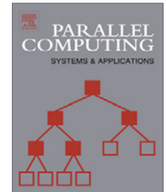
Parallel Computing 36 (2010) 591–617

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



Fast bio-inspired computation using a GPU-based systemic computer

Marjan Rouhipour [a], Peter J. Bentley [b,*], Hooman Shayani [b]

[a] BIHE University (The Bahá'í Institute for Higher Education), Iran
[b] Department of Computer Science, University College London, Malet Place, London WC1E 6BT, UK

ARTICLE INFO

Article history:
Available online 4 August 2010

Keywords:
Bio-inspired computation
Systemic computation
GPU
Parallel architectures
Genetic algorithm

ABSTRACT

Biology is inherently parallel. Models of biological systems and bio-inspired algorithms also share this parallelism, although most are simulated on serial computers. Previous work created the systemic computer – a new model of computation designed to exploit many natural properties observed in biological systems, including parallelism. The approach has been proven through two existing implementations and many biological models and visualizations. However to date the systemic computer implementations have all been sequential simulations that do not exploit the true potential of the model. In this paper the first ever parallel implementation of systemic computation is introduced. The GPU Systemic Computation Architecture is the first implementation that enables parallel systemic computation by exploiting the multiple cores available in graphics processors. Comparisons with the serial implementation when running two programs at different scales show that as the number of systems increases, the parallel architecture is several hundred times faster than the existing implementations, making it feasible to investigate systemic models of more complex biological systems.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

In biological modeling and bio-inspired computation, the demand for fast parallel computation has never been greater. In computer science, entire fields now exist that are based purely on the tenets of simulation and modelling of biological processes (e.g., Computational Neuroscience [1], Artificial Life [2], Computational Biology [3]). In the developing fields of synthetic biology, DNA computing, and living technology, computer modelling plays a vital role in the design, testing and evaluation of almost every stage of the research,[1] e.g., [4,5]. There are also many fields of computer science that focuses on bio-inspired algorithms such as genetic algorithms [6], artificial immune systems [7], developmental algorithms [8], neural networks [9], swarm intelligence [10].

Almost without exception these computer models and algorithms involve parallelism, although they are usually implemented as serial simulations of parallel processes. While multi-core processors, clusters or networked computers provide one way to parallelise the computation, the underlying computer architectures remain serial, and so can significantly limit our ability to scale up our models and bio-inspired algorithms and make them practical for real-world problems [11].

In an attempt to exploit desirable natural properties such as parallelism within a computer,[2] in 2005 a novel model of computation called *systemic computation* was developed [11]. The result of considerable research into bio-inspired

* Corresponding author. Tel.: +44 0 20 7679 1329.
E-mail addresses: marjan.rouhipour@bihe.org (M. Rouhipour), p.bentley@cs.ucl.ac.uk (P.J. Bentley), h.shayani@cs.ucl.ac.uk (H. Shayani).

[1] Evident from the many publications of the European Center for Living Technology: http://www.ecltech.org/publications.html.

[2] Other features include being robust, fault-tolerant, distributed, stochastic, homeostatic, continuous, open-ended, approximate, asynchronous, embodied, complex with circular causality.

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2010.07.004

computation and biological modelling, the model has been developed into a working computer architecture [11,12]. A parallel systemic computer based on this architecture is designed to run on hundreds or preferably thousands of processors, with all computation emerging through interactions, just as the overall result of biological processes emerges through the interaction of thousands of simpler elements. A systemic computer should share some of the same capabilities of biological systems and provide fault-tolerant computation through self-organising, parallel and distributed processing.

To date, two simulations of this architecture have been developed, with corresponding machine and programming languages, compilers and graphical visualiser [11,12]. Extensive work has shown how this form of computer enables useful biological modeling and bio-inspired algorithms to be implemented with ease [13–17] and how it enables properties such as fault-tolerance and self-repairing code [18]. Research is ongoing in the improvement of the PC-based simulator, refining the systemic computation language and visualiser [19,20]. However, the systemic computation model defines a highly parallel, distributed form of computer. While simulations on conventional computers enable the improvement of the model and associated programming tools, the speed of simulated systemic computation is too slow to be useable for larger models. The work described in this article aims to overcome this problem by making use of graphics processing units (GPUs) to parallelize some of the bottlenecks in systemic computation and thus take the first steps towards a fully parallel systemic computer, capable of high-speed biological modeling.

Graphic Processing Units are multi-core processors designed to process graphical information at high speed. Because of their price and power, the use of GPUs for more general-purpose computation is rapidly becoming something of a revolution in affordable parallel computation [21]. More recently the design of GPUs was changed to support more general computation. Today many are programmable to support user-defined software.

There are many programming languages for GPUs and as the design of GPUs has been improved, programming languages have been changed. The first generation of programming languages for computation on GPUs were shading languages such as Cg and HLSL that execute a graphical API such as OpenGL, DirectX explicitly [22]. They were used for complex computation, but they were insufficiently flexible or generic for general-purpose computation [22]. The programmer had to know about many hardware-specific details such as the number of shader outputs, shared instruction count, texture size, etc. [22]. To provide a more portable interface for different GPUs, a new generation of high-level languages, known as GPGPU languages were created. OpenCL, C for CUDA, Brook, Scout, Accelerator, Stream SDK, and CGis are good examples of these languages.

Today GPGPU languages are used widely in scientific computation. They are used in physical based simulation, signal and image processing, global illumination, and geometric computing [21]. GPGPU is frequently used in biological modeling and visualization that requires large-scale computation and real-time processing. For example they have been used in molecular modeling applications [23], string matching to find similar protein and gene sequences [24], and in implementations of bio-inspired algorithms [25].

This work describes a novel GPU-based implementation of the bio-inspired computing approach known as systemic computation. The next section summarizes systemic computation. A general overview of GPU architecture is then provided and the new GPU Systemic Computation Architecture is described in detail, followed by a series of experiments, which compare the GPU version with the single-processor implementation.

2. Systemic computation

"Systemics" is a world-view where traditional reductionist approaches are supplemented by holistic, system-level analysis of the interplay between components and their environment at many different levels of abstractions [26]. *Systemic Computation* takes this approach [11,12], aiming to provide a more "natural" or bio-inspired model of computation compared to conventional von Neumann architectures. It uses the following assertions:

- Everything is a *system*.
- Systems can be transformed but never destroyed.
- Systems may comprise or share other nested systems (see Fig. 1).
- Systems *interact*, and interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems and systems may have partial membership within the scope of a system.
- Computation is transformation.

Systemic computation has been shown to be Turing Complete [11] and thus is directly equivalent to other models of computation. For example, it can also be regarded as a form of bigraph model [11,27]. Although the origins of systemic computation come from studies of natural systems, each system may be viewed as equivalent to an asynchronous bigraph node, and schemata (see later) may be viewed as dynamic links between nodes. Because of this, systemic computation may be rewritten in bigraph form (and also may benefit from expression in π-calculus [27,28]) for future theoretical investigations.

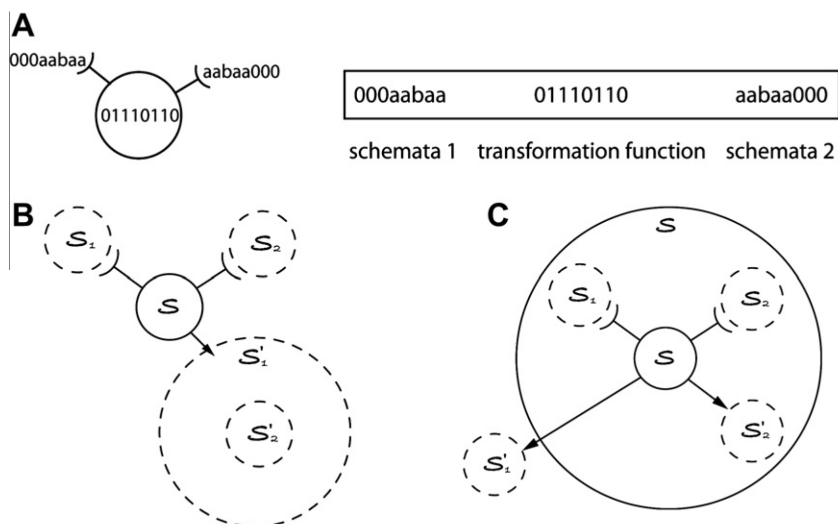


Fig. 1. Systemic computing relies on the concept of a system to perform all computation. A system comprises three elements: two schemata and a transformation function. Systems may be defined in memory as strings; they are graphically depicted as a circle surrounding the transformation function, with two "cups" or receptors representing the two schemata (A). The two schemata of a system define which other systems may match and hence be affected by this system. The transformation function of a system defines how two schemata matching systems are changed when they interact with each other in the context of this system; arrows indicate transformed systems at time $t + 1$. A system may be pushed inside the scope of a second system through interaction with it (B). A system within the scope of a larger system may be pushed outside that scope through interaction with another system (C). Computation occurs by transforming input from the environment into systems, which interact with "organism" systems (equivalent to software or hardware) and potentially each other to produce new systems that provide output by stimulating or altering the environment.

In addition, some useful systemic computation transformation functions resemble those widely used in membrane computing and brane calculus [29], while in approach, it is comparable to the ideas of cellular automata [30] (but influenced by the views of Varela et al. [31]).

Instead of the traditional centralised view of computation, in systemic computation all computation is distributed. There is no separation of data and code, or functionality into memory, ALU, and I/O. Everything in systemic computation is composed of *systems*, which may not be destroyed, but may transform each other through their interactions, akin to collision-based computing [32]. Two systems interact in the context of a third system, which defines the result of their interaction. This is intended to mirror all conceivable natural processes, e.g.:

- molecular interactions (two molecules interact according to their shape, within a specific molecular and physical environment),
- cellular interactions (intercellular communication and physical forces imposed between two cells occurs in the context of a specific cellular environment),
- individual interactions (evolution relies on two individuals interacting at the right time and context, both to create offspring and to cause selection pressure through death).

Systems have some form of "shape" (i.e., distinguishing properties and attributes and may encompass anything from morphology to spatial position) that determines which other systems they can interact with, and the nature of that interaction. The "shape" of a contextual system affects the result of the interaction between systems in its context. This encompasses the general concept that a resultant transformation of two interacting systems is dependent on the context in which that interaction takes place. A different context will produce a different transformation. Since everything in systemic computation is a system, context must be defined by a system.

In order to represent these notions computationally, the notions of schemata and transformation functions are used. The "shape" of a system in this model is the combination of schemata and function, so specific regions of that "shape" determine the meaning and effect of the system when behaving as a context or interacting. Thus, each system comprises three elements: two schemata that define the possible systems that may interact in the context of the current system, and the transformation function, which defines how the two interacting systems will be transformed (Fig. 1A).

Systemic computation also exploits the concept of *scope*. In all interacting systems in the natural world, interactions have a limited range or scope, beyond which two systems can no longer interact (for example, binding forces of atoms, chemical gradients of proteins, physical distance between physically interacting individuals). In cellular automata this is defined by a fixed number of neighbors for each cell. Here, the idea is made more flexible and realistic by enabling the scope of interactions to be defined and altered by another system. Interactions between two systems may result in one system being placed within the scope of another (akin to the *pino* membrane computing operation [29]) see Fig. 1B, or being removed from the

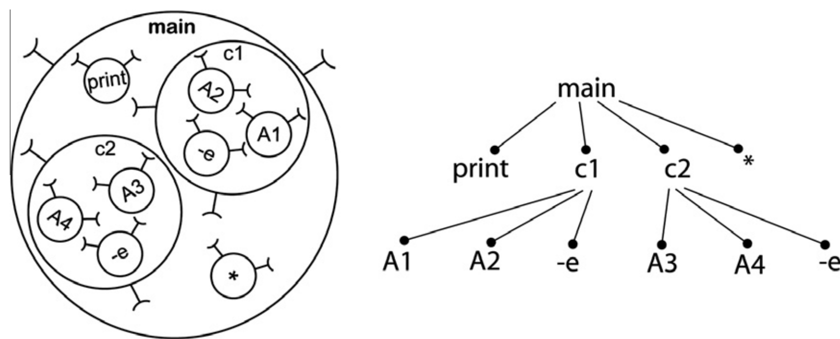


Fig. 2. Systemic computation calculation: PRINT ((A1-A2)*(A3-A4)). Left: graph-based notation. Right: the tree of scope memberships for this calculation. (Systemic computation also permits networks of memberships that may be difficult to draw, e.g., "print" could be placed in the scope of "A1" in addition to "main", without being in the scope of "c1".)

scope of another (akin to the *exo* membrane computing operation [29]), see Fig. 1C. So just as two systems interact according to (in the context of) a third system, so their ability to interact is defined by the scope they are all in (defined by a fourth system). Scope is designed to be infinitely recursive so systems may contain systems containing systems and so on (by default, scopes contain themselves).[3] Scopes may overlap or have fuzzy boundaries; any systems can be wholly or partially contained within the scopes of any other systems, including themselves. Scope also makes this form of computation tractable in simulation by reducing the number of interactions possible between systems to those in the same scope.

Most systemic computation forms a complex interwoven structure made from systems that affect and are affected by their environment. This resembles a molecule, a cell, an organism or a population, depending on the level of abstraction used in the program. Fig. 2 illustrates the organization of a real systemic computation, showing the use of structure enabled by scopes.

Systems can be implemented using representations similar to those used in genetic algorithms and cellular automata. In implementations to date, each system comprises three binary strings: two schemata that define sub-patterns of the two matching systems and one coded pointer to a transformation function. Two systems that match the schemata have their own binary strings transformed according to the appropriate transformation function (e.g., an arithmetic or logical operation). A simple example of a (partially interpreted) system string (where $S1_1$ is the first schema and $S1_2$ is the second schema of system 1) might be:

"zzx00rzz [$S1_1$ = SUM($S1_1$,$S2_1$); $S1_2$ = SUM ($S1_2$,$S2_2$); $S2_1$ = 0; $S2_2$ = 0]   zzx00rzz"

meaning: for every two systems that have functional part of NOP that interact in the context of this system, add their two S_1 values, storing the result in S_1 of the first system and add the two S_2 values, storing the result in S_2 of the first system, then set S_1 and S_2 of the second system to zero. (The table of schema codes is given in Fig. 3.) Given a pool of inert data systems, able to interact but with no ability to act as context, for example (where NOP means "no operation"):

"00010111 NOP 01101011" and "00001111 NOP 00010111"

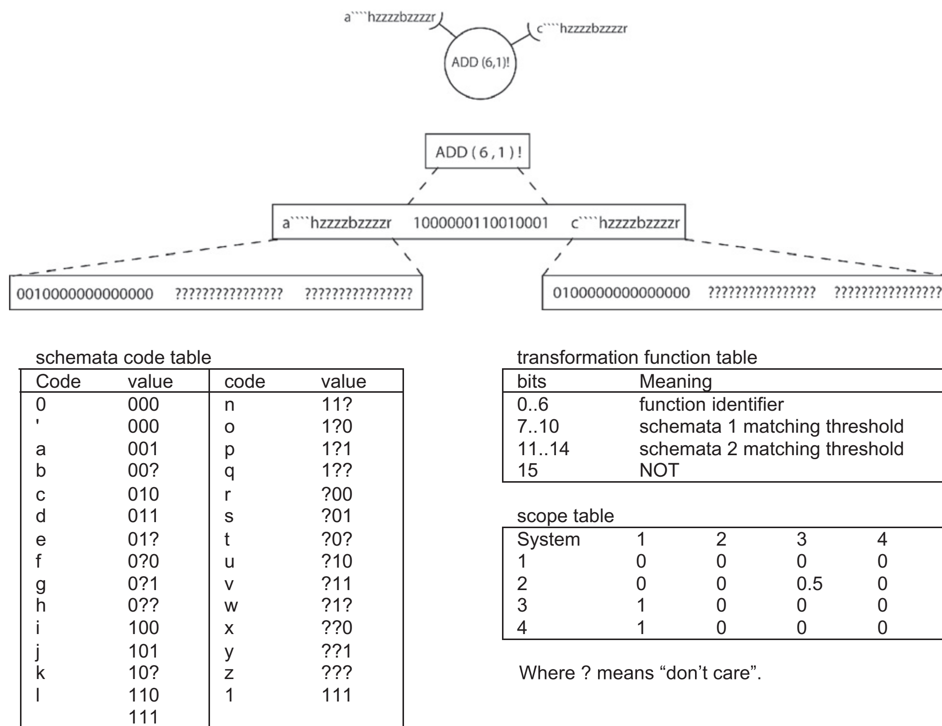
after a sufficient period of interaction, the result will be a single system with its S_1 and S_2 values equal to the sum of all S_1 and S_2 values of all data systems, with all other data systems having S_1 and S_2 values of zero. (The program performing this operation was described in [11].)

Systems within the same scope are currently presented to each other randomly just as most interactions in the natural world have a stochastic element. The computation proceeds asynchronously and in parallel, distributed amongst all the separate systems, structurally coupled to its environment, with parallelism and embodiment providing the same kind of speed-up seen in biological systems. Computation is continuous (and open-ended) with homeostasis of different systems maintaining the program.

Bentley [11] describes the first implementation of the systemic computer, summarized here. The initial work included the creation of a virtual architecture, instruction set, machine code and corresponding assembly language with compiler. These enable systemic computation programs to be simulated using conventional computer processors. Four implementation-specific features enable systemic computers to be tailored to a given application:

1. the word-length/coding method
2. the transformation function set/schemata matching method
3. the order of system interactions and
4. the scope definition method.

[3] Note the concept of one system containing another, or being partially or fully with the scope of another is akin to set theory in mathematics. It does not imply a physical container or physical containment in terms of architecture; it implies a restriction of interactions (which may result from physical arrangements that cause reductions to the probability of interactions).



schemata code table

Code	value	code	value
0	000	n	11?
'	000	o	1?0
a	001	p	1?1
b	00?	q	1??
c	010	r	?00
d	011	s	?01
e	01?	t	?0?
f	0?0	u	?10
g	0?1	v	?11
h	0??	w	?1?
i	100	x	??0
j	101	y	??1
k	10?	z	???
l	110	1	111
	111		

transformation function table

bits	Meaning
0..6	function identifier
7..10	schemata 1 matching threshold
11..14	schemata 2 matching threshold
15	NOT

scope table

System	1	2	3	4
1	0	0	0	0
2	0	0	0.5	0
3	1	0	0	0
4	1	0	0	0

Fig. 3. Graphical representation of one system with function "ADD" (top). Method of coding used in systems to enable two 16-character schemata to define 48-character systems to be matched, and to enable one 16-character transformation function to define function, matching thresholds and NOT operator (middle). Schemata code table (bottom left) defines the codes used in schemata, the transformation function table (middle right) gives the meaning of the 16-bit number. A scope table is given (bottom right), indicating non-fuzzy scopes where systems 3 and 4 are completely within system 1, and fuzzy scope where system 2 is half within system 3.

In the implementation described in [11], (i) schemata and transformation functions were defined by strings of 16 characters of alphabet 29, resulting in each system being 48 characters long[4]; (ii) some thirty transformation functions were implemented, using partial matching against thresholds; (iii) system interactions occur randomly except where a system is changed, in which case changed systems are chosen for subsequent interaction first; (iv) scopes were held globally in a system scope table.

Integers were coded using pure binary coding. A wildcard (defined by any character not a 1 or 0) enabled the schemata of systems to define partial matches. Triplets of system string characters were coded using ASCII values 96 to 122 (characters: ' to z), Fig. 3. In this way a 16-character schemata string was used to define a 48-character system, enabling each system to define the two (types of) systems it could transform with complete precision. Two matching threshold values enabled the accuracy of the required match to be defined, with a NOT operator inverting the matching requirements (i.e., systems had to match the schemata correctly for the function *not* to be carried out). Threshold values and NOT operators are considered to be included within the transformation function. Scopes were implemented globally in a scope table with column entries defining which of the systems were contained within the systems listed in the rows (Fig. 3). The numeric value of each entry defined the degree of membership of each system in the parent, enabling partial or fuzzy membership. Hamming distance [33] was used to calculate the difference between schemata and systems, and compared against the function threshold values (which are provided in the functional part of the schemata with the pointer to the transformation function).

Since this virtual machine was running on a sequential computer, the parallelism and asynchrony were simulated. The procedure determining the order in which system interactions takes place in a systemic computer is equivalent to the fetch-execute cycle of a conventional computer. In this implementation, interaction order was random (context and two interacting systems determined by pseudo-random number generator seeded by current time). Systems were held as a fixed numbered array in memory; as systems are never lost, this array structure and order is never modified. The use of a global scope table enables systems to be moved between scopes using negligible computation time and zero change to system ordering in memory.

This simulation was implemented in ANSI C on a PowerBook Macintosh G4. An assembly language and corresponding compiler was created. Over 30 transformation functions were implemented (e.g., arithmetic and logical operations, and basic

[4] The word-length is user-definable; here 16 characters was chosen.

i/o). Later work by Le Martelot created a second implementation on PCs with a higher-level language and visualization tools [12,14–18,20]. Other work provided a discussion on the use of sensor networks to implement a systemic computer [19]. Many systemic computation models have been written, showing that simulations of this parallel computer can perform tasks from investigations of neurogenesis to a self-adaptive genetic algorithm solving a travelling salesman problem [16]. Work on the language and refinements to systemic computation and its use for modeling are underway. However, perhaps the biggest single problem with all implementations to date has been the speed of execution. The simulation of a parallel bio-inspired computer on a conventional serial computer can be excessively slow for models using large numbers of systems. For this reason, we hypothesize that an implementation on GPUs may provide significant speedup.

3. GPU implementation of systemic computation

3.1. GPU architecture

In the last few years we have begun to reach the limits of CPU clock speeds. While improvement can be achieved by adding new sequential processor cores, the underlying architecture means that there are limits to the number of cores that can be added. However multi-core or many-core devices, which have larger numbers of processor cores, are necessary for parallel programming to increase performance. One solution is to modify the architecture, as can be seen in GPUs. The difference between CPU and GPU design is shown in Fig. 4. Most modern CPUs have a small number of cores and large cache memory; in contrast, newer GPUs have many processors and small cache. CPUs execute instructions in a thread out of sequence but preserve sequential interface [21]. However, GPUs are suitable for large numbers of threads to run in parallel on shared data

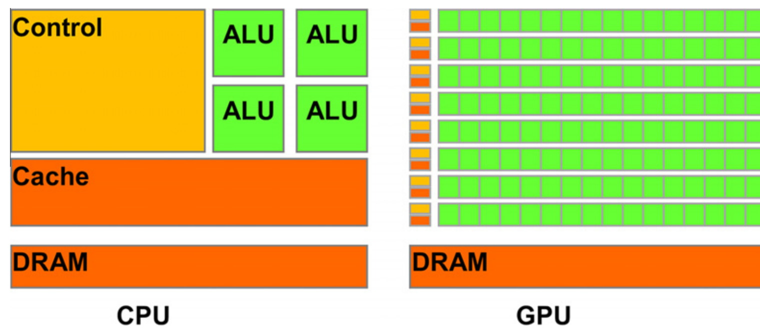


Fig. 4. Difference in GPU and CPU design [34].

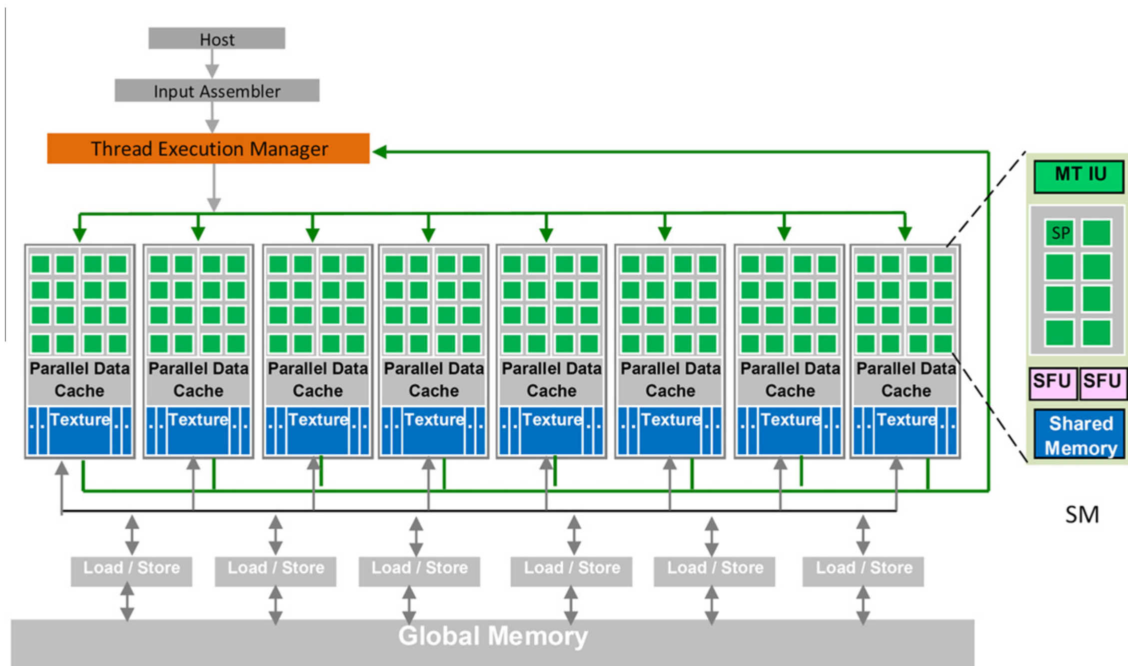


Fig. 5. Architecture of a CUDA-capable GPU [35].

[21]. They are good for floating-point data and since the computing processing on GPU has become popular, GPU architecture have been changed to support IEEE floating-point standard and processing of non-graphical computations [21].

As shown in Fig. 5, the new GPU structure has many stream multiprocessors (SM), with each SM having some stream processors (SP) or cores, which are used for computational purposes. In the figure a CUDA-capable GPU block structure is shown. Each SM has eight SPs, two special function units (SFU) for transcendental functions, a small shared memory as cache, a register file, and some other regions for fetch and execute instructions. Two SMs share a texture memory, used for video image and 3D rendering [35].

The CPU part is called the host and the cores in the GPU are called devices. The program that is run on a device is called the kernel. The program that is run on the host is responsible for transferring data between device memory and host memory. The GPU architecture is SIMT (single instruction multiple thread) that is similar to SIMD of Flynn's taxonomy [36] and it can manage threads even when their control flow is different. CUDA has its own thread hierarchy for managing threads: block and grid. Each thread runs on a SP. A group of 32 threads in a block that manages and run together on a SM is called a warp. Some warps form a block of threads that is run on a SM. All thread blocks of a kernel are known as the grid. Global memory is shared between all threads in grid, shared memory between all threads in a block and registers only for one thread. The number of warps that can run at the same time on a SM is limited and depends on hardware architecture limitation and resources: the number of registered and shared memory used by threads in one warp [34].

3.2. GPU systemic computation architecture

Systemic computation is a Turing Complete parallel computer [11], but it is still a significant challenge to exploit the parallel architecture of the GPU to implement such a flexible bio-inspired approach.

The constituent elements of Systemic Computation are systems. Two systems can only interact with a context and within a shared scope, which are also systems. Akin to Bentley's implementation [11], membership of scopes can be implemented as a global scope table, a row and column for every system, with each entry defining the membership of one system within another system. Also akin to Bentley's original implementation [11], in the GPU design we can implement the concept of a system by storing it in three binary parts: two schemata and one function. If the current system is acting as a context, then its two schemata define all possible pairs of systems that could interact within the current context. The system function defines the interaction between the two systems, i.e., it provides a transformation function. In the implementation created by Bentley [11], the transformation function contains a matching threshold for schemata 1 and 2. The length of each part of the system is 16 character codes. Each code is decoded to three characters (0, 1, and wildcard) as was shown in Fig. 3. If the difference between a system and the decoded schema's part is less than the schema's threshold, that system matches the context system's schema [11].

We call a system that has a valid transformation function, a 'context' or 'functional' system. A context system and two interacting systems together are called a triplet. If two interacting systems can match the schema parts of a functional system to form a triplet, and all of them are in the same scope, it is defined as a matched or valid triplet.

The GPU Systemic Computation Architecture has two main tasks: (1) finding a valid triplet, and (2) performing a transformation to the interacting systems [11]. In the sequential implementation of the architecture, a matched triplet must first be found and then transformation is performed [11]. We can therefore consider the two main parts of Systemic Computation in parallel. As shown in Fig. 6, one part produces a list of matched triplets and the other part consumes the triplets. The finding of valid triplets is the biggest bottleneck in systemic computation, so in our design, the *producer* finds matched triplets and puts them in a shared buffer while the *consumer* picks one of triplets and performs the interaction between them. The producer and consumer are two threads, running in parallel on the CPU. In the following section we explain details of each thread's task shown in Fig. 6.

3.2.1. Consumer: performing system interactions

The consumer is a thread running on the CPU. It is responsible for enabling the interactions between systems. This thread selects valid triplets (each a valid context and two matching systems) randomly from the shared buffer. It then uses the transformation function defined in the context system to transform the pair of interacting systems. However, performing the transformation may change the systems' scopes and definitions. As a result, other triplets in shared memory that share the one of the current triplet's systems may no longer match (i.e., the transformation of one triplet may invalidate other triplets). In order to solve this problem there are two solutions to handle this conflict between triplets: (1) delete conflicting triplets before adding them to the shared buffer and (2) check validity of triplets before performing interaction. If the first solution is followed, just one of the conflicting triplets is added to the shared buffer (i.e., no triplets which share interacting systems or interacting and context systems can be added to buffer). In addition, many triplets may also be removed from triplet list before adding them to the shared buffer. However, most of the time interaction does not change systems in a way that invalidates other triples. Also, selecting one triplet between triplets with shared systems is itself time-consuming and difficult. Therefore, the second solution was deemed better. As shown in Algorithm 1 and Fig. 6 (right), after selecting a triplet, if the triplet is still a matched triplet, the transformation is performed. Then the flags of systems definitions and scopes are set. These flags are necessary to update data on GPU memory for other thread, *producer*. As both threads access the memory used for two flags and change it, they have to access them in a critical section. This thread is also responsible for counting the number of interactions.

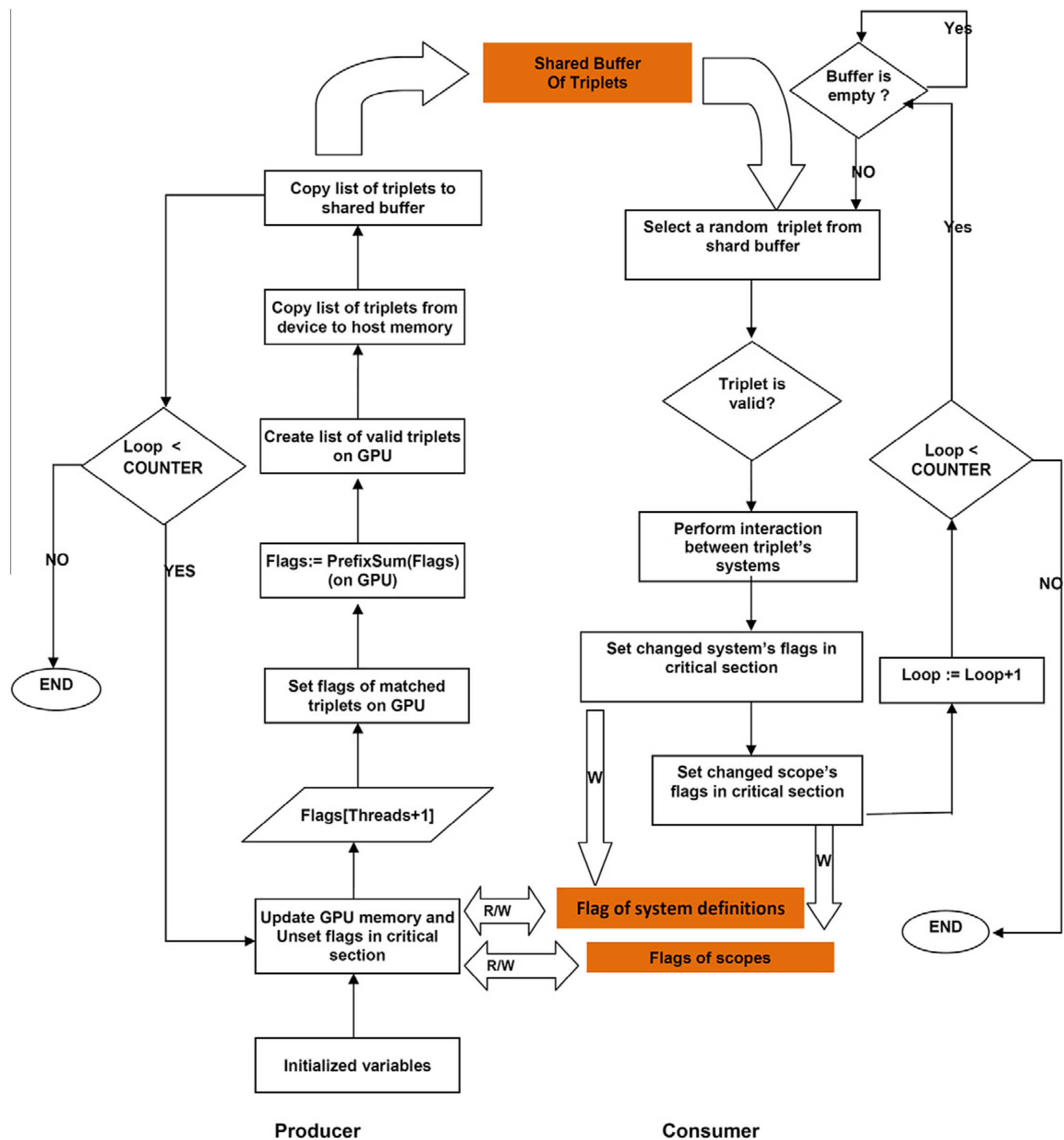


Fig. 6. Producer and consumer flowchart.

Algorithm 1. Perform transformation on systems on CPU

---

1. **Begin**
2.
3. `INITIALIZE CPU variable`
4. `Loop := 0 //interaction counter`
5.
6. **WHILE** `(loop < COUNTER)`
7. **Begin**
8. `Triplet triplet`
9. `select a random triplet from shared buffer and put in`
10. `triplet`
11. `//check validity of triplet`

Algorithm 1 (*continued*)

```
12.          IF triplet's system are matched
13.          THEN
14.              Perform transformation on triplet systems
15.              IF any system of triplet is changed set its flag
16.              in critical section
17.              IF any system's scope is changed set its flag
18.              in critical section
19.
20.              Loop++
21.          ENDIF
22.      ENDWHILE
23.      END
```

3.2.2. Producer: finding matching systems

General purpose graphic processing unit (GPGPU) languages are based on shared memory [34]. In the GPU Systemic Computation Architecture, systems are shared data and the instructions that check validity of triplets in a scope are the same for all threads. So, CUDA is a good choice to find matched triplets in parallel. The number of threads on a GPU is equal to all combinations of context systems, scopes, and all systems that could behave as one of the interacting systems. Each thread is responsible for checking a triplet in one scope. However, the problem is how the index of two interacting systems, a context, and a scope are assigned to a thread. To solve this problem, we used CUDA's thread hierarchy [34]: block and grid.

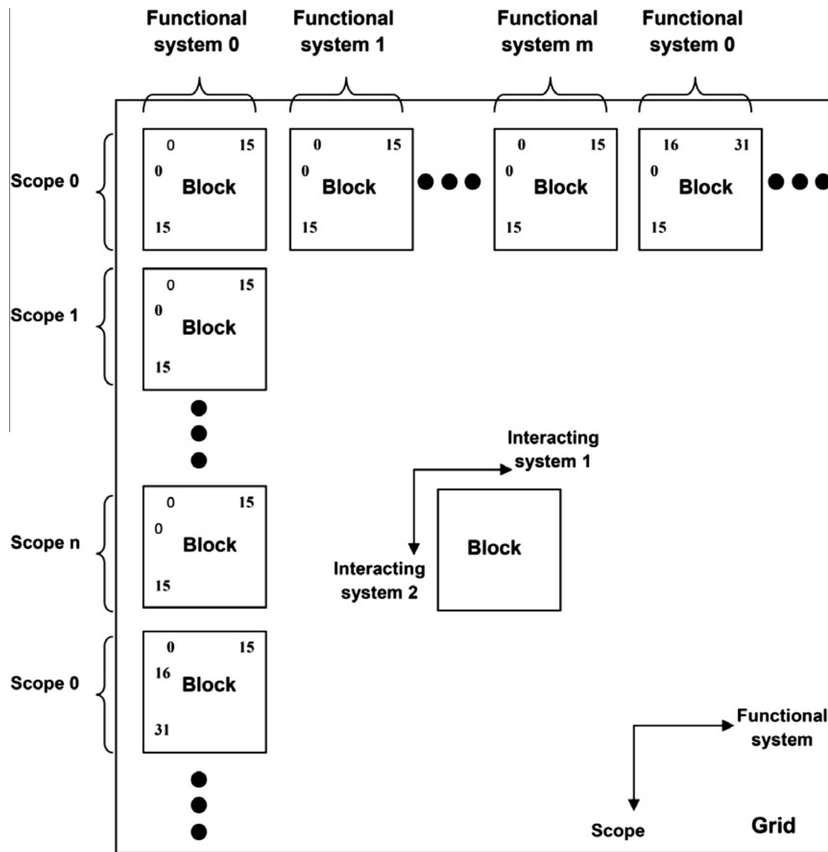


Fig. 7. The outer rectangular is a grid. One dimension of the grid is assigned to scopes and the other is assigned to functional (context) systems. The block dimension also is assigned to interacting systems. Each block examines 16 systems as interacting system 1, and 16 systems as interacting system 2.

Each thread can have two block and two grid dimensions. As shown in Fig. 7, the *X* and *Y* dimensions of blocks identify the index of schemata 1 and 2 systems. The *Gx* and *Gy* dimensions of a grid identify the index of the scope and context systems. Each block therefore processes only the interacting systems under one context and scope. In addition, each context or scope may have many number of blocks that depend on block size and number of systems. The size of *Bx* and *By* block dimensions is set as 16. In other words, there are 256 threads in one block. This selection is based on warp size, the maximum number of threads in a block (for example 768 threads for devices with computation capability 1.0 and 1.1), and limited memory resources and registers in each block [34]. The last factor is related to kernel implementations, as the code runs on GPU. The block size should be a multiple of warp size and not more than maximum block size. The two dimensions of blocks are both 16 because the number of both interacting systems as schemata 1 and 2 are equal. So the number of threads may be more than needed. The number of blocks *B* for each scope or context is:

$$B = Ceil\ (number\ of\ systems/16),$$

where *Ceil* is the ceiling function. The Grid dimensions are:

$$Gx = scp \times B$$

and

$$Gy = func \times B,$$

where *scp* is the number of scopes and *func* is the number of functional systems.

As can be seen in Algorithm 2, finding a list of matched triplets is both a sequential and parallel procedure. There are six main steps: initializing, updating, finding matched triplet, prefix sum, creating a list of matched triplets, and copying them to the shared buffer. These steps are run sequentially on the CPU. The third, fourth, and fifth steps are the main parts that are run on GPU. Each of these steps is a kernel, a section of code that is run on the GPU. As only one kernel can be run on the GPU in CUDA, and the output of each step is an input of the next step, these steps run sequentially on the CPU, but individually parallel on the GPU. Here, we mainly focus on kernels.

Algorithm 2. This algorithm shows sequential steps of finding matched triplets. Triplets are checked, matched triplets found and copied to the shared buffer by a thread on the CPU

```
1.   BEGIN
2.
3.   INITIALIZE CPU and GPU variable
4.
5.   WHILE (loop < COUNTER)
6.   BEGIN
7.       UPDATE CPU and GPU variables
8.
9.       CALCULATE number of threads
10.
11.      ALLOCATE MEMORY for d_flags[threads + 1] and
12.      d_triplets[threads]
13.      SET d_flags to zero
14.
15.      // this kernel set flags appropriate triplet in
16.      // d_triplets
17.      CALL Find_Matched_triplet_kernel (d_flags,d_triplets)
18.
19.      d_flags := prefixsum (d_flags)
20.
21.      size := d_flags[threads]//number of valid triplet
22.      ALLOCATE MEMORY for d_validTriplet[size] and
23.               triplets[size]
24.      // make a list of vaid triplet in d_validTriplets
25.      CALL make_Triplet_list_kernel (d_flags, d_triplets,
26.               d_validTriplet)
29.      copy d_validTrplets to triplets
30.      Randomize triplets
31.      COPY triplets to shared_buffer
32.  ENDWHILE
33.  END
```

STEP 1: Initialize

During the initialize step, memory is allocated on the GPU for different variables: system definitions, scope table and decoded systems (including the two decoded schemata and threshold function).

STEP 2: Update

The other thread that performs transformation functions changes the scope table and system definitions, and so the producer thread always updates variables on GPU before checking all possible combination of triplets. Then new values of variables are copied from the host, (a part of the hardware that is on the CPU's processing part) to the device, (part of hardware that is on the GPU's part for processing). In addition, functional systems and valid scopes (scopes with equal or greater than three systems inside and at least one functional system) are found and copied to the GPU Memory. For finding differences between the schemata part and the system, the schemata is decoded; therefore, a list of decoded functional systems is prepared on the device and updated after being changed by the consumer thread. In summary, some parts of the GPU memory is devoted to the scope table, the list of system definitions, index of valid scopes, index of functional systems, and decoded systems. (In addition, temporary memory is used for updating variables, but this is deallocated; memory that is used in the following steps is greater than these temporary memories.)

STEP 3: Finding matched triplets

In this step a kernel is called that searches through all possible triplets in order to find matched triplets. As shown in Algorithm 2, line 11, before calling the kernel, memory is allocated for the list of flags. Each thread has a flag that is initialized to zero (line 13). Next, the grid and block dimensions are set (as explained above).

As can be seen in Algorithm 3, all threads in a block check one scope and context for some interacting systems. Thus, one thread in each block, usually the first one, calculates the index of context and scope systems and stores it in the shared memory of the block; meanwhile, other threads in a block wait (lines 9–16). If the context is in the scope, the index of interacting systems is then calculated. After that if interacting systems are in the scope and three systems are matched, the triplet's flag is set to one.

Algorithm 3. How all combinations of triplets are examined and their flags are set on the GPU

```
1.      Do all thread in parallel
2.      INPUT:
3.      d_flags, d_triplets
4.
5.      // shared data in one block:
6.      Shared Variable:
7.          scope, context, // index of scope and context
8.
9.      IF this is first thread in a block
10.     THEN
11.         CALCULATE scope by using grid dimensions
12.         CALCULATE context by using grid dimensions
13.
14.     ENDIF
15.
16.     WAIT for all threads in one block to reach this point
17.
18.     IF context is in scope
19.     THEN
20.         CALCULATE schemata1 and schemata2 by using block and
21.                         grid dimensions
22.
23.         IF schemata1and schemata2 are in scope and
24.             Schemata1 and schemta2 and context are not
25.             equal
26.         THEN
27.           // find differences
```

(continued on next page)

Algorithm 3 (*continued*)

```
28.
29.                 diff1 := difference (schemtal,
30.                       context.schematal)
31.                 diff2 := difference (schemta2,
32.                       context.schemata2)
33.
34.                    // check matching
35.                 IF diff1 and diff2 is less than context
36.                   Thresholds
37.                 THEN
38.                    d_flags[threadID] := 1
39.
40.                 ENDIF
41.           ENDIF
42.        ENDIF
```

Optimization: The global memory of the GPU is long-latency, off-chip memory and is accessible by all threads in a grid [34]. However, each block has its own short-latency, on-chip memory (shared memory) that is smaller than global memory [34]. Therefore only the first thread reads the membership value of context in current scope from the scope table on global memory, storing the result in shared memory; enabling other threads to read the result from shared memory. Because the decoded context system is the same for all threads, it is read by the first thread and loaded to shared memory if necessary. Interacting system definitions and their membership value in the current scope are read multiple times by a thread in the block; so, the first 16 threads in a block load them from the global memory to shared memory, if the context system is in the scope. Finally, threads read value from shared memory.

STEP 4: Prefix sum

In this step we want to create a list of parallel matched triplets, whose flags have been set in the previous step. In order to do so, the index of matched triplets in the new list have to be found. The prefix sum kernel is run on the flags to find indexes of matched triplets. The prefix sum calculates number of previously matched triplets in the current list for each matched triplet. A parallel implementation of this algorithm is available in CUDA SDK 2.2.[5] The current implementation of this algorithm is only run in one grid dimension, but we have changed it to two dimensions to support a larger size array, if sufficient memory is available.

STEP 5: Creating list of triplets

The kernel for creating lists of matched triplets is run with the same grid's and block's dimensions of the found matched triplets kernel. As shown in Algorithm 4 the first thread calculates the block and context, stores in the shared memory (lines 12–16). Then threads read two subsequent memories of prefix sum flags. If a thread identified two different values, the thread's dimensions indicate indexes of matched triplet's systems and scope. Here each flag memory is read twice, for optimization the first 16 threads of each block load the 17 subsequent memory of flags to shared memory. This helps to reduce the number of reading from global memory to half.

Algorithm 4. How the list of matched triplets is created with a kernel on the GPU, by using flags that are set in previous steps

```
1.       Do thread in parallel
2.       INPUT:
3.       d_Indexes // prefix sum of d_flags
4.
5.
6.       validTriplets // output list
7.
8.       //shared data in one block:
```

[5] Available online at: http://www.nvidia.com/object/cuda_get.html.

Algorithm 4 (*continued*)

```
9.     SHARED VARIANBLEs:
10.        scope, context, // index of scope and context
11.
12.    IF This is first thread in a block
13.    THEN
14.        CALCULATE scope and context by using grid
15.        dimensions
16.    ENDIF
17.    WAIT until all threads reach this point
18.
19.    IFd_Indexes [threadId] !=
20.    d_Indexes [threadId + 1]
21.    THEN
22.        CALCULATE schematal and schemta2
23.        by using block and grid dimensions
24.        validTriplets[d_Indexes [threaId]] :=
25.            Triplet (scope,context,schematal,schemata2)
26.    ENDIF
```

STEP 6: Copy matched triplet list to buffer

In this step, a list of matched triplets is copied to the host. It is then randomized and copied to the shared buffer. For a large number of systems, it is not possible to process all combination of triplets at once, because there is not enough memory for flags and list of matched triplets on the GPU. So each time step, a subset of all possible triplets is chosen randomly and processed, a list of matched triplets of the subset triplets is created. After all possible triplets checks, variables update the GPU memory and the second round starts.

4. Architecture testing and evaluation

In order to assess the new implementation, we need to run systemic programs on the original architecture created by Bentley [11] and the new GPU Systemic Computation Architecture, and compare the results. (Note that it is not the purpose of this paper to compare traditional or non-systemic computation with the GPU architecture – here we focus on the performance gain for systemic computation with and without parallelism.) In the following section we outline two problems that are implemented in the Systemic Computation language. The first is the natural phenomenon of chemical diffusion, and the second is the knapsack problem implemented using a genetic algorithm (GA). These problems are specifically designed to challenge the systemic computer in different ways – the first allows us to investigate the effects of frequently moving systems between many different scopes; the second investigates a more stable arrangement of systems but more complex parallel computation.[6]

4.1. Problem 1: chemical diffusion

The diffusion of chemicals (Brownian motion) is a natural phenomenon. In this model, we assume that space is one-dimensional, circular and divided into spatial regions called cells. Cells act as scopes, which limit the interactions within their local regions. "Water" and "ink" molecules are systems that can move or "jump" between neighboring cells. The details of these systems' definitions are shown in Figs. 8 and 9.

As shown in Fig. 10, neighboring cells are defined by overlapping their scopes like a chain (the two neighbors of every system are within the scope of that system). To enable the left and right neighbors of each cell to be distinguishable, cells are given types (types A, B and C) where every triplet of neighboring cells are of different types. In order to initialize ink and water systems in cells, an *Initialiser* system is defined, see Fig. 11. On first starting the program, the ink and water molecules are in the main scope. They are selected randomly and initialized by the Initialiser system. All the ink systems are placed into a single randomly selected cell; all water systems are placed into the other cells, see Fig. 8. Therefore, two initialiser systems are necessary for the two types of molecules.

[6] Exactly how each program is implemented will also affect the performance – see [11–20] for details of how a systemic analysis can be performed to help create the best systemic model of a biological system. The two programs here are used purely to test the relative speed of the two systemic computers and are not intended to be optimal implementations for these algorithms on any computer.

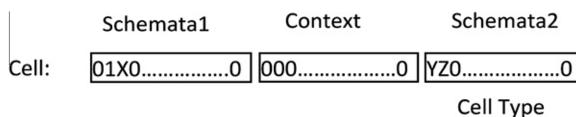


Fig. 8. Cell system definition. Ink molecules are put in one cell during initialization and **X** indicate this special cell: 0 for cell_ink or and 1 for cell_water. **YZ** indicates the three types of cell: A: 01, B: 10, and C:11.

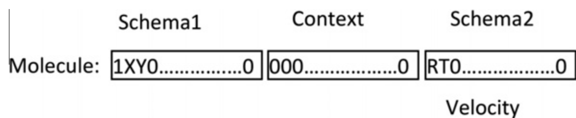


Fig. 9. A molecule system. **X** defines the type of molecule: 0 for ink and 1 for water. **Y** indicates whether the system is initialized (=1) or not initialized (=0). RT shows velocity that could be 10 (= +1 for right) and 11 (=−1 for left).

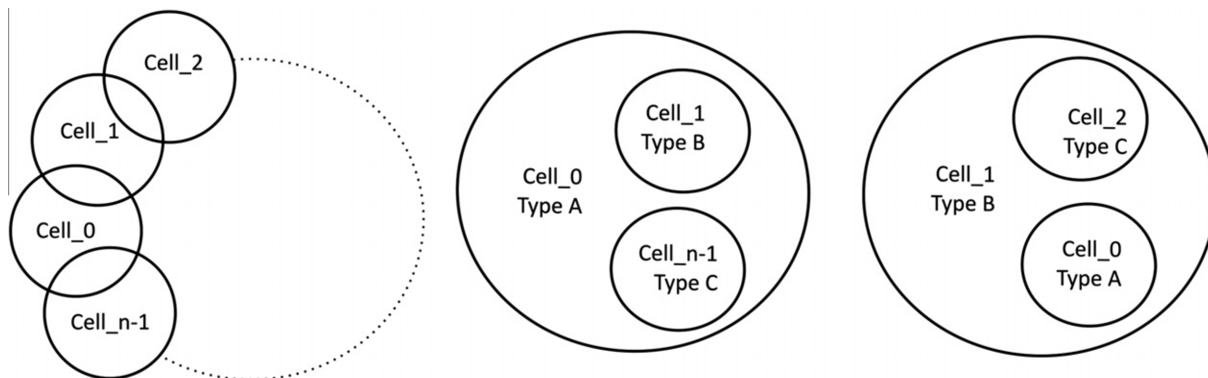


Fig. 10. How neighboring cells (spatial regions) are identified and organised. Left: there are *n* cells arranged in a circle; each cell has two neighbors; every pair of neighboring cells are within the scope of the middle cell. Every three cells adjacent to each other are of different types to allow each to be distinguished from each other. For example, the center image shows that the two neighbors of cell_0 are within its scope, with cell_0 being type A, cell_1 being type B and cell_n-1 being type C. The rightmost image shows that the two neighbors of cell_1 are within its scope, with cell_1 being type B, cell_2 being type C and cell_0 being type A.

Each water or ink system has a velocity, +1 or −1. The molecule's velocity shows speed and direction of the move (−1: jump to the left cell, +1: jump to the right cell). Therefore, there are six kinds of *Jump* functions that move ink and water systems in two directions, and three kinds of cells. One *Jump* system (jump to right and cell of type B) is shown in Fig. 12. The *Jump* context accepts an initialized molecule and one neighbor cell; the result of their interaction is that the molecule is moved into the neighboring cell of type B. After performing the jump, the molecule is not in the current cell any more. Two *Jump* systems are placed in each cell to enable molecules to circulate in either direction through the chain of cells. The direction of movement is effected by *Impact* – another context system. As shown in Fig. 13 (left), the *Impact* system accepts two molecule systems (e.g., an ink and water system) with different velocities. When these two systems interact in the context of the *Impact* system, their velocities are swapped, see Fig. 13 right. There are two kinds of *Impact* systems for two ink and water systems with different velocities in each cell. In addition, there is a main scope that contains all the systems inside. The definition of functional systems in more detail and the SC code for the chemical diffusion program is provided in Appendix A.

4.2. Problem 2: genetic algorithm optimization of binary knapsack

In the knapsack problem there are n objects with value $v_i > 0$ and weight $w_i > 0$. We want to find a set of objects with maximum total weight that fits into a knapsack with capacity C. Thus, we wish to maximize:

$$\sum_{i=1}^{n} v_i x_i \quad \text{where} \ \sum_{i=1}^{n} w_i x_i \leqslant C \ \text{and} \ x_i \in \{0,1\}.$$

Here, we use a genetic algorithm (GA) [6] as a bio-inspired algorithm to implement the optimization program. The main idea of the binary knapsack implementation in the Systemic Computation language is derived from the genetic algorithm model

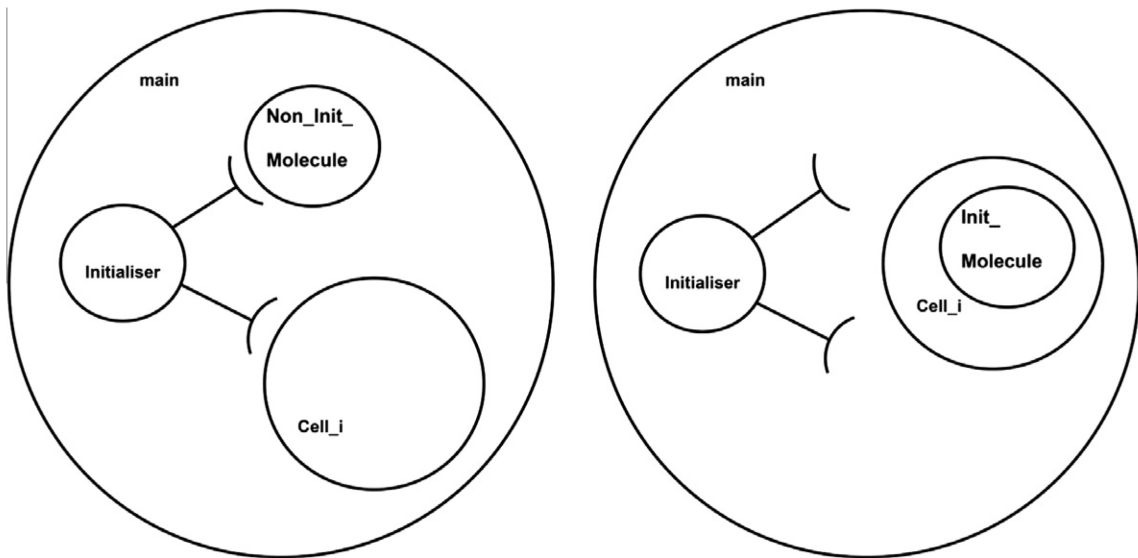


Fig. 11. The behavior of the **Initialiser** context system. **Cell_i** indicates the **i**th cell. **Non_Init_Molecule** shows a non initialized molecule. The left image shows how the Initialiser context affects the interaction of an uninitialized molecule and a cell. The right image shows the result after the interaction in the context of the Initialiser: the molecule's velocity is set and it moves inside cell_i.

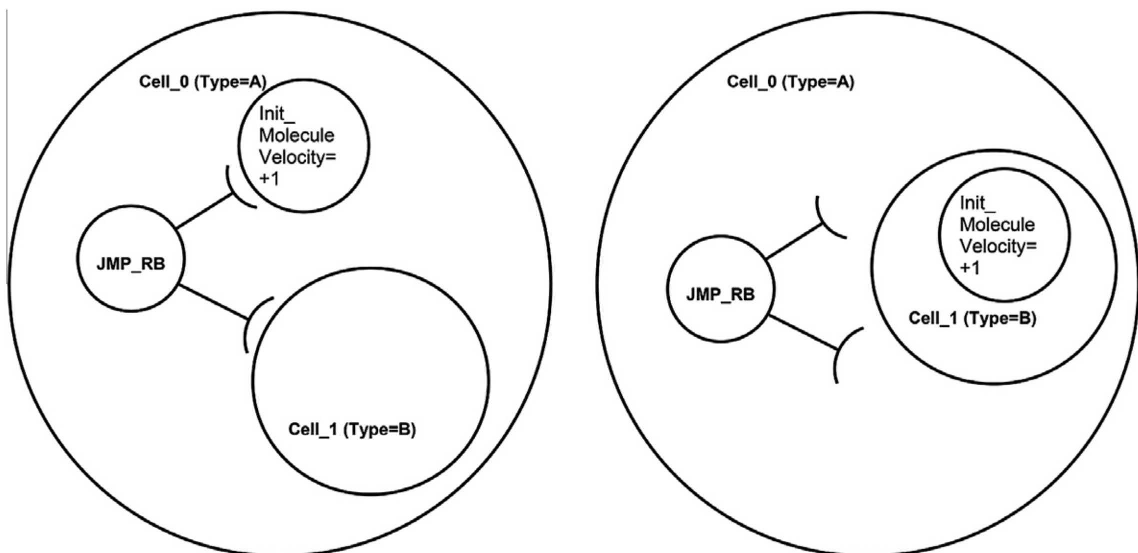


Fig. 12. The behavior of the jump context system. **Init_Molecule** is an initialized molecule with a velocity of +1. **JMP_RB** is the context system, which causes a jump right from the current cell into a neighboring cell of type B. The context is triggered when a molecule with velocity = +1 interacts with a neighboring cell of type B. The left images shows the systems before interaction. The right image shows how the systems have been affected after performing the jump.

developed for systemic computation in [12]. There are three different solutions, as shown in Fig. 14: uninitialized solutions, initialized solutions, and final solutions. The limitation of this problem is the chromosome size that can only be as long as the schemata size (=16 in this implementation). So, this program only supports knapsack with 16 objects.

Initializing solutions is done randomly. An Initialiser system selects one solution and initializes it randomly then puts it in the Computation scope, see Fig. 15. Three GA operators are used: uniform crossover, one-point crossover and binary mutation [6]. As shown in Fig. 16, candidate (evolving) solutions to the problem are implemented as systems, and each GA operator is implemented as a context system, which affects the results of two solution systems interacting. Each operator

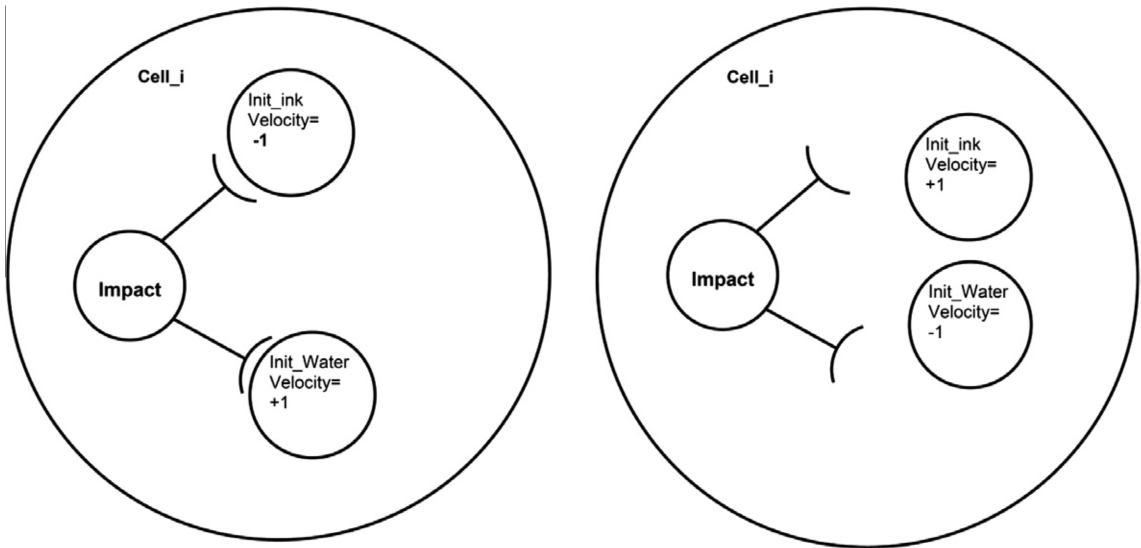


Fig. 13. An ink and water system interacting in the context of **Impact**. **Init_water** and **Init_ink** are initialized water or ink molecules. The left images shows how the **Impact** context affects the interaction of two different molecules (water or ink) with different velocity as inputs. The right image shows the result after the interaction in this context: after performing the impact function on the two molecules, their velocities change.

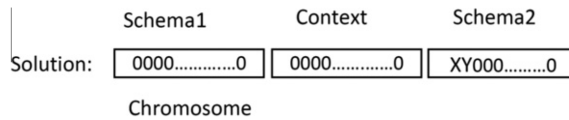


Fig. 14. The **Solution** system's details. The first part of the system is where the chromosome is stored. It can be at most as long as the schemata size. There are three different types of solution type, indicated by **XY** in the third part of the system (00: non initialized solution, 10: initialized solution, 11: final solution).

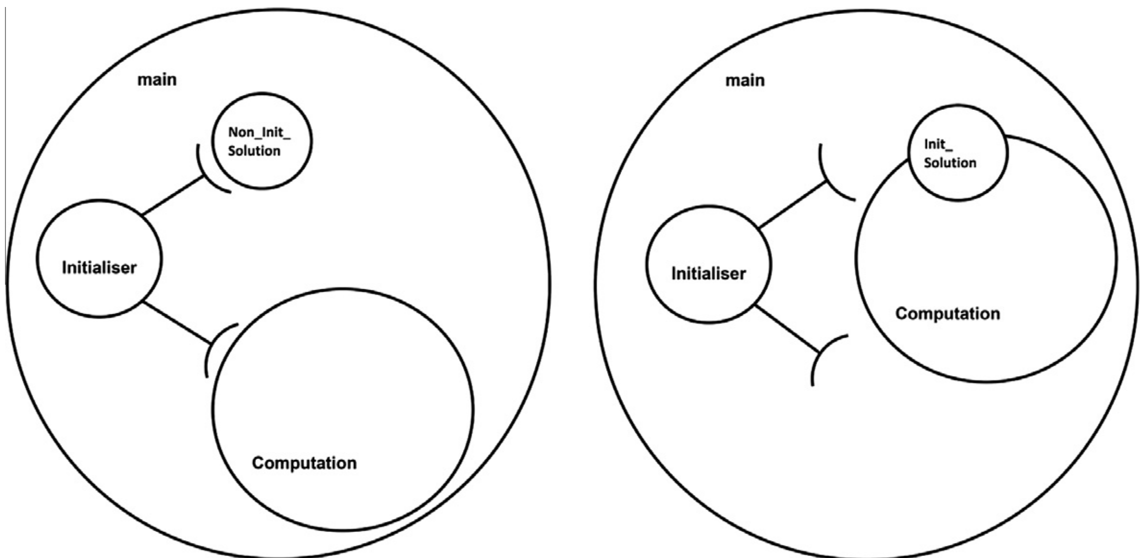


Fig. 15. The **Initialiser** context system. **Non_Init_Solution** is a solution that is not initialized yet. **Init_Solution** is an initialized solution. **Computation** is the scope for the genetic algorithm functions. The left image shows the interaction of a non-initialized solution and the computation scope in the context of the Initialiser system. The right images shows the result of the interaction: the Initialiser initializes the solution and moves it into the Computation scope (it also remains in the context of the main scope).

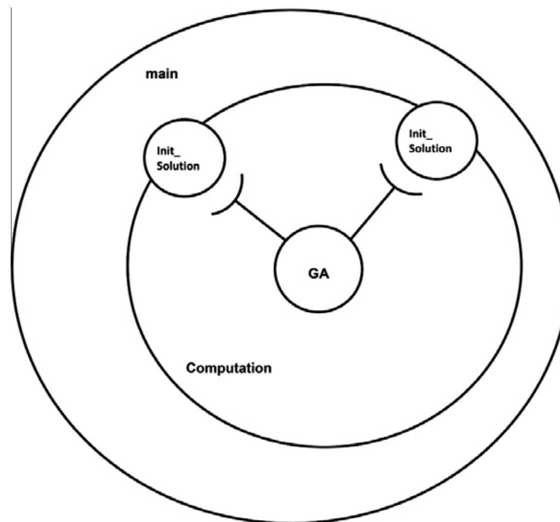


Fig. 16. The **GA** systems implements the GA operators: crossover or mutation. **Init_Solution** are initialized solutions. Initialized solutions are placed in both the computation and main scopes. Initialized solution systems interact in the context of GA operations in Computation scope.

performs crossover or mutation, to generate a new solution, this is evaluated and then the less fit solution is replaced by the new one. The Final Solution system is used to store the final chromosome. As shown in Fig. 17, the Output context accepts the Final solution and one Initialized solution, and updates the Final Solution if the input Solution system has better total value than Final Solution system.

The details of functional system definitions and the sample of this program in SC language is available in Appendix B.

4.3. Experiments

To assess any performance gain of the GPU Systemic Computation Architecture, comparisons are made with Bentley's original serial implementation [11], running the chemical diffusion and GA knapsack programs. We study the effect of number of systems and scopes in the first program in Experiment 1, and the effect of number of systems on the second program in Experiment 2. The execution time in both experiments includes the running time for 10,000 interactions. It does not include reading of input and program files and initializing variables on CPU or storing results, but it does contain initializing and

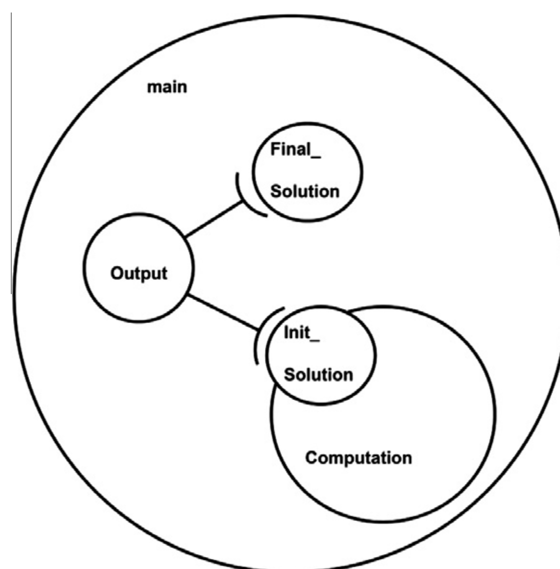


Fig. 17. Final_Sol is the final solution system and the output of the program. **Init_sol** is an initialized solution. The output context selects an initialized solution system and updates final solution system, i.e., it randomly searches the population in order to find the best overall solution.

Table 1
Hardware and operating system specifications are used for experiments.

CPU	Intel® dual core™, 2.40 GHz	
RAM	2 GB	
OS	Microsoft Windows XP professional 2002 SP1	
GPU	Name	GeForce 9800 GT
	CUDA	1.1
	Size of Global memory	1 GB
	Multiprocessors	14
	Number of cores	112
	Clock Rate	1.62 GHz

updating GPU memory, allocating and releasing memory from it. The hardware and software specification used in this work is given in Table 1. Bentley's original Systemic Computation Architecture was written with C language [11], thus we use C language based on CUDA as programming language.

Experiment 1. The goal of the first experiment is to compare the performance of the new GPU parallel implementation with the previous sequential implementation of Systemic Computation, by increasing the number of scopes and non-context systems on the chemical diffusion program explained in Section 4.1. The configuration of experiment is:

– Number of Context systems: 9.
– Number of non-context systems (ink and water): 250 to 2000 for sequential implementation and 250 to 4000 for parallel implementation (each increment is double the previous increment).
– Number of Scopes or cells: 9 to 144 scopes (each increment is double the previous increment) and one main scope.

In this experiment, the increasing the number of systems refers to the increasing the number of non-context system. Increasing the number of scopes refers to increasing the number of cells. The number of ink and water systems are equal in each program. In other words, the amount of ink that is diffused in water is equal to the amount of water. Each systemic program runs for each combination of number of scopes and non-context systems. Each program was run five times on sequential implementation with non-context systems of 500 or less and three times for more than 500, and 10 times on the parallel implementation. Reported running time is the average of programs' running times.

Results 1. As shown in Fig. 18, the performance shows fluctuation. Sometimes increasing the number of scopes helps context systems to find their input systems more easily, for example finding cells as input of Initialiser context system. But sometimes makes it difficult to find matches for context systems, for example for a large number of scopes it takes time for the Initialiser to fill scopes with water and ink, and there are no systems available to interact in the context of the jump and impact systems for a while. Fig. 19 shows the gradual increase in execution time on the parallel implementation because the number of threads on the GPU increases significantly. As shown in Fig. 18, the variation of curves increases significantly as

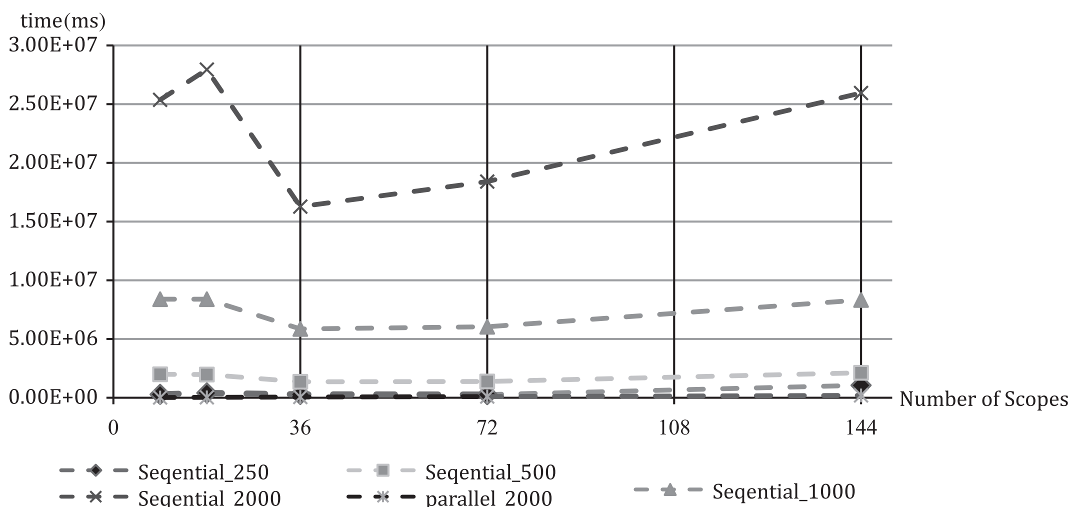


Fig. 18. Execution time of diffusion ink in water program on sequential implementation with increasing number of scopes. **X** in **Sequential_X** and **Parallel_X** refers to number of ink and water systems in the experiment. The **Parallel_2000** line is shown for comparison purposes, falling under all of the sequential curves.

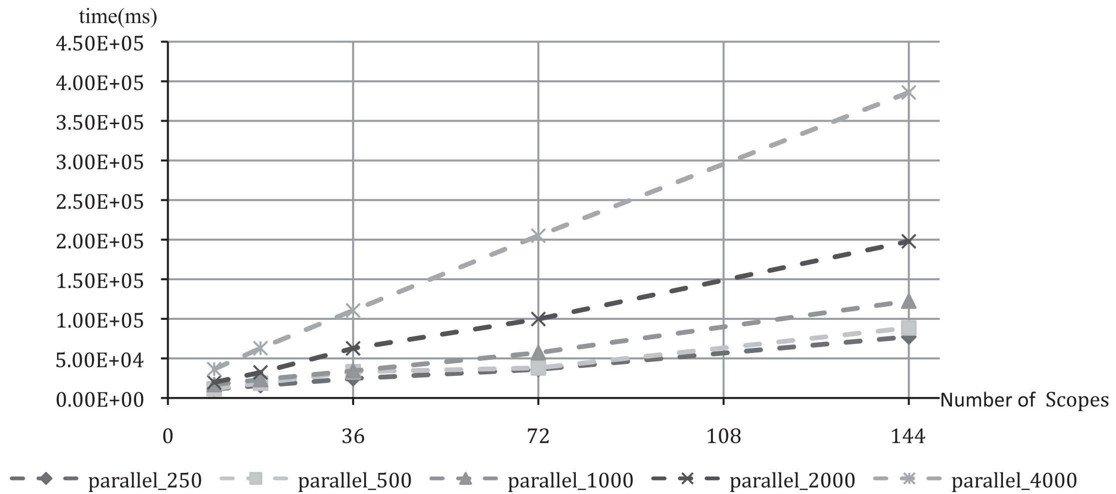


Fig. 19. Execution time of diffusion ink in water program on parallel implementation with increasing number of scopes. **X** in **parallel_X** refers to number of ink and water systems in the experiment.

number of systems increases. In Fig. 18 execution time for 2000 systems on the parallel implementation is shown. As can be seen, it is less than all program execution times for the sequential implementation. Fig. 20 shows an average of the results for the sequential and parallel experiments from 250 to 2000 systems.

As can be seen in Fig. 20, on average the execution time of the sequential implementation increases with a 3rd degree polynomial, characteristic of algorithms and implementations with time complexity $O(n^3)$. The parallel implementation appears to increase linearly, characteristic of algorithms and implementations with time complexity $O(n)$. Fig. 21 shows the sequential divided by parallel times. As can be seen, the increase in performance varies according to the number of systems, but when more scopes are added, the increase in speed becomes less significant for larger numbers, e.g., it is 606 times as fast for nine scopes, 429 times as fast for 18 scopes, 154 times as fast for 36 scopes and 77 times as fast for 144 scopes.

When varying the number of systems with a fixed number of scopes, the results are more clear-cut. Figs. 22–24 show the results. In Figs. 22 and 23 it is clear that the execution time of the two implementations increases in proportion to the number of systems. In this experiment, increasing the number of systems has a larger negative effect on the performance of the sequential implementation compared to that of the parallel implementation. The reason is because in the parallel implementation all combinations of triplets are checked, but in the sequential version triplets are found randomly. For example, increasing number of ink and water systems decreases the selection probability of cells; therefore, Initialiser and

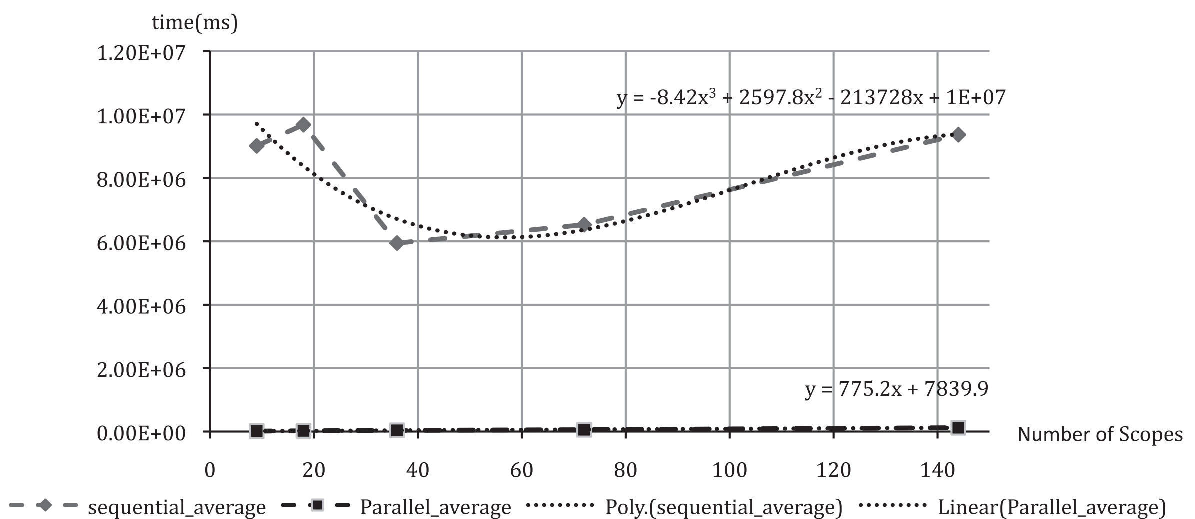


Fig. 20. Average execution time of diffusion ink in water program on both implementations. Both line and curve are average of the experiment with 250 to 2000 systems shown in Figs. 18 and 19. **Parallel_average** is average of four experiments on the parallel implementation and **Sequential_average** is average of four experiments on the sequential implementation.

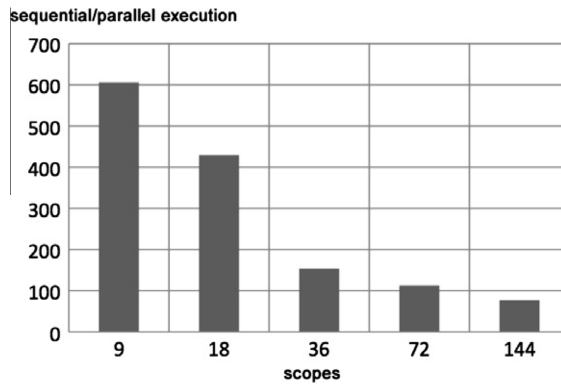


Fig. 21. Sequential divided by parallel execution times for different numbers of systems in the program.

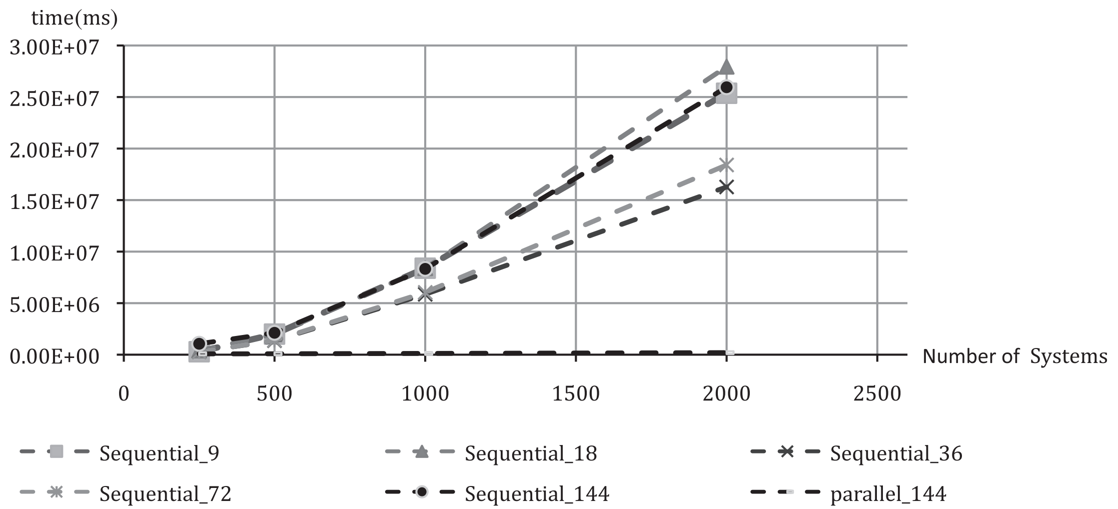


Fig. 22. Execution time of diffusion ink in water program on sequential implementation with increasing number of systems. **X** in **Parallel_X** and **Sequential_X** refers to the number of scopes in the experiment. **Parallel _X** shows the slowest result for the experiment using the parallel implementation for comparison; it falls under all the sequential curves.

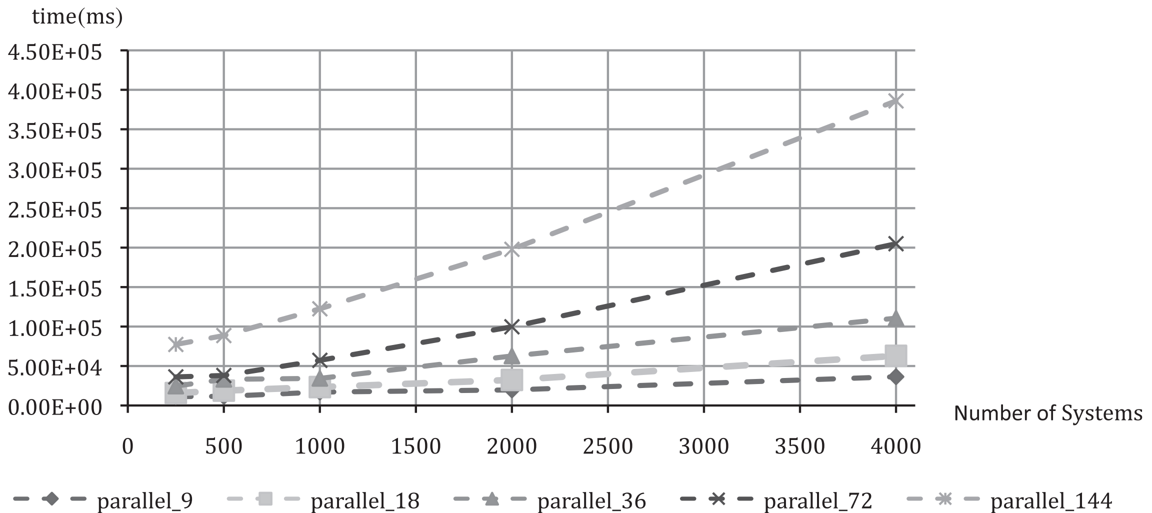


Fig. 23. Execution time of diffusion ink in water program on parallel implementation with increasing number of systems. **X** in **Parallel_X** refers to number cells or scopes in the experiment.

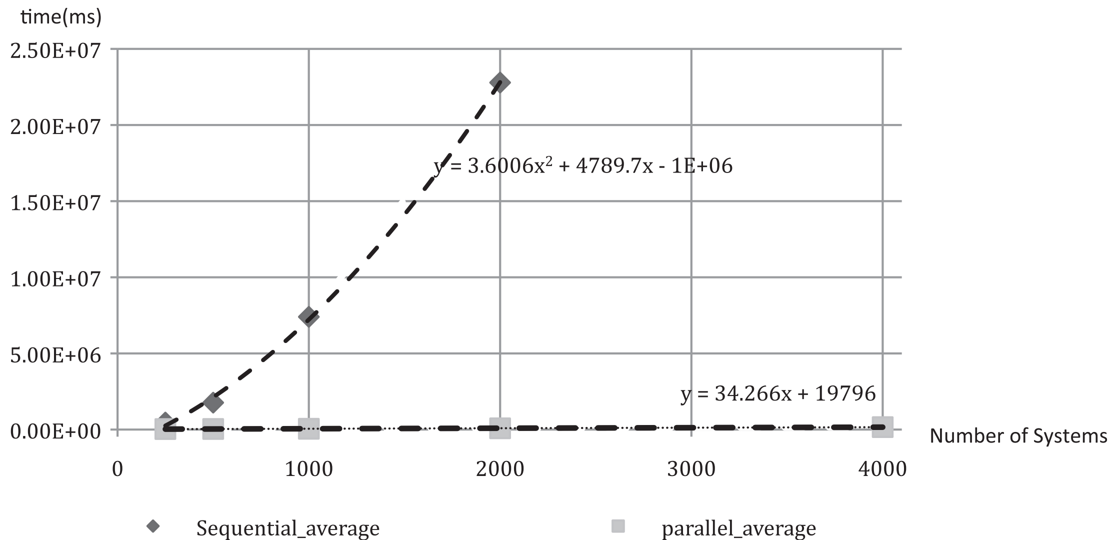


Fig. 24. Average execution time of chemical diffusion program on both implementations with increasing number of systems. Curve is average of experiment with 250 to 2000 systems for sequential implementation that is shown in Fig. 21 and line is average of experiment with 250 to 4000 systems for parallel implementation that shown in Fig. 22. **Parallel_average** is average of four experiments on parallel implementation and **Sequential_average** is average of four experiments on sequential implementation.

jump contexts cannot find matching systems easily. Fig. 24 shows the average results. As can be seen, on average the execution time of the sequential implementation increases with a 2nd degree polynomial, characteristic of algorithms and implementations with time complexity $O(n^2)$. The parallel implementation appears to increase linearly, characteristic of algorithms and implementations with time complexity $O(n)$. Fig. 25 shows the sequential divided by parallel times. As can be seen, the increase in performance varies according to the number of systems with more improvement seen for larger numbers of systems, e.g., it is 14 times as fast for 250 systems, 46 times as fast for 500 systems, 146 times as fast for 1000 systems and 276 times as fast for 2000 systems.

Experiment 2. The goal of the second experiment is to compare the performance of the new parallel implementation and Bentley's original sequential implementation of the systemic computation architecture with increasing number of solution systems on the GA binary knapsack program explained in Section 4.2. In this experiment, increasing number of systems refers to increasing number of GA solution systems. Each systemic computation program was run for 10 times; reported execution time is the average of all programs' execution time. In this experiment, the number of knapsack objects is 16; the maximum knapsack's weight is 80.0 kg. The configuration of experiment is:

– Context systems: 3 GA systems and 1 output system.
– Solution systems: 50 to 4000 systems for sequential implementation and 50 to 8000 systems for parallel implementation (each increment is double the previous increment except 800 to 1000 with an increment of 200).

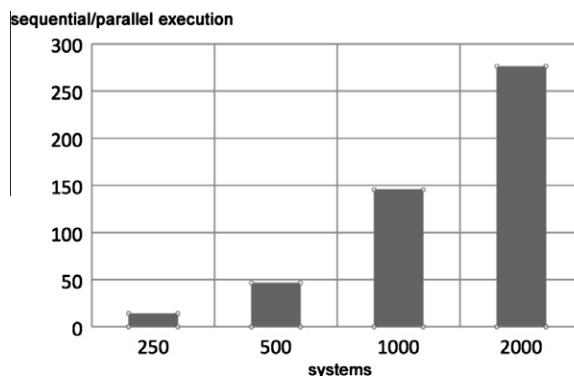


Fig. 25. Sequential divided by parallel execution times for different numbers of systems in the program. (Only four points shown as sequential was too slow to run for higher numbers of systems.)

– Final Solution system: 1 system.
– Scope: 1 main scope and 1 computation scope.
– Initial increment: 50

Results 2. Fig. 26 shows the execution time for both parallel and sequential implementation with increasing number of systems. As can be seen in the figure, the execution time of the sequential implementation increases with a 2nd degree polynomial, characteristic of algorithms and implementations with time complexity $O(n^2)$. The parallel implementation appears to increase linearly, characteristic of algorithms and implementations with time complexity $O(n)$. The increase in performance varies according to the number of systems, e.g., it is 2.3 times as fast for 50 systems, 108 times faster for 800 systems, 256 times faster for 2000 systems, and 465 times faster for 4000 systems.

4.4. Discussion

In Bentley's original sequential implementation of systemic computation [11], the random selection of systems when finding valid triplets decreases performance. In his sequential implementation, increasing the number of systems helps context systems to find their two input interacting systems more easily, so the performance increases. Increasing the number of systems can also decrease the selection probability of other systems, so in this case context systems cannot find one or two matching systems easily, and so performance decreases. Sometimes it is very difficult to assess this situation because systems may change during the execution of the program and so some systems will not match contexts that they previously matched; instead they may match new context systems. Sometimes even the context system schemata may change, altering which systems it matches. In addition, the implementation of the program in the systemic computation language itself affects the performance. For example in the chemical diffusion example, more scopes mean more time is needed to fill them with ink and water systems. Therefore, there are scopes without ink and water or even

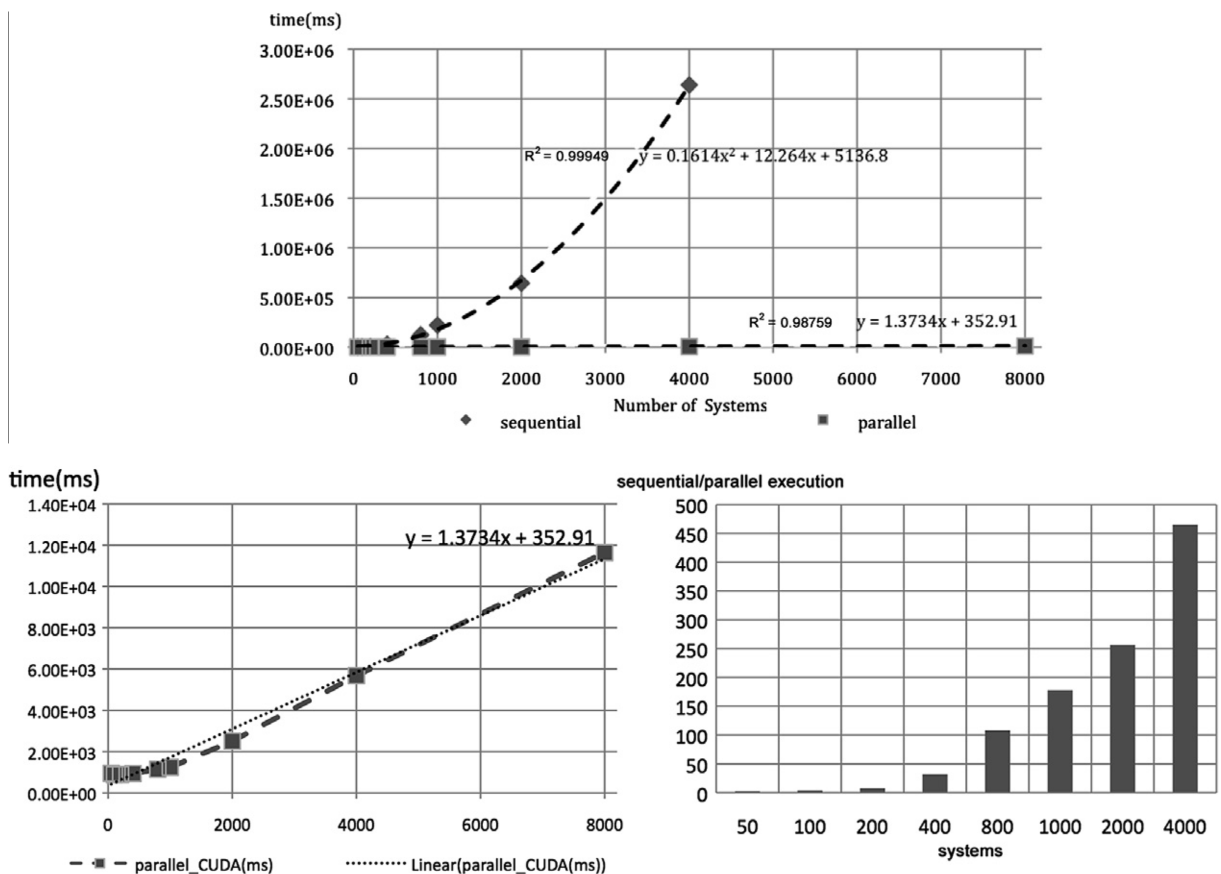


Fig. 26. Top: execution time of knapsack problem on both sequential and parallel implementation with increasing number of systems. Bottom left: execution time of parallel implementation alone. Bottom right: sequential divided by parallel execution times for different numbers of systems in the program.

ink and water with different velocity. However, the context systems in these scopes have equal probability for selection in the sequential implementation.

In the parallel implementation, increasing number of systems means that more data is transferred from CPU to GPU, which takes more time. If system definitions or system scopes change frequently then more data has to be transferred in order to update the GPU memory, and performance decreases. If a new scope is added then the number of blocks increases on the device. Adding one scope to the program adds

$$Ceil((Systems + 1)/16)^2 \times (scp + 1) \times func - Ceil((Systems)/16)^2 \times scp \times func$$

blocks of threads to the device for processing, where *Systems* is number of systems except of new scope, *scp* is the number of existing scopes and *func* is number of context systems. When the number of blocks increases, more time is needed to process them, and performance decreases. Because of the structure of the parallel implementation, increasing the number of scopes or contexts decreases performance more than increasing other systems, but execution time for both cases increases linearly. In addition, in both implementations more systems mean a longer search through more systems in order to find valid scopes and functional systems, which decreases performance.

Although the parallel implementation shows linear time complexity for the experiments reported here, the improvement derives from the efficient division of labor using the parallel resources of the GPU; it is likely that as the number of systems increases beyond this capacity, the execution time will appear as $O(n^2)$. Alternative approaches which may alter the underlying algorithm and thus improve the time complexity will be investigated in the future.

Neither implementation is a perfect representation of systemic computation, for both suffer from biases in probability of interaction caused by architecture-specific issues. In theory, given one context system that matches a pool of other systems, the probability of those systems interacting in that context should not change even if there is more than one identical context system. The probability of interaction should depend on the degree to which systems match the context systems, the degree to which the systems share membership of a parent scope and on the relative numbers of interacting systems. The new parallel GPU Systemic Computation Architecture is thus much closer to the systemic computation concept compared to Bentley's original sequential implementation [11] because the *producer* stage creates a complete list of valid systemic triplets, which are then used to determine the interactions. More or fewer identical context systems in the same scope should not alter the probability of interaction; however more or fewer interacting systems in one scope relative to another may affect the probability of interaction. Both implementations had to simulate the concept of asynchrony by choosing systems randomly for interaction, as both used clocked architectures.

5. Conclusion

The need for fast bio-inspired computation has never been greater. Systemic computation is a new bio-inspired model of computation that has shown considerable success for biological modeling and bio-inspired computation [11–20]. However until now it has only been available as a serial simulation running on conventional processors. In this work the first parallel GPU Systemic Computation Architecture was presented. Its performance was assessed by comparing the change in execution time needed when scaling up the number of systems within two programs: chemical diffusion and a genetic algorithm knapsack problem. As the number of systems increased, the parallel GPU architecture became several hundred times faster than the existing implementations. It also appeared to have a linear scaling as more systems are added to the systemic computation program, compared to the previous implementation, which showed execution times increasing with time complexity $O(n^2)$ or worse, where n = number of systems.

These highly successful results will make it possible to investigate systemic models of more complex biological systems in the future. Further improvements are also planned. The GPU SC architecture is just the first step towards creating a fully parallel systemic computer. Future work will continue the development of parallel SC architectures and will investigate the use of reconfigurable hardware such as FPGAs.

Appendix A

The systemic computation code for the chemical diffusion program (ink in water) is provided in this section. There are only three cells, two ink systems and six water systems. The details of functional systems is shown in Fig. 27.

```
#systemic start
// define functions here
#function initial_Diffusion %b1001100000000000
#function Impact            %b0101100000000000
#function JUMP              %b1101100000000000

// define some useful labels
#label zero          %b0000000000000000
#label dontcare      %b????????????????
```

Context System Name	[Schemata 1]	[Context]	[Schemata 2]
JMP_LA	[1?10...0 NOP 110…0] (Init_molecule , velocity = -1)	Jump(0,0)	[01?0…0 NOP 01…0] (cell, type A)
JMP_LB	[1?10...0 NOP 110…0] (Init_molecule , velocity = -1)	Jump(0,0)	[01?0…0 NOP 10…0] (cell, type B)
JMP_LC	[1?10...0 NOP 110…0] (Init_molecule , velocity = -1)	Jump(0,0)	[01?0…0 NOP 11…0] (cell, type C)
JMP_RA	[1?10...0 NOP 100…0] (Init_molecule , velocity = +1)	Jump(0,0)	[01?0…0 NOP 01…0] (cell, type A)
JMP_RB	[1?10...0 NOP 100…0] (Init_molecule , velocity = +1)	Jump(0,0)	[01?0…0 NOP 10…0] (cell, type B)
JMP_RC	[1?10...0 NOP 100…0] (Init_molecule , velocity = +1)	Jump(0,0)	[01?0…0 NOP 11…0] (cell, type C)
Impact_LR	[1010…0 NOP 110…0] (Init_Ink , velocity = -1)	Impace(0,0)	[1110…0 NOP 100…0] (Init_Water , velocity = +1)
Impact_RL	[1010…0 NOP 100…0] (Init_Ink , velocity = +1)	Impact(0,0)	[1110…0 NOP 110…0] (Init_Water , velocity = -1)
Ink_Initialiser	[100…0 NOP ??...?] (non_Init_Ink)	Initialiser(0,0)	Initialiser(0,0) (Cell_Ink)
Water_Initialiser	[110…0 NOP ??...?] (non_Init_Water)	Initialiser(0,0)	[011…0 NOP ?...?] (Cell_Water)

Fig. 27. The definition of each context system for the chemical diffusion problem. **NOP** means no operation. If a system's functional part is **NOP** it means it is non-context system (i.e., it may interact with other systems but it cannot act as a context and affect the result of the interaction of others).

```
#label init_Molecule    %b1?10000000000000
#label nonInit_Ink      %b1000000000000000
#label nonInit_Water    %b1100000000000000
#label init_Ink         %b1010000000000000
#label init_Water       %b1110000000000000
#label CELL             %b01?0000000000000
#label CELL_INK         %b0100000000000000
#label CELL_WATER       %b0110000000000000
#label TypeA            %b0100000000000000
#label TypeB            %b1000000000000000
#label TypeC            %b1100000000000000
#label LEFT             %b1100000000000000
#label RIGHT            %b1000000000000000
// and the program begins here:
main (%d0 %d0 %d0)
init_ink ([nonInit_Ink %d0 dontcare] initial_Diffusion (0,0) [CELL_INK %d0

dontcare])
init_water ( [nonInit_Water %d0 dontcare] initial_Diffusion (0,0) [CELL_WATER
%d0 dontcare])
JmpLA ( [init_Molecule %d0 LEFT] JJUMP (0,0) [CELL %d0 TypeA])
JmpLB ( [init_Molecule %d0 LEFT] JJUMP (0,0) [CELL %d0 TypeB])
JmpLC ( [init_Molecule %d0 LEFT] JJUMP (0,0) [CELL %d0 TypeC])
JmpRA ( [init_Molecule %d0 RIGHT] JJUMP (0,0) [CELL %d0 TypeA])
JmpRB ( [init_Molecule %d0 RIGHT] JJUMP (0,0) [CELL %d0 TypeB])
JmpRC ( [init_Molecule %d0 RIGHT] JJUMP (0,0) [CELL %d0 TypeC])

conflictLR ([init_Ink %d0 LEFT] Impact (0,0) [init_Water %d0 RIGHT])
conflictRL ([init_Ink %d0 RIGHT] Impact (0,0) [init_Water %d0 LEFT])

cell_0 ( CELL_INK %d0 TypeA )
cell_1 ( CELL_WATER %d0 TypeB )
```

```
cell_2 ( CELL_WATER %d0 TypeC )

ink_0 ( nonInit_Ink %d0 zero )
ink_1 ( nonInit_Ink %d0 zero )
water_0 ( nonInit_Water %d0 zero )
water_1 ( nonInit_Water %d0 zero )
water_2 ( nonInit_Water %d0 zero )
water_3 ( nonInit_Water %d0 zero )
water_4 ( nonInit_Water %d0 zero )
water_5 ( nonInit_Water %d0 zero )

// set up the scopes
#scope main
{
   init_ink
   init_water
   cell_0
   cell_1
   cell_2
   ink_0
   ink_1
   water_0
   water_1
   water_2
   water_3
   water_4
   water_5
}
#scope cell_0
{
   cell_2
   cell_1
   JmpLC
   JmpRB
   conflictLR
   conflictRL
}
#scope cell_1
{
   cell_0
   cell_2
   JmpLA
   JmpRC
   conflictLR
   conflictRL
}
#scope cell_2
{
   cell_1
   cell_0
   JmpLB
   JmpRA
   conflictLR
   conflictRL
}
#systemic end
```

Appendix B

The systemic computation code for the GA binary knapsack program is provided in this section. The details of functional systems is shown in Fig. 28.

Context System Name	[Schemata 1]	[Context]	[Schemata 2]
BinaryMutation	[??......? NOP 100…0] (Init_Solution)	BinaryMutation (0,0)	[??......? NOP 100…0] (Init_Solution)
UniformCrossover	[??......? NOP 100…0] (Init_Solution)	UniformCrossover (0,0)	[??......? NOP 100…0] (Init_Solution)
One-pointCrossover	[??......? NOP 100…0] (Init_Solution)	One-pointCrossover (0,0)	[??......? NOP 100…0] (Init_Solution)
Output	[??......? NOP 100…0] (Init_Solution)	Output (0,0)	[??......? NOP 110…0] (Final_Solution)
Initialiser	[??......? NOP 100…0] (Init_Solution)	Output (0,0)	[1111111100000000 NOP 1111111100000000] (Computation)

Fig. 28. The implementation of context systems for the GA binary knapsack problem in SC language. **NOP** means no operation. If a system's functional part is **NOP** it means it is non-context system (i.e., it may interact with other systems but it cannot act as a context and affect the result of the interaction of others).

```
#systemic start
// define functions here
#function OUTPUT            %b0010100000000000
#function INITIALIZE        %b1010100000000000
#function UniCrossOver      %b0110100000000000
#function OnePointCrossOver  %b1110100000000000
#function BinMutation       %b0001100000000000

  // define some useful labels
#label zero                 %b0000000000000000
#label dontcare             %b????????????????
#label comp                 %b1111111100000000
#label Sol                  %b1000000000000000
#label FinalSol             %b1100000000000000

  // and the program begins here:
main (%d0 %d0 %d0)
OutSolution ( zero %d0 FinalSol )
solution1 ( zero %d0 zero )
solution2 ( zero %d0 zero )

//define more solution system here
initializer ([zero %d0 zero] INITIALIZE (0,0) [comp %d0 comp])
output ([dontcare %d0 Sol] OUTPUT (0,0) [dontcare %d0 FinalSol])
UniformCross ([dontcare %d0 Sol] UniCrossOver (0,0) [dontcare %d0 Sol])
OnePointCross ([dontcare %d0 Sol] OnePointCrossOver (0,0) [dontcare %d0 Sol])
BMuatation ([dontcare %d0 Sol] BinMutation (0,0) [dontcare %d0 Sol])
computation ( comp %d0 comp )

// set up the scopes
#scope main
{
    OutSolution
    solution1
    solution2
    //write other solution system here if any exists
    initializer
    computation
    output
}
#scope computation
{
```

```
    UniformCross
    OnePointCross
    BMuatation
}
#systemic end
```

References

[1] W.R. Holmes, R. Jung, P. Roberts, Computational neuroscience (CNS*2008), BMC Neuroscience 9 (Suppl. 1) (2008) I1 (11 July 2008).
[2] S. Bullock, J. Noble, R. Watson, M.A. Bedau (Eds.), Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems, MIT Press, Cambridge, MA, 2008, ISBN 978-0-262-75017-2.
[3] J.D. Wren, D. Wilkins, J.C. Fuscoe, S. Bridges, S. Winters-Hilt, Y. Gusev, Proceedings of the 2008 midsouth computational biology and bioinformatics society (MCBIOS) conference, BMC Bioinformatics 9 (Suppl. 9) (2008) S1.
[4] L. Dupuy, J. Mackenzie, T. Rudge, J. Haseloff, A system for modelling cell interactions during plant morphogenesis, Annals of Botany (2007).
[5] D. Blain, M. Garzon, S.-Y. Shin, B.-K. Zhang, S. Kashiwamura, M. Yamamoto, A. Kameda, A. Ohuchi, Development, evaluation and benchmarking of simulation software for biomolecule-based computing, Natural Computing 3 (4) (2004) 427–442.
[6] F. Rothlauf (Ed.), Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2009, July 8–12, ACM, Montreal, Québec, Canada, 2009, ISBN 978-1-60558-325-9.
[7] P. Andrews, J. Timmis, N. Owens, U. Aickelin, E. Hart, A. Hone, A. Tyrrell (Eds.), Proceedings of the Artificial Immune Systems: Eighth International Conference, ICARIS 2009, August 9–12, Lecture Notes in Computer Science/Theoretical Computer Science and General Issues, Springer, York, UK, 2009.
[8] S. Kumar, P.J. Bentley (Eds.), On Growth, Form and Computers, Academic Press, London, 2003.
[9] C. Alippi, M. Polycarpou, C. Panayiotou, G. Ellinas, in: Proceedings of the Artificial Neural Networks ICANN 2009: 19th International Conference, September 14–17, Lecture Notes in Computer Science and General Issues, Springer, Limassol, Cyprus, 2009.
[10] Y. Shi, R.C. Eberhart (Eds.), IEEE Swarm Intelligence Symposium 2009, IEEE Publications, 2009.
[11] P. Bentley, Systemic computation: a model of interacting systems with natural characteristics, International Journal of Parallel, Emergent and Distributed Systems 22 (2007) 103–121.
[12] E. Le Martelot, P.J. Bentley, R.B. Lotto, A systemic computation platform for the modelling and analysis of processes with natural characteristics, in: Proceedings of the Ninth Genetic and Evolutionary Computation Conference (GECCO 2007) Workshop: Evolution of Natural and Artificial Systems – Metaphors and Analogies in Single and Multi-Objective Problems, July 7–11, University College London, London, UK, 2007, pp. 2809–2816.
[13] P.J. Bentley, Methods for improving simulations of biological systems: systemic computation and fractal proteins, in: Special Issue on Synthetic Biology, Journal of the Royal Society, Interface 6 (2009) S451–S466, doi:10.1098/rsif.2008.0505.focus.
[14] E. Le Martelot, P.J. Bentley, R.B. Lotto, Exploiting natural asynchrony and local knowledge within systemic computation to enable generic neural structures, in: Proceedings of the Second International Workshop on Natural Computing (IWNC 2007), December 10–13, Nagoya University, Nagoya, Japan, 2007, pp. 122–133.
[15] E. Le Martelot, P.J. Bentley, Metabolic systemic computing: exploiting innate immunity within an artificial organism for on-line self-organisation and anomaly detection, in: P.J. Bentley, D. Lee (Eds.), Special Issue on Artificial Immune Systems, The Journal of Mathematical Modelling and Algorithms (JMMA) 8 (2) (2009) 203–225.
[16] E. Le Martelot, P.J. Bentley, Modelling biological processes naturally using systemic computation: genetic algorithms, neural networks, and artificial immune systems, in: R. Choing (Ed.), Nature-Inspired Informatics for Intelligent Applications and Knowledge Discovery: Implications in Business, Science and Engineering, IGI Global, 2008, pp. 204–241.
[17] E. Le Martelot, P.J. Bentley, R.B. Lotto, Eating data is good for your immune system: an artificial metabolism for data clustering using systemic computation, in: Proceedings of the Seventh International Conference on Artificial Immune Systems (ICARIS 2008), August 10–13, Phuket, Thailand, 2008, pp. 412–423.
[18] E. Le Martelot, P.J. Bentley, R.B. Lotto, Crash-proof systemic computing: a demonstration of native fault-tolerance and self-maintenance, in: Proceedings of the Fourth IASTED International Conference on Advances in Computer Science and Technology (ACST 2008), April 2–4, Langkawi, Malaysia, 2008, pp. 49–55.
[19] P.J. Bentley, Designing biological computers: systemic computation and sensor networks, bio-inspired computing and communication, notes arising from BIOWIRE 2007, in: A Workshop on Bioinspired Design of Networks, in Particular Wireless Networks and Self-Organizing Properties of Biological Networks, 2007.
[20] E. Le Martelot, P.J. Bentley, On-line systemic computation visualisation of dynamic complex systems, in: Proceedings of the 2009 International Conference on Modeling, Simulation and Visualization Methods (MSV'09), July 13–16, Las Vegas, Nevada, USA, 2009, pp. 10–16.
[21] J. Owens et al, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.
[22] I. Buck et al, Brook for GPUs: stream computing on graphics hardware, ACM Transactions on Graphics 23 (2004) 777–786.
[23] C. Rodrigues et al., GPU acceleration of cutoff pair potentials for molecular modeling applications, in: Conference on Computing Frontiers, 2008, pp. 273–282.
[24] M. Schatz, C. Trapnell, Fast Exact String Matching on the GPU, Center for Bioinformatics and Computational Biology, 2007.
[25] T. Clayton, L. Patel, G. Leng, A. Murray, I. Lindsay, Rapid evaluation and evolution of neural models using graphics card hardware, in: Genetic and Evolutionary Computation Conference, 2008, pp. 299–306.
[26] D. Eriksson, A principal exposition of Jean-Louis Le Moigne's systemic theory, Review Cybernetics and Human Knowing 4 (1997) 2–3.
[27] R. Milner, Axioms for biographical structure, Journal of Mathematical Structures in Computer Science 15 (2005) 1005–1032.
[28] R. Milner, The polyadic pi-calculus: a tutorial, in: F.L. Hamer, W. Brauer, H. Schwichtenberg (Eds.), Logic and Algebra of Specification, Springer-Verlag, 1993.
[29] L. Cardelli, Brane calculi. interactions of biological membranes, in: Computational Methods in Systems Biology (CMSB'04), LNCS, vol. 3082, Springer, 2005, pp. 257–280.
[30] A. Adamatzky, Computing in Nonlinear Media and Automata Collectives, IoP Publishing, Bristol, 2001, ISBN 075030751X. 410 pp.
[31] F.J. Varela, Humberto R. Maturana, R. Uribe, Autopoiesis: the organization of living systems, its characterization and a model, BioSystems 5 (1974) 187–196.
[32] A. Adamatzky (Ed.), Collision-Based Computing, vol. XXVII, Springer-Verlag, 2002, ISBN 1-85233-540-8. 556 pp.
[33] R.W. Hamming, Error detecting and error correcting codes, Bell System Technical Journal 29 (2) (1950) 147–160.
[34] NVIDIA Corporation, Programming Guide, version 2.2, NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050, 2009. Available online at: <http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf>.
[35] NVIDIA GeForce 8800 GPU Architecture Overview, Technical Brief TB-02787-001_v01, 2006.
[36] M. Flynn, Some computer organizations and their effectiveness, IEEE Transactions on Computers C-21 (1972) 948.