leading to the correct answer for nondeterministic computations are discussed. In Fig. 7 we present a priority LSN for computing $z = x + y$, its behavior shown in Fig. 8. Here, the priority arc only restricts the order of firing alternative transitions t_1 and t_2 (for illustrative purposes).

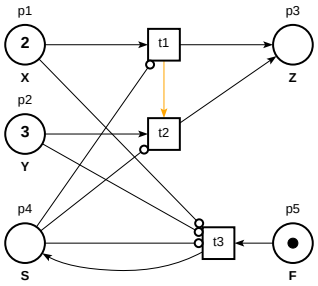


Fig. 7. A priority LSN for computing $z = x + y$

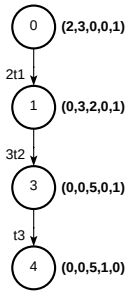


Fig. 8. Behavior of priority LSN shown in Fig. 7

An SN, having m places and n transitions, is specified by the following three matrices and one vector: B – an $m \times n$ matrix of transitions' incoming arcs; D – an $m \times n$ matrix of transitions' outgoing arcs; R – an $n \times n$ matrix of priority relation defined on a set of transitions; μ – an m vector of initial marking. We use nets with multiple arcs, arc multiplicity represented by a natural number, and value -1 represents an inhibitor arc. For reasons of compatibility with some verification tools, we also use raw matrix format represented by the sequence: m, n, B, D, R, μ; usually we insert blank lines to separate matrices. A variant, using transitive closure RC of the priority relation matrix R, is called an MSN (matrix SN) and applied as input for *SN-VM-GPU* [23].

Matrix representation of LSN shown in Fig. 7 follows:

$$m = 5,\ n = 3,\ B = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 0 & 1 \end{bmatrix},\ D = \begin{bmatrix} 0\ 0\ 0 \\ 0\ 0\ 0 \\ 1\ 1\ 0 \\ 0\ 0\ 1 \\ 0\ 0\ 0 \end{bmatrix},\ R = \begin{bmatrix} 0\ 1\ 0 \\ 0\ 0\ 0 \\ 0\ 0\ 0 \end{bmatrix},$$

$$\mu = (2, 3, 0, 0, 1). \quad (1)$$

In LSN file, we use sparse way of matrices and vectors specification, moreover, we consider an overlapped specification of B, D, R just listing the net arcs, classifying them into the following types: regular arc from place to transition, inhibitor arc from place to transition, arc from transition to place, priority arc from transition to transition. Formal specification of LSN file format and its example for the priority SN of addition (Fig. 7) are represented in Table 1.

Table 1. Table of LSN format and an example for SN shown in Fig. 7.

LSN format	LSN example
m n k l nst	6 3 12 3 0
v1 v2 w	5 3 1
	4 2 -1
	4 1 -1
	1 1 1
	4 3 -1
	1 3 -1
	-4 3 1
	2 2 1
	2 3 -1
	-3 1 1
	-3 2 1
	-1 -2 0
p mup	1 2
	2 3
	5 1

LSN file is divided into three sections: the header, specification of arcs, specification of initial marking; rows starting with ";" character are considered comments. In the LSN header line, m and n specify, as it was defined above, the number of places and transitions, respectively, k – the total number of arcs, l – the number of nonzero initial markings, and nst – the number of substitution transition (zero for LSN). An arc is specified by a triple $v1$ $v2$ w, which can be divided into the four following types:

(1) If $v1 < 0, v2 < 0$, a priority arc from transition $v1$ to transition $v2$, multiplicity is not applicable, $w = 0$.

(2) If $v1 > 0, v2 > 0, w > 0$, a regular arc from place $v1$ to transition $v2$ of multiplicity w.
(3) If $v1 > 0, v2 > 0, w < 0$, an inhibitor arc from place $v1$ to transition $v2$, multiplicity $w = -1$.
(4) If $v1 < 0, v2 > 0$, an arc from transition $v2$ to place $v1$ of multiplicity w.

Marking is represented in sparse form by a sequence of pairs $p\ \mu(p)$ for nonzero markings. An LSN file represents a kind of SN machine language file to run on SN (virtual) machine.

4.2. *High-level SN - HSN*

Conventional software design is based on hierarchical structure of a program, represented by calls of functions. For the SN program design, we use substitution of a transition by a subnet and dashed arcs to denote copying and moving data [4, 17]. The concept was preliminary considered in Section 3 and is described in detail in Section 6. An SN, that contains substitution of transitions, is called a high-level Sleptsov net (HSN); HSN file [24] can be also recognized by its header, containing nonzero value of nst – the number of substitution transitions. Compared to LSN, an HSN file contains the fourth section that specifies the transition substitution.

The transition substitution by a subnet requires to specify the subnet name and mapping of its contact (input and output) places into places of the high-level net. Since we implement here the explicit control flow approach, we classify the set of contact places into four classes: start and finish for the control flow; input and output for data.

Substitution of a transition is specified by the header row followed by a list of place mapping rows. The header row contains substituted transition number $tnum$, the number of place mappings $pmnum$, the subnet file name $fname$. A place mapping row (totally $pnnum$ rows) contains HSN place number hp and subnet place number lp. Each line represents a pair of place mappings separated by space. At first, the mappings of the data input places are listed, then – the mappings of the data output places, and finally – the mappings of the control flow places.

The different types of place mapping are specified using four combinations of the mapping components sign as follows:

(1) If $hp > 0, lp > 0$, input data place mapping.
(2) If $hp > 0, lp < 0$, output data place mapping.
(3) If $hp < 0, lp > 0$, start control-flow place mapping.
(4) If $hp < 0, lp < 0$, finish control-flow place mapping.

Let us consider an example of addition of three numbers shown in Fig. 5. We use Table 2 to illustrate the composition of the corresponding HSN. In the file header, 8 places, 2 transitions, 12 arcs, 5 non markings, all 2 transitions need to be substituted in the HSN section. And then specification of 12 arcs are listed, followed

by 5 nonzero place markings. In the HSN section, there are 2 transitions which are substituted by a subnet (add.lsn), and each transition has 5 pairs of places mapping.

Table 2. HSN file format and example for addition of three numbers shown in Fig. 5.

HSN format	HSN example
m n k l nst	8 2 12 5 2
; arcs and marking	
...	...
; transition substitution section	
$tnum_1$ $pmnum_1$ $fname_1$	1 5 *add.lsn*
; place mappings	
hp lp	1 1
	2 2
	3 -3
	-6 4
	-7 -5
$tnum_2$ $pmnum_2$ $fname_2$	2 5 *add.lsn*
hp lp	3 1
	4 2
	5 -3
	-7 4
	-8 -5

4.3. *Tina graphical editor nd and NDR file format*

LSN and HSN files can be inputted and edited manually in a text editor. Though graphical programming, especially for an intrinsically graphical language as SN, is considered more user-friendly technology. We use graphical editor *nd* of system Tina [20] to draw HSNs and LSNs which is rather simple and convenient in its laconic concept. Besides, system Tina allows us to apply a set of tools for verification of concurrent SN programs [37]. From graphical environment of *nd*, we can stars directly tools for state space and structural analysis of nets as well a stepper-simulator convenient for debugging SN programs. Professional command line tools of Tina for state-space exploring and model-checking are well known, especially because their ability to handle big nets.

Graphical editor *nd* provides menu items and hot keys to draw a place-transition net with multiple and inhibitor arcs, connecting places and transitions, and also with priority arcs between transitions. It provides the following attributes of a place: name, label, and marking. Attributes of a transition are: name, label, and interval specifying timed characteristics; we do not use timed characteristics for programming in SNs. Tina provides 5 types of arcs directed from a place to a transition of which we use a regular and an inhibitor arc. The arc weight attribute represents the arc multiplicity; for inhibitor arcs we use unary multiplicity only. Arcs are draws as straight initially and then their curvature can be adjusted moving two handlers

which appear when pointing out the corresponding arc.

Present version of *nd* does not provide Sleptsov firing rule of transitions that is why application of a built-in stepper simulated is rather limited. Though we use it for debugging purposes because when programming in SN, we deal with nets which invariant with regard to the actual sequence of transitions firing, computing the same result.

Formally, using *nd*, we can not specify the transition substitution and dashed arcs stipulated by the SN programming technology [4]. To specify substitution of a transition, we use its label attribute which content is not formalized within Tina and devised for descriptive purposes. As for the dashed arcs, they are implicitly defined by the set of subnet contact places which are not the two control flow places Start and Finish. Though we have no possibility to emphasize the transition substitution details with dashed lines of the corresponding arcs. Specifications of the transition substitution label format are represented with Table 3.

Table 3. Format of transition substitution label and an example for t_1 substitution in SN shown in Fig. 5.

Label format	Label example
*HSN(	*HSN(
subname	add.lsn
atype hpname lpnum	i p1 1
	i p2 2
	o p3 3
	s p6 4
	f p7 5
)	)

Prefix "*HSN(" serves to recognize the transition substitution label; *subname* specified the file name of subnet; a triple *atype hpname lpnum* specifies mapping of a contact place with the arc type, high-level net place name, and low-level net place number, respectively; the above mentioned four types of contact places are classified: 'i' – input, 'o' – output, 's' – start, 'f' – finish; suffix ")" completes the label specification. The transition name within high-level net is recognized from the NDR file row of transition specification containing the label. Let us write two transition substitution labels (t_1 and t_2, respectively) for SN shown in Fig. 5:

```
*HSN(add.lsn i p1 1 i p2 2 o p3 3 s p6 4 f p7 5)

*HSN(add.lsn i p3 1 i p4 2 o p5 3 s p7 4 f p8 5)
```

Tina uses two file formats to store a net – graphical NDR and textual NET, – of which we use NDR only having dedicated LSN and HSN formats for specifying SNs in textual form. Formal specification of the file formats id a part of Tina [20] documentation. Here we provide a brief overview for NDR format. There are 4 types

of line recognized by the first character: 'p' – place, 't' – transition, 'e' – arc, 'h' – header. Place specification line contains place name and label, coordinates, and marking. Transition specification line contains transition name and label, coordinates, and possible timed characteristics. Arc specification contains names of start and end node, arc type and weight, and also curvature parameters. The header specifies the net name and default scale for graphical representation.

4.4. *Converting NDR into LSN and HSN*

To develop a dedicated converter of an NDR file into an LSN or an HSN file [21], we use a library of components early developed for Tina plug-ins *Deborah* and *Adriana* [38] and also for recently issued software *ParAd* [39]. We read and parse lines of NDR file processing them according to the line type considered in the previous subsection. We create tables of transitions, places, and arcs, containing their names and the transition substitution labels. Then we generate either LSN or HSN file based on the presence of transition substitution labels. Generator inserts also comment lines explaining formatting accomplished with tables of place and transition names. These information helps to grasp correspondence of NDR node names to LSN/HSN node numbers. The command line format uses two file names for input and output file, respectively:

```
NDRtoSN file1.ndr file2.lsn
NDRtoSN file1.ndr file2.hsn
```

Note that, since the program uses NDR file format, it can be implemented as *nd* plug-in similar to *Deborah* and *Adriana* [38].

5. SN Virtual Machine

The universal Sleptsov net [18] is considered as a prototype of the Sleptsov net Virtual Machine (VM). SN VM is implemented as a software emulator of SN behavior ("token game"), which allows firing a transition in multiple instances at a single step. An SN specification, in the form of LSN file, represents VM input, and then, as a result of the net run, we obtain VM output in the form of SN final marking. A marking, more precisely its subset with respect to places specifying variables [4], represents data while the SN graph represents a program from conventional point of view. SN Virtual Machine has been developed in two variants, a single-thread and a multi-thread based on OpenMP [40] facilities.

5.1. *Algorithm SN VM work*

Starting from an initial marking, SN VM fires transitions of a given SN while there are firable transitions. When the net halts, its final marking represents a result of computations. Thus, generally, VM is organized as a repetition of an SN step

represented by the following three stages according to the SN transition firing rule [4]:

(1) Find fireable transitions and their firing multiplicities (vector f) by Algorithm 1.
(2) Choose the transition to be fired ($firing\_n$) at a step (with firing multiplicity fm) according to the flowchart shown in Fig. 9; for priority LSN, the fireable transition with the highest priority is chosen by Algorithm 3.
(3) Fire transition $firing\_n$ and calculate the next marking mu by Algorithm 2.

To implement the SN transition firing rule, we start with computing the transition firing multiplicity and the firing transition choice, defined as Algorithm 1 and flowchart in Fig. 9, respectively; Algorithm 1 is specified in a pseudo-language notation. Calculations of transition firing multiplicity start on each incoming arc of a transition.

Variables m and n, matrices B, D, and R, and the current marking mu (a variant of notation for μ) specify an SN as it was considered in Section 4. Variable $fireable$ denotes the transition firing multiplicity. Variable $fireable\_m$ denotes the current minimal transition firing multiplicity; it is assigned an initial value equal to infinity (the maximal integer value in the current implementation). mu denotes the current marking at a step, t and p represent the current transition and place, respectively. To calculate $fireable$, we use a conditional statement. When $B[p-1][t-1] > 0$, it is a regular arc, and $fireable$ is equal to $mu[p-1]/B[p-1][t-1]$. When $B[p-1][t-1] = -1$, it is an inhibitor arc and there are two situations as follows: if $mu[p-1] > 0$, then $fireable = 0$, otherwise, $fireable = \infty$. Finally, $fireable\_m$ is obtained by comparison and stored in f – a vector of transition firing multiplicity, where the vector elements are considered also as indicators of fireable transitions (when greater than zero). The purpose of taking the minimum value is to ensure that the negative marking will not appear after firing f_i instances of transition t_i at a step.

The flowchart in Fig. 9 specifies a random choice of a fireable transition to be fired. It uses such input data as the number of fireable transitions nf. Within the flowchart, ct is the number of firable transitions, rn denotes a random number, l denotes the remainder of ct divided by rn and $firing\_n$ denotes the number of transition to fire. To choose a fireable transition t randomly, each transition is traversed. When the firing multiplicity of the transition t is greater than 0 and $l == ct$, t is chosen as the transition to fire and the transition number is stored as $firing\_n$.

Algorithm 1 Computing transition firing multiplicity

Input: mu: current marking; B: matrix of incoming arcs of transitions

Output: $f[n]$: a vector of transition firing multiplicity

```
1:  for t = 1 → n do {
2:      fireable_m ⇐ ∞;
3:      for p = 1 → m do {
4:          if B[p − 1][t − 1] > 0 then
5:              fireable ⇐ mu[p − 1]/B[p − 1][t − 1];
6:          else
7:              if B[p − 1][t − 1]) < 0 then
8:                  if mu[p − 1] > 0 then
9:                      fireable ⇐ 0;
10:                 else
11:                     fireable ⇐ ∞;
12:             else
13:                 fireable ⇐ ∞;
14:
15:         if fireable < fireable_m then
16:             fireable_m = fireable;
            }
        f[t − 1] ⇐ firable_m; }
```

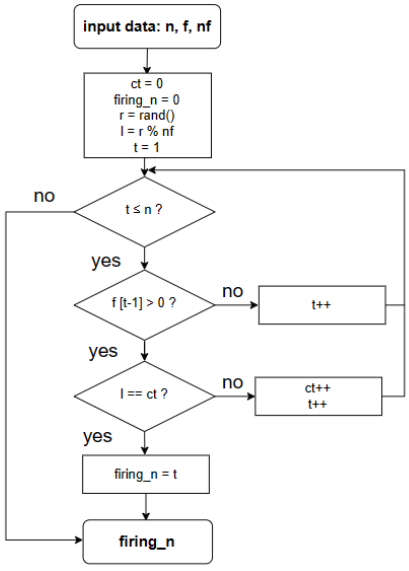


Fig. 9. A flowchart for the firing transition choice.

Transition firing causes the current marking change. In order to obtain the new current marking, the SN next marking expression [4] is specified as Algorithm 2. tb and td denote the current element in matrix B and D, respectively; $firing\_n$ denotes the firing transition number; fm denotes the multiplicity of transition firing. When $tb < 0$, it is inhibitor arc, and we only add $td \cdot fm$ tokens to the places marking. Otherwise, for a regular arc, we extract $tb \cdot fm$ tokens and add $td \cdot fm$ tokens to the places marking. Note that when both values tb and td are nonzero, there is a kind of self-loop with the current place.

Algorithm 2 Computing the next marking

Input: mu: current marking; B and D: matrices of transition incoming and outgoing arcs, respectively; $firing\_n$: the number of firing transition; fm: the multiplicity of transition to be fired

Output: mu: the new current marking (after firing a transition)

1: **for** $p = 1 \rightarrow m$ **do**
{
2: $tb \Leftarrow B[p-1][firing\_n-1]$;
3: $td \Leftarrow D[p-1][firing\_n-1])$;
4: **if** $tb < 0$ **then**
5: $mu[p-1] \Leftarrow mu[p-1] + td * fm$;
6: **else**
7: $mu[p-1] \Leftarrow mu[p-1] - tb * fm + td * fm$;
}

For priority nets, the transitions choice is also based on their priorities and only one fireable transition with the highest priority fires at a step. For firing the transition with the highest priority, we remove low priority firable transitions as shown in Algorithm 3. A matrix RC represents a transitive closure of the priority relation matrix R. $RC[t_1-1][t_2-1] > 0 \&\& f[t_1-1] > 0$ indicates that firable transition t_1 has higher priority than t_2 that, because of it, can not fire $f[t_2-1] = 0$.

Algorithm 3 Remove Low Priority Transitions

Input: n: the number of transitions; f: a vector of minimum multiplicities of fireable transitions; R: priority relation matrix

Output: f: updated vector of transition firing multiplicity

1: **for** $t_1 = 1 \rightarrow n$ **do**
2: **for** $t_2 = 1 \rightarrow n$ **do**
3: **if** $RC[t_1-1][t_2-1] > 0 \&\& f[t_1-1]! = 0$ **then**
4: $f[t_2-1] = 0$;

Note that, based on an initial matrix of transition priority relation R, we com-

pose a matrix of its transitive closure RC using an ad-hoc algorithm.

The time complexity of SN VM depends on doubly nested loops and an indeterminate constant k, where k is the number of steps. The nested loop contains two data sizes, namely m (the number of places) and n (the number of transitions), so the time complexity of the program is $O(k(m \cdot n))$. When the values of m and n are very close, the time complexity can be approximated as $O(k \cdot n^2)$.

5.2. *An example of SN VM run*

We use priority LSN for computing $z = x + y$, shown in Fig. 7, as an example to explain how SN VM runs, detailed representation of (1) follows: Fig. 10 and Fig. 11 specify a pair of matrices B and D, which are matrices of transition incoming and outgoing arcs; in Fig. 12, the priority relation matrix is shown, which indicates that transition t_1 has higher priority than t_2; the matrix coincides with its transitive closure RC; the initial marking is $(2, 3, 0, 0, 1)$.

P \ T	1	2	3
1	1	0	-1
2	0	1	-1
3	0	0	0
4	-1	-1	-1
5	0	0	1

Fig. 10. Matrix B of transition incoming arcs

P \ T	1	2	3
1	0	0	0
2	0	0	0
3	1	1	0
4	0	0	1
5	0	0	0

Fig. 11. Matrix D of transition outgoing arcs

T \ T	1	2	3
1	0	1	0
2	0	0	0
3	0	0	0

Fig. 12. Priority relation matrix $R = RC$.

Let us trace VM work on the SN:

(1) Find fireable transitions and their firing multiplicities: for t_1, $fireable\_m = 2$; for t_2, $fireable\_m = 3$. Vector f (2 fireable transitions): $(2, 3, 0)$
(2) The fireable transition with the highest priority is t_1. In this case, vector f is updated (1 fireable transition): $(2, 0, 0)$. Choose the only transition to fire: $firing\_n = t_1$, $fm = 2$.
(3) Fire transition t_1:

$$\begin{aligned} mu &= (2, 3, 0, 0, 1) - (1, 0, 0, 0, 0) * fm + (0, 0, 1, 0, 0) * fm \\ &= (2, 3, 0, 0, 1) - (2, 0, 0, 0, 0) + (0, 0, 2, 0, 0) \\ &= (0, 3, 2, 0, 1). \end{aligned}$$

(4) Find fireable transitions and their firing multiplicities: t_2, $fireable\_m = 3$. Vector f (1 fireable transition): $(0, 3, 0)$.
(5) Choose the transition to fire: $firing\_n = t_2$, $fm = 3$.

(6) Fire transition t_2:

$$\begin{aligned} mu &= (0,3,2,0,1) - (0,1,0,0,0) * fm + (0,0,1,0,0) * fm \\ &= (0,3,2,0,1) - (0,3,0,0,0) + (0,0,3,0,0) \\ &= (0,0,5,0,1). \end{aligned}$$

(7) Find fireable transitions and their firing multiplicities: t_3, $fireable\_m = 1$. Vector f (1 fireable transition): $(0,0,1)$.
(8) Choose the transition to fire: $firing\_n = t_3$, $fm = 1$.
(9) Fire transition t_3:

$$\begin{aligned} mu &= (0,0,5,0,1) - (0,0,0,0,1) * fm + (0,0,0,1,0) * fm \\ &= (0,0,5,0,1) - (0,0,0,0,1) + (0,0,0,1,0) \\ &= (0,0,5,1,0). \end{aligned}$$

Since there is no firable transitions, the SN program halts. The final marking is $(0,0,5,1,0)$ that means that the addition result is 5 as represented by marking of p_3.

5.3. *Parallel implementation of SN VM using OpenMP*

OpenMP [40] provides a directive-based programming approach for writing multi-threaded applications and composing parallel versions of programs. We use OpenMP for the parallel implementation of Algorithm 1 and Algorithm 2 to improve their performance. We use the following facilities of OpenMP: parallel implementation of loop *for*; SIMD (Single Instruction Multiple Data) facilities; unrolling loops. Moreover, to speed-up parallel implementation for loops which have dependence between the loop passages, expressed in the form of a single operation, we use *reduction* on such operations as sum and minimum.

Let us illustrate application of OpenMP directives to speed-up some basic stages of the SN step implementation:

```
// Find firable transitions
#pragma omp parallel for private(p,t,firable) num_threads(nth)
for(t=1; t<=n; t++) {
int firable_m = INT_MAX;
#pragma omp simd reduction(min:firable_m)
#pragma unroll
for(p=1; p<=m; p++) {
...
}
...
}
...
// Count the number of firable transitions
```

```
#pragma omp parallel for reduction(+: nf) num_threads(nth)
for(t = 1; t <= n; t++) {
nf +=(f[t-1]>0) ? 1 : 0;
}
```

Benchmarks, considered in Section 7, show that for parallel implementation of nested loops, a combination of "omp parallel for" with "omp simd" and "unroll" provides considerably better speed-up than "collapse" option devised for parallel execution of nested loops.

5.4. *GPU SN VM accelerator*

Foreseen SNC hyper-performance is achieved in case of dedicated hardware implementation of the SN machine in the form of re-configurable computing memory that implements, on memory words, basic operations of division-subtraction and multiplication-addition considered while prototyping SN machine in the form of a universal SN [18]. Though at the present time rather reasonable compromise can be achieved using available mass parallel computing devices such as GPU. For the implementation, we choose NVIDIA GPU architecture 35 (for compatibility reasons) and CUDA toolkit.

Programming in CUDA [35], is clear in its concept of a two-level structure of the computing grid which consists of blocks of threads: the grid is 1 or 2 or 3 dimensional array of blocks, and a block is 1 or 2 or 3 dimensional array of threads. We take into consideration rather pressing restriction of CUDA architecture 35, that we can synchronize only threads of a single block that yields two approaches embodied into two *GPU-SN-VM* alternative modules: using a single block with entire VM implemented as CUDA kernel (suffix "1b"); using many blocks with VM implemented on host calling a few kernels to implements a single SN step (suffix "fk").

An SN represents a two dimensional structure, though during a single step of SN firing, the number of required threads is changing and, after each of the following basic stages, synchronization is required:

(1) Compute firing conditions on arcs (incoming arcs of transpositions) – $m \times n$ matrix structure.
(2) Compute firing multiplicity of transitions – n array structure ($m \times n$ matrix structure in case of using reduction on minimum).
(3) Remove low priority firable transitions – $n \times n$ matrix structure.
(4) Count the number of firable transitions – a single thread (n array structure in case of using reduction on sum).
(5) Find firing transition based on a random number – a single thread.
(6) Compute the next marking – m array structure.

Because of absence of synchronization between blocks in CUDA architecture 35, within a few kernel implementation (suffix "fk"), we use specific structures of grids

for the above mentioned stages, each stage is represented as a separate kernel, stages (4) and (5) overlapped. For each kernel, we specify a dedicated grid structure. It yield the maximal possible parallelism, say for stage (1) with:

```
__global__ void fire_arc(int *b, int *mu, int m, int n, int *y)
{
  int i = threadIdx.x;
  int j = blockIdx.x;
  MELT(y,i,j,m,n) = (MELT(b,i,j,m,n)>0)? mu[i] / MELT(b,i,j,m,n) :
      (MELT(b,i,j,m,n)<0)? ((mu[i]>0)? 0: INT_MAX): INT_MAX;
}
```

We use macros MELT for the matrix element access. An auxiliary matrix Y serves for storing the current SN step actual data: firing multiplicity of arcs, firing multiplicity of transitions (stored in the first row, overwriting the arc multiplicity), the number of firing transition and its firing multiplicity (in the first two elements of the first row, overwriting the firing multiplicity of transitions). After each kernel call, we synchronize the device:

```
dim3 block (m);
dim3 grid  (n);
...
fire_arc<<<grid, block>>>(d_b, d_mu, m, n, d_y);
cudaDeviceSynchronize();
...
dim3 block3 (m);
dim3 grid3  (1);
...
next_mu<<<grid3, block3>>>(d_b, d_d, d_mu, m, n, d_y, f[0], f[1]);
cudaDeviceSynchronize();
```

Remind that we allocate the device memory with *cudaMalloc()* and copy matrices B, D, RC, and initial marking vector mu from host to device, and the resulting marking d_m from device to host with cudaMemcpy(); data structure copies within a device have prefix "d_".

For the simplified version (suffix "1b"), based on a single block, we use the linear block structure because of rather essential limitation on the total number of threads within a block (about 1024). Within a thread, we organize loops to process two-dimensional matrices and select thread numbers according to the current vector size (m, n or 1).

6. Compiler-linker of SNs

Conventional programming technology [42] uses a compiler to convert a program, written in an algorithmic language (source code), into machine code with unresolved

external references (object code), then a linker, processing object files and libraries, allocates object files and resolves external references to produce an executable program in machine language. Recent update is represented by graphical design with UML [19] that further develops principles of R-technology of programming [41] with respect to the object-oriented design.

Since we use the same graphical language of SNs on different levels of program hierarchy, implemented via the operation of transition substitution, functions of compiler are rather limited and consist in expansion of dashed arcs for HSN. Functions of a linker consist in allocation and connection of subnets (via global enumeration of nodes), and implementation of place mapping (passing parameters). Actually, the tool is called *HSNtoLSN* and it accepts HSN and LSN file names as parameters, also it uses LSN files of subnets mentioned within HSN file and, implicitly, subnets COPY and CLEAN_MOVE for expansion of dashed arcs.

Note that in the present implementation, we use inline style of transition substitution and explicit control flow [4].

6.1. *Expansion of dashed arcs*

Since basic SM modules, which implement basic arithmetic and logic operations, are designed as consuming tokens from their input places and moving resulting tokens into their output places, we support an abbreviation, expressed via dashed arcs connecting a module, for copying its input and output parameters [4].

The dashed arc represents the mapping relationship of places between HSN and LSN, as shown in Fig. 13. We call it a dashed input arc, when the dashed arc points from the place to the transition and we call it a dashed output arc, when the dashed arc points from the transition to the place.

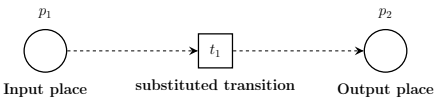


Fig. 13. Dashed Arcs

Shown in Fig. 14, the COPY module copies the data from the HSN place to the corresponding LSN input place according to the place mapping. That is a dashed input arc corresponds to a copy module.

Shown in Fig. 15, the CLEAN_MOVE module transfers the data of LSN's output place to the corresponding HSN's place according to the mapping relationship between LSN's output place and HSN's place. That is a dashed output arc corresponds to a CLEAN_MOVE module.

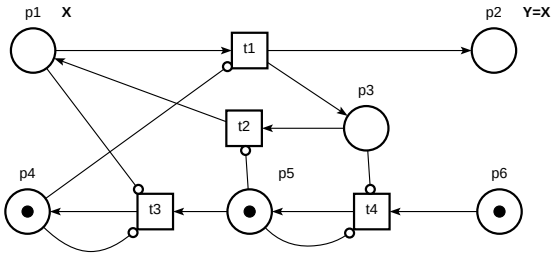


Fig. 14. COPY Module: copy the data from X to Y.

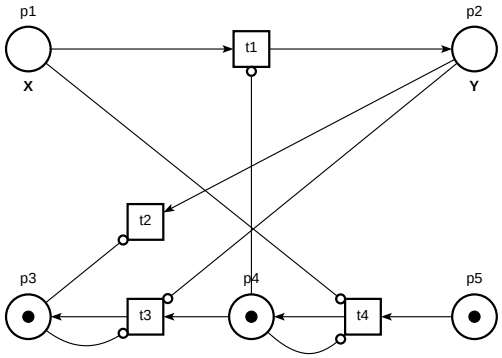


Fig. 15. CLEAN_MOVE Module: 1) clean data in Y; 2) move the data from X to Y.

6.2. *Basic Data Structures*

In the process of HSN conversion into LSN, we use data structures to save the information obtained from the files. The final LSN is generated using the information within the data structures. The following Fig. 16, 17, 18 show the relationship between the basic structures in the form of multi-linked lists.

As shown in Fig. 16, *the structure HSN* stores the composition information of an HSN, including the struct pointer to the current LSN part *struct* $lsn * l$, the total number of transition substitutions nst, the struct pointer to the transitions substitution and places mapping part *struct* $tspm * t$.

The structure LSN stores the composition information of an LSN, including the total number of places m, the total number of transitions n, the total number of arcs k, the total number of non zero marking places l, the pointer to the initial marking mu, and a struct pointer to the arc's array *struct* $arc * a$.

The structure arc stores the information of an arc, including the connected place p, transition t, and the weight of the arc w. The above three attributes represent the direction and type of an arc as specified in Section 4.1.

The structure of transitions substitution and places mapping *tspm* stores the information of each transition substitution, including the number of the substituted

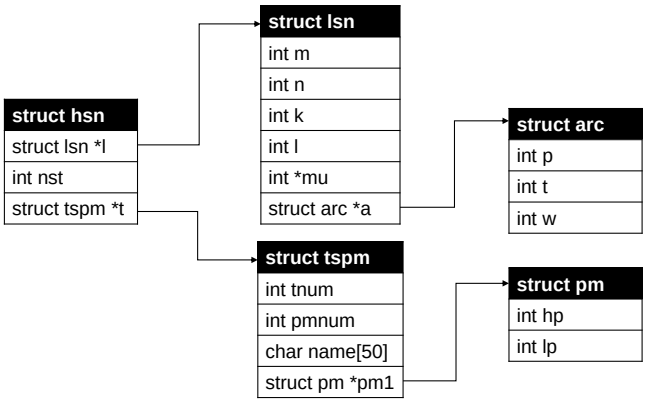


Fig. 16. Structure HSN.

transitions *tnum*, the total number of places mapping in each transition substitution *pmnum*, the file name of the subnet *name*[ ], and the struct pointer to the array of place mapping *struct pm* $*$ *pm*1.

The structure of places mapping *pm* stores a pair of the place number within HSN *hp* and the place number within subnet *lp* that specifies the place mapping.

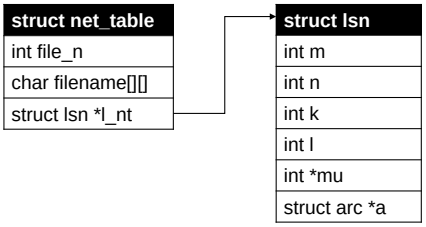


Fig. 17. Structure *net_table*.

As shown in Fig. 17, the structure *net_table* stores all the LSN information required in the process of converting an HSN to an LSN, including the number corresponding to each LSN $file\_n$, the net name $filename$[ ][ ], and it links the structure LSN via *struct lsn* $*$ *l_nt*.

As shown in Fig. 18, the structure *resulting_net* stores general net information during the HSN conversion to LSN, including the struct pointer to the place number array *struct p_num* $*$ *pn*, the struct pointer to the transition number array *struct t_num* $*$ *tn*, the pointer to the initial marking $*mark$, the struct pointer to the array of converted arcs *struct arc* $*$ *new_a*, and the struct pointer to the array of converted LSN *struct lsn* $*$ *l_result*.

The structure of the place number *p_num* stores the number of a definite place in the process of the HSN conversion to LSN, including the place number before con-

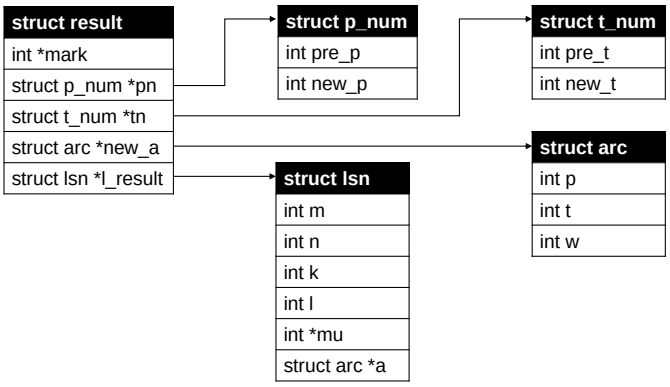


Fig. 18. Structure *resulting_net*.

version *pre_p*, and the sequential place number rearranged after conversion *new_p*.

The structure of the transition number *p_num* stores the number of a definite transition in the process of the HSN conversion to LSN, including the transition number before the conversion *pre_t* and the sequential transition number rearranged after the conversion *new_t*.

6.3. *Basic Algorithms of SN CL*

To implement the conversion from HSN to LSN, we read the HSN file and check input data, store the data within linked data structures (described in the previous section), compile and link modules of subnets. The final LSN is formed according to the specified substitution of a transition and the place mapping, including expansion of dashed arcs, and is written into a new LSN file to run on an SN (virtual) machine.

The algorithms of the conversion process and their call-return relationships are shown in Fig. 19, some basic algorithms are described in detail later on in this section.

According to the HSN file, we read its LSN part. Then, for each transition substitution, we read the subnet modules that substitute the corresponding transition. In case $hp > 0 \&\& lp > 0$, we insert the COPY module; and in case $hp > 0 \&\& lp < 0$, we insert the CLEAN_MOVE module. Subnet modules are not inserted if the place map represents control flow. In addition, we enumerate each subnet module, and store each subnet module within the corresponding structure variable. At the same time, we create a table to record the subnet modules.

After reading the subnets, the next step is to compile and link them, which is defined as Algorithm 1 (specified separately), it means using the storage information of the net table to rearrange the subnet modules. During this process, we merge places preserving their marking, re-numerate places and transitions, connect them by arcs (with respect to new numbers of nodes), add new transitions for splitting and merging control flow, and finally obtain an LSN, write it into a file to run it on

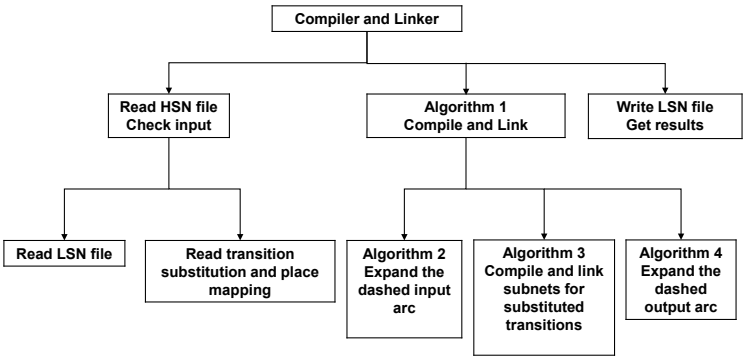


Fig. 19. Hierarchy scheme of compiler-linker algorithms.

an SN machine.

Algorithm 1 Compile and Link

Input: struct result *mt, struct net_table *nt, struct hsn *h
Output: struct result mt
Calculate the number of places, transitions and arcs based on the table;
Allocate memory for final LSN;
for (*each of nst*) **do**
    Compile subnets for substitute transitions;
    for (*each of pmnum*) **do**
        if ($hp > 0 \&\& lp > 0$) **then**
            Expand the dashed input arc;
        if ($hp > 0 \&\& lp < 0$) **then**
            if (The number of input data is more than 0) **then**
                Add split transition, connect it to the start-control-flow places of COPY module based on $Array4$;
                Add joint transition, connect it to the finish-control-flow places of COPY module based on $Array5$, and connect it to the start-control-flow place of substituion net based on $Array2$;
            Expand the dashed output arc
        if ($hp < 0 \&\& lp > 0$) **then**
            Preserve the start-control-flow place of the HSN;
            Connect it to split transition;

            Preserve the finish-control-flow place of the HSN;
            Connect it to joint transition;

Algorithm 2 defines an expansion of the dashed input arc using the COPY

module. Its role is to transmit the input data from the HSN to the subnet input place. It also implements the re-numeration of places, the assignment of tokens, and the updating the place number of each arc, the original weight of each arc is retained. After that, it re-numerates the transitions and updates the transition number of each arc.

The subnets, which substitute HSN transitions, are compiled and linked as defined in Algorithm 3. At first, we record the original place number in the structure variable. Then, depending on the place type of the subnet, according to the place mapping information, we process arcs, connecting the substitution transition: if it is an input or output data place, we merge it with the data place of COPY or CLEAN_MOVE; if it is the start-control-flow place, we connect it with the joint transition; if it is a finish-control-flow place, we provide its merging with the start-control-flow place of CLEAN_MOVE. Further, we re-numerate places, assign the corresponding markings, update the place number for each arc, and retain the original weight of each arc. Finally, we re-numerate transitions and update the transition number of each arc.

Algorithm 4 defines an expansion of the dashed output arc using the CLEAN_MOVE module, which role is to transmit the output data of the subnet to the output place of the HSN. At first, we determine the type of place connected via CLEAN_MOVE: if it is an input place, we merge it with the output place of the subnet; if it is an output place, we use the place number and the marking value corresponding to the HSN; if it is a start-control-flow place, we merge it with the finish-control-flow place of the subnet; the other places are re-numbered. At the same time, the token assignment is completed in the above process, the place number of each arc is updated, and the original weight of each arc is retained. After that, we re-numerate the transitions and update the transition number of each arc.

If there are elementary transitions in HSN, we keep the information of arcs connected to them, including the place number, the transition number, and the arc weight.

For the transition substitution example, shown in Fig. 5, the final LSN is written into a new textual file as shown in Table 4. The final LSN contains 36 places, 34 transitions, and 126 arcs. We draw the net graph, corresponding to the file, in Fig. 6.

Table 4. Final LSN file for HSN shown in Fig. 5.

6.4. *Evaluation of algorithm complexity*

As if follows from the hierarchical scheme of algorithms shown in Fig. 19, Algorithm 1 calls Algorithm 2, Algorithm 3, and Algorithm 4. Thus, Algorithm 1 has the highest complexity. This section first analyzes the complexity of Algorithms 2-4 to estimate the complexity of Algorithm 1.

Suppose that, the HSN has M places, N transitions, K arcs, nst transitions are substituted, and the maximal subnet has m places, n transitions, k arcs, and

Content annotation	File content
m n k l nst	36 34 126 23 0
;comments	;arcs
p_{arc1} t_{arc1} w_{arc1}	9 1 1
p_{arc2} t_{arc2} w_{arc2}	9 4 -1
p_{arc3} t_{arc3} w_{arc3}	10 3 1
p_{arc4} t_{arc4} w_{arc4}	10 4 -1
...	...
p_{arc123} t_{arc123} w_{arc123}	-36 34 1
p_{arc124} t_{arc124} w_{arc124}	8 34 1
p_{arc125} t_{arc125} w_{arc125}	-7 29 1
p_{arc126} t_{arc126} w_{arc126}	7 29 -1
;comments	;non zero marking
place_number marking	1 3
place_number marking	2 2
place_number marking	4 4
...	...
place_number marking	35 1
place_number marking	36 1

pmnum places mappings within HSN, and, moreover, there is no certain relationship between these parameters.

In Algorithm 2, the COPY module contains 6 places, 4 transitions, and 15 arcs. Therefore, to compile a COPY module, the number of loops, required to re-numerate the places and copy the corresponding tokens, and update the place number of each arc, is $m \times k = 90$. The number of loops, required to re-numerate the transitions and update the transition number of each arc, is $n \times k = 60$.

Then in Algorithm 3, the number of loops, required to re-numerate the places, copy the corresponding tokens, and update the place number of each arc, is $m \times (pmnum + k)$. The number of loops, required to re-numerate the transitions and update the transition number in each arc, is $n \times k$. We obtain the time complexity of the transition substitution as $O(m \times (pmnum + k) + n \times k)$, and the space complexity as linear.

In Algorithm 4, the CLEAN_MOVE module contains 5 places, 4 transitions, and 13 arcs. Therefore, to compile a CLEAN_MOVE module, the number of loops, required to re-numerate the places and copy the corresponding tokens, and update the place number of each arc, is $m \times k = 65$. The number of loops, required to re-numerate the transitions and update the transition number in each arc, is $n \times k = 52$.

So the time complexity and space complexity of compiling a COPY module or CLEAN_MOVE module are both $O(C_1) = O(1)$, we obtain the constant C_1 summing up the mentioned above constants.

Therefore, in Algorithm 1, because there are nst transitions to substitute in an HSN, the number of required loops to substitute all transitions is about $nst \times (m \times (pmnum + k) + n \times k + C_1 \times pmnum)$. With regard to elementary transitions, in the worst case, there is no elementary transition, that is, $N = nst$, so $nst \times K$ times passages of loops are required to execute, to accomplish the computations.

We conclude that the time complexity of Algorithm 1 is $O(nst \times (m \times (pmnum + k) + n \times k + C_1 \times pmnum + K))$. If the number of input places of subnets is small, it means that $pmnum$ is small, then the complexity is approximated as $O(nst \times ((m + n) \times k + K))$. Further we suppose that the size of the maximal subnet does not change, it means m, n, and k are constant, and only the size of the HSN is increasing, and the time complexity is evaluated as $O(nst \times K)$. Furthermore, we found that, when the same subnets are repeatedly used, such as calculating the sum or product of a series of random numbers, or multiplying matrices, values of K and nst maintain a linear relationship, that is, $K = C_2 \times nst$, where C_2 is a constant. Thus the time complexity is finally approximated as $O(nst^2)$. Since we do not use intermediate structures which exceed the size of the resulting LSN, we estimate the space complexity as $O(nst)$.

7. Tests and benchmarks

We use benchmarks as the basic indicator for efficient algorithms and corresponding data structures design. The SN VM implementation has been accelerated using CPU multicore and GPU mass parallel facilities. We were using selected examples of SN programs as tests and also we developed dedicated software generators to obtain big nets for benchmarks. The following HSN program generators have been developed: sum of an array, product of an array, computing a polynomial, matrix multiplication. We were using also the generator of net for computing double exponent described in [36]. Having NDR and HSN source files allowed us to test the entire toolset including *NDRtoSN* and *HSNtoLSN* as well.

7.1. *SN programs for benchmarks*

To obtain a series of SNs for benchmarks, we use ad-hoc software generators of nets. For prolonged run of a net on SN VM, we use double exponent exact computer that implements Lipton's [44] net amended with four priority arcs described in [36]; the net can be considered as a busy beaver [43] for place-transition nets. It is generated in NDR format and then transformed by *NDRtoSN* into LSN format to run on SN VM. The generator *depnz* inputs parameter n to produce a net computing 2^{2^n}.

Similar technique is applied to compose dedicated generators of HSN files to test both programs *HSNtoLSN* and *SN-VM*. The following generators of SN programs have been developed:

- *gen_vadd* for summing up a vector (a sequence of additions generalizing Fig. 5);
- *gen_pol* for computing value of a polynomial;
- *gen_mmul* for matrix multiplication (with two variants for parallel and sequential implementation).

To specify the corresponding nets we use prefixes "de", "vadd", "mmul", and "pol" followed by the parameter value.

An example of SN that implements multiplication of 2×2 matrices is represented in Fig. 20. The picture was obtained manually using *nd* editor. The nerator produces HSN files of similar structure (though without graphical layout).

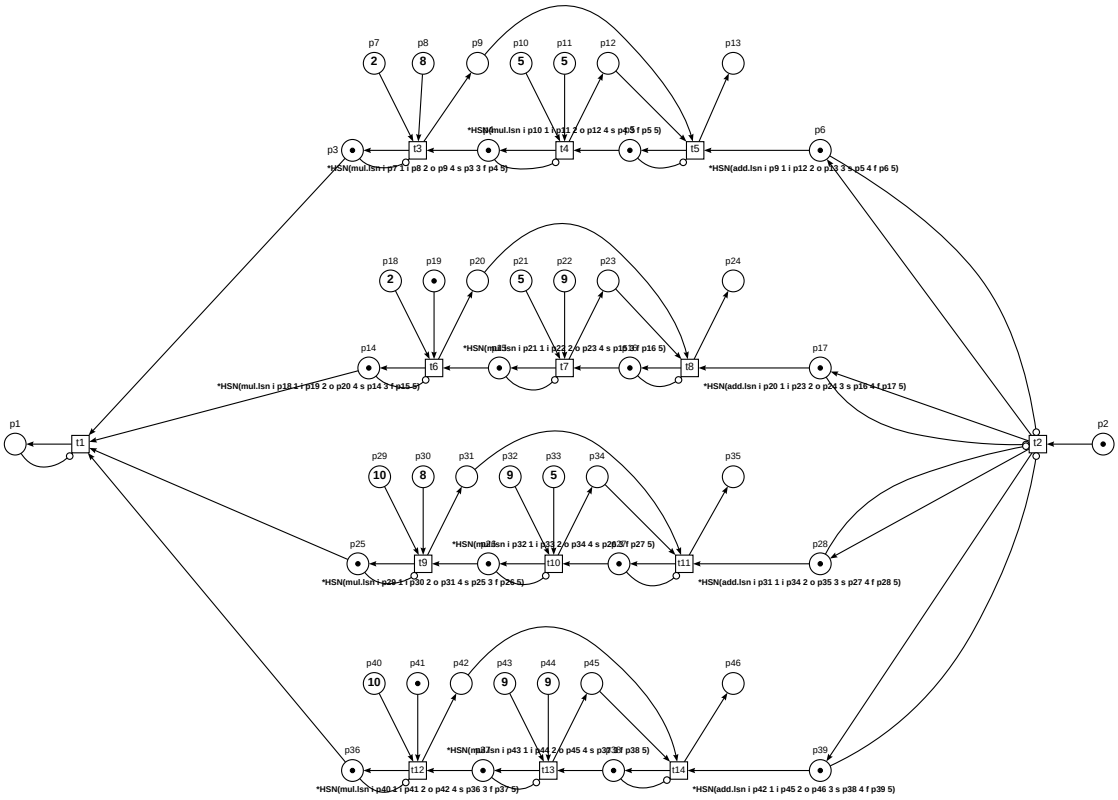


Fig. 20. HSN for multiplying 2x2 matrices that uses ADD and MUL as components.

Preliminary generated SNs, which we use for benchmarks, are collected in Table 5

together with detailed specification of their size.

Table 5. Specification of selected models for benchmarks.

Notation	HSN				LSN		
	places	transitions	arcs	substitutions	places	transitions	arcs
de3	-	-	-	-	68	53	281
de4	-	-	-	-	90	72	382
pol10	197	65	390	65	1657	1820	7065
pol15	407	135	810	135	3497	3855	14985
pol20	692	230	1380	230	6012	6640	25830
mmul3	155	47	300	45	1055	1118	4323
mmul4	370	114	723	112	2578	2738	10563
mmul5	727	227	1428	225	5127	5452	21003

7.2. *Benchmarks of SN VM*

To obtain benchmarks, we were using models specified in Table [**?**], and also the addition chain and the multiplication chain with priority arc. The addition and multiplication chains, same as *pol* and *mmul*, produced by the corresponding generator, represent HSN files. After converting the HSN files into LSN files, the LSN files are run on the SN VM. To run programs, we use a laptop computer, Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz (4 cores). SN VM is started from a command line that specifies input and output file names, and options. The input file specifies an SN that contains m, n, k places, transitions, and arcs, respectively, and the output file stores the output data representing the result of an LSN run. The "-d" option specifies the debug level: level 0 outputs final marking only; level also prints matrices of incoming and outgoing arcs, and the matrix of the priority arc; level 2 shows detailed information about the net run. The "-nth" option specifies the number of threads.

Table 6 contains the benchmarks of the mentioned SN programs' run on 1, 2, 4, and 8 threads, the corresponding diagram for selected models, shown in Fig. 21, reveals rather good speed-up, exceeding 3 times on 8 cores.

Table 6. Benchmarks on SN programs (1-8 threads) in seconds.

SN:Threads	1	2	4	8
de3	0.637s	1.722s	2.091s	3.040s
de4	262.335s	573.406s	644.255s	1107.798s
pol10	38.220s	23.122s	19.409s	18.703s
pol15	553.777s	363.434s	255.844s	192.493s
pol20	3540.0801s	2327.307s	1486.017s	1155.436s
mmul3	17.277s	11.943s	11.000s	7.614s
mmul4	222.941s	152.328s	102.127s	92.814s
mmul5	2246.331s	1374.648s	1164.548s	736.581s

We should mention also an anomaly with busy beaver nets, when *SN-VM* per-

formance slows down with the increasing number of threads that requires futher investigation.

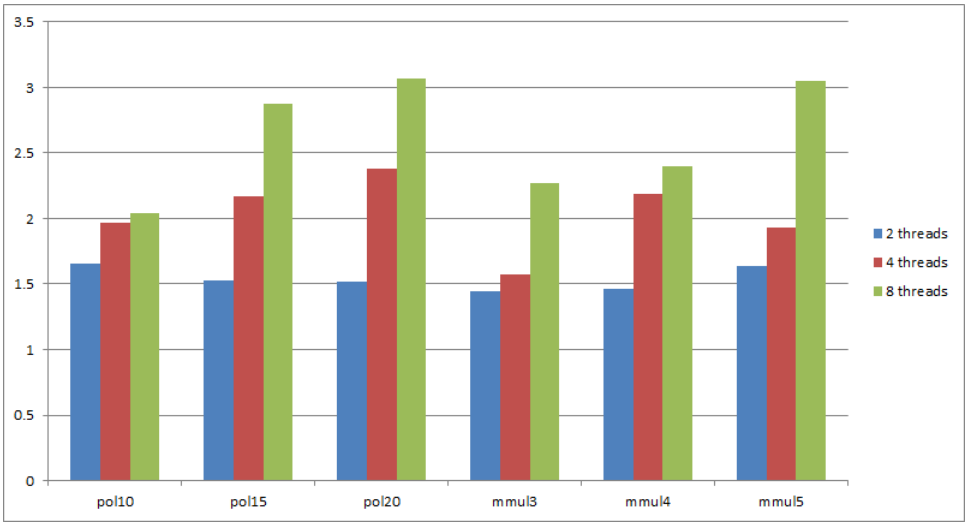


Fig. 21. Diagram of SN-VM benchmarks in the number of speed-ups on multi-core compared with a single thread.

For GPU SN VM accelerator, that uses a sequence of kernel calls at a step because of compatibility with CUDA architecture 35, there was no considerable speed-up of computations achieved. The best obtained speed-up does not exceed two times when using NVIDIA GeForce 920M GPU with 2GB of video memory.

7.3. *Benchmarks of SN CL*

To test the efficiency of SN compiler-linker, besides nets, shown in Table 5, we composed dedicated generators to produce HSN files of various size and run these nets on SN CL to obtain the corresponding LSNs. We were using the number of transition substitutions as an independent variable and the running time as a dependent variable to perform quadratic polynomial fitting (according to the CL time complexity evaluations obtained in Section 6) because all tests conform the case where m, n and k are constants.

We list the three following series of tests, which results and fitting curves are represented in Fig. 22:

(1) the two-parameter adder as the transition substitute subnet, in which case HSNs of various sizes actually represent the sum of random numbers (add);
(2) the two-parameter multiplier as the transition substitute subnet, in which case HSNs of various sizes actually represent the product of random numbers (mul);

(3) the two-parameter adder and the two-parameter multiplier as the transition substitute subnet that represents the product of randon square matrices of various dimensions (matrix).

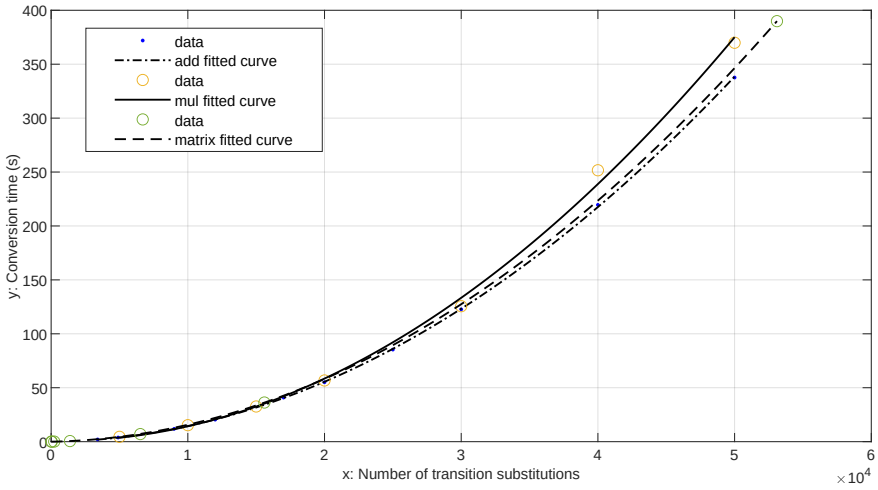


Fig. 22. Quadratic polynomial fitting of CL time complexity on tests: sum of random numbers (add fitted curve); product of random numbers (mul fitted curve); multiplication of random square matrices (matrix fitted curve).

The corresponding sum of squares approximation parameters, shown in Table 7, confirms rather good fitting into the obtained in Section 6, quadratic timed complexity evaluation of SN compiler-linker work.

Table 7. Sum of square approximation parameters for SN CL tests.

Curve	Error	R-square
add fitted curve	0.00245	0.9958
mul fitted curve	0.002012	0.9978
matrix fitted curve	0.001696	0.9946

We consider prospects of using ParSEC framework [45] to further speed-up SN CL work on multi-core architecture.

8. Conclusions

In this paper, the first open source implementation of Sleptsov net computing integrated developer environment and virtual machine, including GPU accelerator, has been presented. To integrate the toolset, represented by SN Virtual Machines,

Compiler-Linker, and converter of file formats, graphical editor nd of modeling system Tina has been chosen because of its simplicity, convenience, and friendly GUI.

A series of software generators of SN programs, computing the sum of vector, value of a polynomial, results of matrix multiplication, have been developed and applied for tests and benchmarks. For prolonged test runs, a busy beaver like nets, computing a double exponent after Lipton's design, have been generated and executed.

The presented prototype implementation proves the robustness of SNC and opens prospects for its enterprise-level implementation, especially including dedicated hardware design.

References

References

[1] Sleptsov Net Computing Resolves Modern Supercomputing Problems, The April 21, 2023, edition of ACM TechNews, https://technews.acm.org/archives.cfm?fo=2023-04-apr/apr-21-2023.html

[2] Dmitry Zaitsev (2023) Sleptsov Net Computing resolves problems of modern supercomputing revealed by Jack Dongarra in his Turing Award talk in November 2022, International Journal of Parallel, Emergent and Distributed Systems, https://doi.org/10.1080/17445760.2023.2201002

[3] Dmitry A. Zaitsev, Strong Sleptsov nets are Turing complete, Information Sciences, Volume 621, 2023, 172-182

[4] Zaitsev D.A. Sleptsov Nets Run Fast, IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2016, Vol. 46, No. 5, 682–693.

[5] Jack Dongarra, "A Not So Simple Matter of Software," ACM A.M. Turing Award Lecture, the 2022 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC22), November 8 , 2022, https://youtu.be/lsnRP9akCDk

[6] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, Version 2.3, December 2, 2018 https://netlib.org/benchmark/hpl

[7] LINPACK, https://netlib.org/linpack

[8] Top500. The list. https://top500.org

[9] The High Performance Conjugate Gradients (HPCG) Benchmark. https://www.hpcg-benchmark.org

[10] Fugaku, RIKEN Center for Computational Science, https://www.r-ccs.riken.jp/en/fugaku/.

[11] Zaitsev D.A. Solving operative management tasks of a discrete manufacture via Petri net models. PhD thesis. Kiev, the Academy of sciences of Ukraine, Institute of Cybernetics name of V.M.Glushkov, 1991. http://daze.ho.ua/daze-phd-1991.pdf

[12] Zaitsev D.A., Sleptsov A.I. State equations and equivalent transformations for timed Petri nets, Cybernetics and Systems Analysis, Volume 33, Number 5 (1997), 659-672.

[13] Zaitsev D.A. Enterprise Petri net based CAM software Opera-Topaz, Proc. of 3rd International Industrial Simulation Conference 2005 (ISC 2005), June 9-11, 2005 - IPK Fraunhofer Institute, Berlin, Germany, pp. 124-128.

[14] G. Paun, M. Ionescu, T. Yokomori, Spiking Neural P Systems with an Exhaustive Use of Rules, Int. J. Unconv. Comput. 3 (2007) 135–153.

[15] L. Qian, E. Winfree, Scaling up digital circuit computation with DNA strand dis-

placement cascades, Science 332 (6034) (2011) 1196–1201.
[16] Zaitsev D.A. Toward the Minimal Universal Petri Net, IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2014, Vol. 44, No. 1, 47-58.
[17] Zaitsev D.A., Jürjens J. Programming in the Sleptsov net language for systems control, Advances in Mechanical Engineering, 2016, Vol. 8(4), 1-11.
[18] Zaitsev D.A. Universal Sleptsov Net, International Journal of Computer Mathematics, 94(12) 2017, 2396-2408.
[19] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 2005.
[20] Bernard Berthomieu, François Vernadat, Silvano dal Zilio, Tina: TIme petri Net Analyzer, `https://projects.laas.fr/tina/index.php`
[21] Dmitry A. Zaitsev, NDRtoSN: Converter of Tina NDR Petri net file to Sleptsov Net file, `https://github.com/dazeorgacm/NDRtoSN`
[22] Qing Zhang, SN-VM: Sleptsov net Virtual Machine, `https://github.com/zhangq9919`
[23] Tatiana R. Shmeleva, SN-VM-GPU: Sleptsov Net Virtual Machine on Graphics Processing Unit, `https://github.com/tishtri/SN-VM-GPU`
[24] Hongfei Zhao, HSNtoLSN: Compiler-linker of hierarchical Sleptsov net program, `https://github.com/HfZhao1998/Compiler-and-Linker-of-Sleptsov-net-Program`
[25] Dmitry A. Zaitsev and David E. Probert (2021) Preface for special issue Petri/Sleptsov net based technology of programming for parallel, emergent and distributed systems, International Journal of Parallel, Emergent and Distributed Systems, 36:6, 495-497.
[26] Zaitsev D.A. Paradigm of Computations on the Petri Nets, Automation and Remote Control, 2014, Vol. 75, No. 8, 1369-1383.
[27] Tiplea FL, Diaconu RA. Petri net computers and workflow nets. IEEE Trans Syst Man Cyber Syst. 2015;45(3):496-507.
[28] Alexander A. Kostikov, Nikolay D. Zaitsev, Oleg V. Subotin (2021) Realisation of the double sweep method by using a Sleptsov net, International Journal of Parallel, Emergent and Distributed Systems, 36:6, 516-534.
[29] Tatiana R. Shmeleva, Jan W. Owsiński, Abdulmalik Ahmad Lawan (2021) Deep learning on Sleptsov nets, International Journal of Parallel, Emergent and Distributed Systems, 36:6, 535-548.
[30] Li, Z.W. and Zhou, M.C. (2009) Deadlock Resolution in Automated Manufacturing Systems: A Novel Petri Net Approach. Springer, London.
[31] YuFeng Chen, ZhiWu Li, Kamel Barkaoui,Maximally permissive liveness-enforcing supervisor with lowest implementation cost for flexible manufacturing systems, Information Sciences, Volume 256, 2014,74-90.
[32] Y. Hou, D. Liu and M. Zhou, "On iterative liveness-enforcement for a class of generalized Petri nets," 2012 IEEE International Conference on Automation Science and Engineering (CASE), Seoul, Korea (South), 2012, pp. 188-193.
[33] Qing Zhang, Ding Liu, Yifan Hou, Sleptsov Net Processor, International Conference "Problems of Infocommunications. Science and Technology" (PICST2022), 10-12 October, 2022, Kyiv, Ukraine.
[34] Hongfei Zhao, Ding Liu, Yifan Hou, Compiler and Linker of Sleptsov Net Program, International Conference "Problems of Infocommunications. Science and Technology" (PICST2022), 10-12 October, 2022, Kyiv, Ukraine.
[35] David B. Kirk, Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 3rd Edition, Morgan Kaufmann, 2016.

[36] Dmitry A. Zaitsev and MengChu Zhou, From strong to exact Petri net computers, International Journal of Parallel, Emergent and Distributed Systems, 37(2), 2022, 167-186.

[37] B. Berthomieu, D. Le Botlan, S. Dal Zilio, Counting Petri net markings from reduction equations. International Journal on Software Tools for Technology Transfer, 22(2), 163-181, 2020.

[38] Zaitsev D.A. Clans of Petri Nets: Verification of protocols and performance evaluation of networks, LAP LAMBERT Academic Publishing, 2013, 292 p.

[39] Dmitry Zaitsev, Stanimire Tomov, Jack Dongarra. Solving Linear Diophantine Systems on Parallel Architectures, IEEE Transactions on Parallel and Distributed Systems, 30(5), 2019, 1158-1169.

[40] Barbara Chapman, Gabriele Jost and Ruud van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, MIT Press, 2007.

[41] Vel'bitsky, I.V., Kovalev, A.L., Kasatkina, I.V. et al. R-technology of programming: Basic notions and implementation. J. of Comput. Sci. & Technol. 7, 345–355 (1992).

[42] Douglas Thain, Introduction to Compilers and Language Design, 2nd edition, 2020, ISBN: 979-8-655-18026-0.

[43] Pascal Michel. The Busy Beaver Competition: a historical survey. 2017, hal-00396880v6.

[44] Lipton RJ. The reachability problem requires exponential space. Tech. Rep., 1976. [Online]. Available from: `http://www.cs.yale.edu/publications/techreports/tr63.pdf`

[45] Abdulah, S., Q. Cao, Y. Pei, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief, and Y. Sun, "Accelerating Geostatistical Modeling and Prediction With Mixed-Precision Computations: A High-Productivity Approach With PaRSEC," IEEE Transactions on Parallel and Distributed Systems, vol. 33, issue 4, pp. 964 - 976, April 2022.