Acceleration of Genetic Algorithms for Sudoku Solution on Many-core Processors

Yuji Sato, Naohiro Hasegawa and Mikiko Sato

Abstract In this chapter, we use the problem of solving Sudoku puzzles to demonstrate the possibility of achieving practical processing time through the use of many-core processors for parallel processing in the application of evolutionary computation. To increase accuracy, we propose a genetic operation that takes building-block linkage into account. As a parallel processing model for higher performance, we use a multiple-population coarse-grained genetic algorithm (GA) model to counter initial value dependence under the condition of a limited number of individuals. The genetic manipulation is also accelerated by the parallel processing of threads. In an evaluation using even very difficult problems, we show that execution times of several tens of seconds and several seconds can be obtained by parallel processing with the Intel Core i7 and NVIDIA GTX 460, respectively, and that a correct solution rate of 100% can be achieved in either case. In addition, genetic operations that take linkage into account are suited to fine-grained parallelization and thus may result in an even higher performance.

1 Introduction

Research on the implementation of evolutionary computation on massively parallel computing systems to attain faster processing [1, 2, 3, 4, 5] has been going on since

Yuji Sato
Faculty of Computer and Information Sciences, Hosei University, 3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan, e-mail: yuji@k.hosei.ac.jp

Naohiro Hasegawa
Graduate School of Computer and Information Sciences, Hosei University, 3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan, e-mail: naohiro.hasegawa.af@stu.hosei.ac.jp

Mikiko Sato
Faculty of Engineering, Tokyo University of Agriculture and Technology, 2-24-16 Naka-cho Koganei-shi, Tokyo 184-8588, Japan, e-mail: mikiko@namikilab.tuat.ac.jp

about 1990, but it has not come into widespread use. On the other hand, multi-core processors, graphics processing units (GPU) and other such many-core processors have been coming into wide use in ordinary personal computers in recent years. The features of many-core processors include suitability for small and medium scale parallelization of from several to several hundreds of nodes, and low-cost compared to massively parallel computing systems. This environment has stimulated research on parallelization of evolutionary computation on many-core processors [6, 7, 8, 9, 10]. These current report, however, focus on benchmark tests for evolutionary computation using typical GPUs.

The objective of our research is to use an actual and practical problem to demonstrate that practical processing time is possible through the use of a GPU for parallelization of evolutionary computation, even for problems for which the use of evolutionary computation has not been investigated previously because of the processing time problem.

As the first step towards that objective, we take the problem solving Sudoku puzzles [11] and investigate acceleration of the processing with a GPU. The reasons for this approach are listed below.

1. Sudoku puzzles are popular throughout the world.
2. Assuming a single core processor, the processing time for genetic algorithms is much higher than for backtracking algorithms [12]. On the other hand, backtracking algorithms pose problems for parallelization, whereas evolutionary computation is suitable for parallelization. Therefore, increasing the number of GPU cores may make the processing time for genetic algorithms equal to or less than that for backtracking algorithms.
3. The use of multi-core processors has recently expanded to familiar environments like desktop PCs and laptop computers, and it has become easy to experiment with parallelization programs on multi-core processors through thread programming. GPUs are designed for the processing of computer graphics in games. But research on General-Purpose computation on Graphics Processing Units (GPGPU) has begun, and GPUs can be used to support solving a logical game.
4. Although the processing time for the backtracking algorithm increases exponentially as the puzzle size increases from 9×9 to 16×16, the fact that a genetic algorithm (GA) is a stochastic search algorithm opens up the possibility of reversing the processing time ratio.

High-speed evolutionary computation by a GPU requires the design of a parallel program that is dependent on the number of cores and memory capacity in the GPU. This means adjusting the degree of parallelization and the amount of processing allocated to each task according to GPU specifications. In the case of Core i7, however, we focus only on the degree of parallelization afforded by the number of cores in a multi-core processor and propose a method for speeding up Sudoku problem solving by general-purpose thread programming conforming to POSIX specifications [13].

In this paper, we show the possibility of a large reduction in processing time for evolutionary computation by parallelization using a many-core processor. In the following section (Section 2), we explain the rule of Sudoku. In Section 3, we show an improvement in the accuracy of Sudoku puzzle solution for by using a genetic operation that takes building-block linkage into account [14]. In Section 4, we propose an implementation of a parallel genetic algorithm on a GPU. Section 5 describes a comparative evaluation of the solution of a difficult Sudoku puzzle executed on a CPU and on a many-core processor. Section 6 presents a discussion and Section 7 concludes this paper.

2 The Rules of Sudoku

The Sudoku rules are explained in Fig. 1. General Sudoku puzzles consist of a 9×9 matrix of square cells, some of which already contain a numeral from 1 to 9. The arrangement of given numerals when the puzzle is presented is called the starting point. In Fig. 1, it contains 24 non-symmetrical given numbers, and the correct number for the other 57 points should be solved. The degree of difficulty varies with the number of given numerals and their placement. Basically, fewer given numerals means a higher number of combinations among which the solution must be found, and so raises the degree of difficulty. But, there are about 15 to 20 factors that have an effect on difficulty rating [11]. A Sudoku puzzle is completed by filling in all of the empty cells with numerals 1 to 9, but no row or column and no 3×3 sub-block (the sub-blocks are bound by heavy lines in Fig. 1) may contain more than one of any numeral. An example solution to the example Sudoku puzzle given in Fig. 1 is shown in Fig. 2. In this figure, the given numbers marked in bold-face.

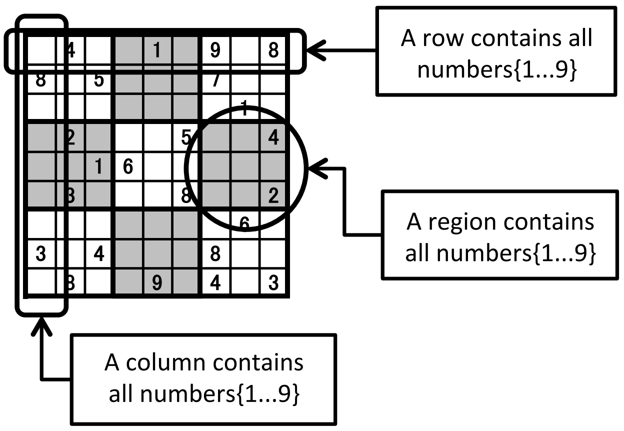

A row contains all numbers{1...9}
A region contains all numbers{1...9}
A column contains all numbers{1...9}

Fig. 1 An example of Sudoku puzzles, each cell of 24 positions contains a given number, the other position should be solved.

	4			**1**		**9**		**8**
8		**5**				**7**		
							1	
	2				**5**			**4**
		1	**6**					
	3				**8**			**2**
							6	
3		**4**				**8**		
	8			**9**		**4**		**3**

6	**4**	3	5	**1**	7	**9**	2	**8**
8	1	**5**	3	2	9	**7**	4	6
2	9	7	8	6	4	3	**1**	5
9	**2**	8	1	7	**5**	6	3	**4**
4	7	**1**	**6**	3	2	5	8	9
5	**3**	6	9	4	**8**	1	7	**2**
7	5	9	4	8	3	2	**6**	1
3	6	**4**	2	5	1	**8**	9	7
1	**8**	2	7	**9**	6	**4**	5	**3**

Fig. 2 A solution for the Sudoku puzzles given in fig 1. The given numbers marked in bold-face.

For these basic puzzles, methods such as back-tracking, which counts up all possible combinations in the solution, and meta-heuristics approach [15] are effective. There also exist some effective algorithms to solve Sudoku puzzles [16, 17, 18] and faster than GA.

On the other hand, there are also many variations of Sudoku. Some are puzzles of larger size, such as 16×16 or 25×25. Others impose additional constraints, such as not permitting the same numeral to appear more than once in diagonals or in special sets of 9 cells that have the same color, etc. For larger puzzles such as 16×16 or 25×25, GA or other stochastic search method may be effective. Methods for speeding up evolutionary computations through implementations on graphics processing units (GPU) may be also effective.

3 Improved Accuracy in Sudoku Solution Using Genetic Operation that Takes Linkage into Account

3.1 Genetic Operations That Takes Linkage into Account

A number of studies on application of GA to solving Sudoku have already been made. On the other hand, there seems to be relatively few scientific papers. References [19, 20], for example, defines a one-dimensional chromosome that has a total length of 81 integer numbers and consists of linked sub-chromosomes for each 3×3 sub-block of the puzzle, and applies uniform crossover in which the crossover positions are limited to the links between sub-blocks. References [21, 22] compares the effectiveness for different crossover methods, including one-point crossover that limit crossover points to links between sub-blocks, two-point crossover, crossover in units of row or column, and permutation-oriented crossover operation. In these examples, optimum solutions to simple puzzles are easily found, but the optimum solutions for difficult puzzles in which the starting point has few givens are often not obtainable in realistic time. We believe the reason for the failure of this design is that the main GA operation, crossover, tends to destroy highly fits, schemata (BB) [23]. To avoid that problem, we defined 9×9 two-dimensional arrays as the GA chromosome and proposed a crossover operation [14] that takes building-block linkage into account.

The fitness function, Eq. (1), is based on the rule that there can be no more than one of any numeral in a row or column. The score of a row (column) is the number of different elements in the row (column). The score f of a given individual x is defined as

$$f(x) = \sum_{i=1}^{9} |g_i| + \sum_{j=1}^{9} |h_j| \tag{1}$$

where $|g_i|$ ($|h_j|$) denotes the number of different numerals in the ith row (jth column). Therefore, maximum score of the fitness function $f(x)$ becomes 162.

An example of this crossover is shown in Fig. 3. In this figure, we assumed that the highest score of each row or column is 9. Therefore, the highest score of each row or column in sub-blocks becomes 27. Child 1 inherits the row information from parent 1, parent 2, and parent 1 in order from top to bottom. Child 2 inherits the column information from parent 1, parent 2, and parent 2 in order from left to right. Mutations are performed for each sub-block. Two numerals within a sub-block that are not given in the starting point are selected randomly and their positions are swapped. We added a simple local search function in which multiple child candidates are generated when mutation occurs, and the candidate that has the highest score is selected as the child. These experiments use tournament selection.

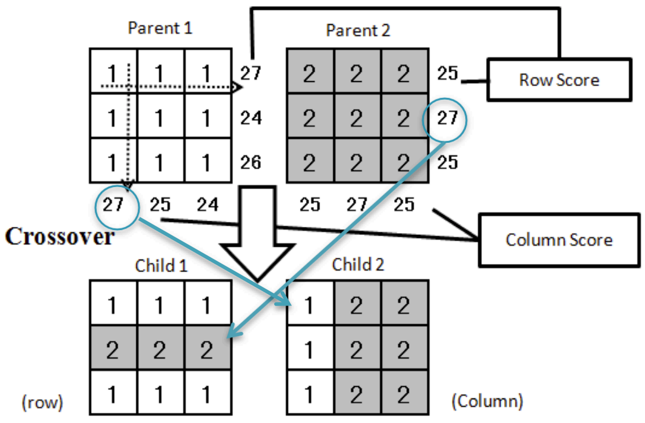
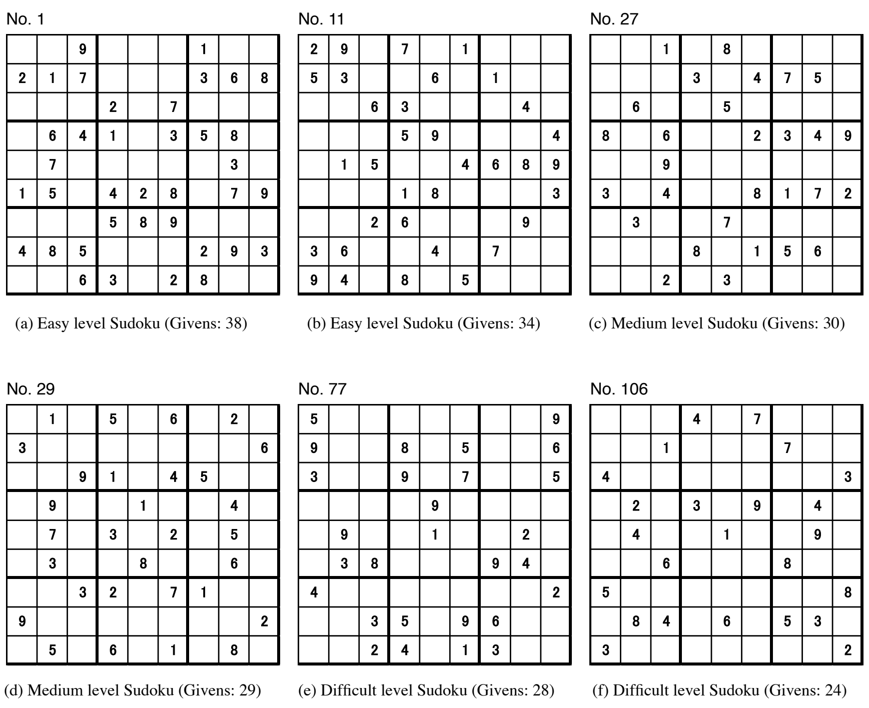


Fig. 3 An example of the crossover considering the rows or the columns that constitute the sub-blocks.

3.2 Sudoku Solution Accuracy by GA

For the puzzles used to investigate the effectiveness of the genetic operations proposed in [14], we selected two puzzles from each level of difficulty in the puzzle set from a book [24]: puzzles 1 and 11 from the easy level, 29 and 27 from the intermediate level, and 77 and 106 from the difficult level, for a total of six puzzles. We also used the particularly super difficult Sudoku puzzles introduced in reference [25]. An example of the puzzles used in the experiment is shown in Fig. 4 and Fig. 5 respectively. The experimental parameters are population size: 150, number of child candidates/parents: 2, crossover rate: 0.3, mutation rate: 0.3, and tournament size: 3.

The relation between the number of givens in the starting point and the number of generations required to reach the optimum solution is shown in Table 1 and Fig. 6. For the three cases in which only mutation is applied (a kind of random search), when mutation and the proposed crossover method are applied (mut+cross), and when the local search improvement measure is applied in addition to mutation and crossover (mut+cross+LS), the tests were run 100 times and the averages of the results were compared. The termination point for the search was 100,000 generations. If a solution was not obtained before 100,000 generations, the result was displayed



No. 1

		9				1		
2	1	7				3	6	8
			2		7			
	6	4	1		3	5	8	
	7						3	
1	5		4	2	8		7	9
			5	8	9			
4	8	5				2	9	3
		6	3		2	8		

(a) Easy level Sudoku (Givens: 38)

No. 11

2	9		7		1			
5	3			6		1		
		6	3				4	
			5	9				4
	1	5			4	6	8	9
			1	8				3
		2	6				9	
3	6			4		7		
9	4		8		5			

(b) Easy level Sudoku (Givens: 34)

No. 27

		1		8				
			3		4	7	5	
	6			5				
8		6			2	3	4	9
		9						
3		4			8	1	7	2
	3			7				
			8		1	5	6	
		2		3				

(c) Medium level Sudoku (Givens: 30)

No. 29

	1		5		6		2	
3								6
		9	1		4	5		
	9			1			4	
	7		3		2		5	
	3			8			6	
		3	2		7	1		
9								2
	5		6		1		8	

(d) Medium level Sudoku (Givens: 29)

No. 77

5								9
9			8		5			6
3			9		7			5
				9				
	9			1			2	
	3	8				9	4	
4								2
		3	5		9	6		
		2	4		1	3		

(e) Difficult level Sudoku (Givens: 28)

No. 106

			4		7			
		1				7		
4								3
	2		3		9		4	
	4			1			9	
		6				8		
5								8
	8	4		6		5	3	
3								2

(f) Difficult level Sudoku (Givens: 24)

Fig. 4 The puzzles used to investigate the effectiveness of the proposed genetic operations.

SD1

7	9							3
							6	
8		1			4			2
		5						
3			1					
	4				6	2		9
2				3				6
	3		6		5	4	2	1

(a) GA generated Super difficult Sudoku (Givens: 24)

SD2

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

(b)AI Escarcot - Claim to be the most difficult Sudoku (Givens: 23)

SD3

					3		1	7
	1	5			9			8
	6							
1					7			
		9				2		
			5					4
							2	
5			6			3	4	
3	4		2					

(c) Super difficult Sudoku from www.sudoku.com (Givens: 22)

Fig. 5 The puzzles used for the particularly super difficult Sudoku puzzles.

Table 1 The comparison of how effectively GA finds solutions for the Sudoku puzzles with different difficulty ratings.

Difficulty rating	Givens	mut+cross+LS		mut+cross		Swap mutation	
		Count	Average	Count	Average	Count	Average
Easy(#1)	38	100	62	100	105	100	105
Easy(#11)	34	100	137	100	247	100	247
Medium(#27)	30	100	910	100	2247	100	2247
Medium(#29)	29	100	3193	100	6609	100	6609
Difficult(#77)	28	100	9482	100	20658	100	20658
Difficult(#106)	24	96	26825	74	56428	74	56428

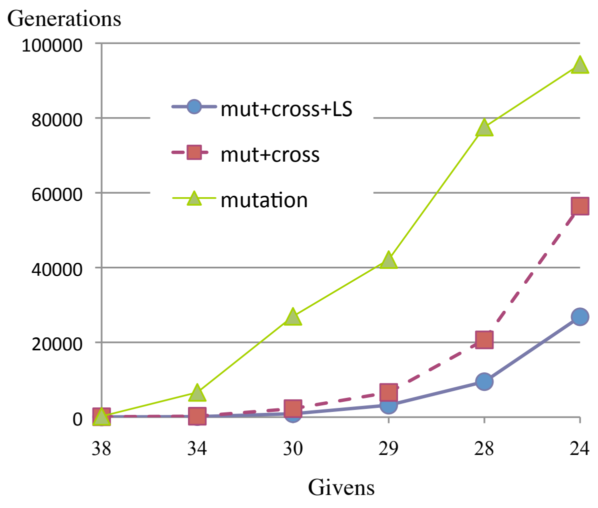


Fig. 6 Relationship between givens and the average number of GA generations needed to find the solution.

as 100,000 generations. When the search is terminated at 100,000 generations, the proportion of obtaining an optimum solution for a difficult puzzle was clearly improved by adding the proposed crossover technique to the mutation, and improved even further by adding the local search function. The mean number of generations until a solution is obtained is also reduced.

In Table 2, the number of times the optimum solution was obtained in 100 test runs using the super difficult Sudoku puzzles is shown [25]. Without any trial limitation the method was solved every time. On the other hand, when using a limit of 100,000 trials, our method was solved 99 times, 83 times, and 74 times out of 100 test runs for three super difficult problems, respectively.

Table 2 The number of times the optimum solution was obtained in 100 test runs using super difficult problems.The numbers represents how many times out of 100 test runs each method reached the optimum.

Sudoku puzzle	SD1	SD2	SD3
Count	99	83	74

On the other hand, Fig. 7 shows the relation of the average number of generations and dispersion. For puzzles that have the same number of initial givens, there is a dependence on the locations of the givens, and a large variance is seen in the mean number of generations needed to obtain the optimum solution. Furthermore, for difficult puzzles that provide few initial givens, there were cases in which a solution was not obtained even when the search termination point was set to 100,000 generations. The reason for that result is considered to be that the search scope for the solution to a difficult puzzle is large and there exist many high-scored local solutions that are far from the optimum solution. Another possibility is that there are puzzles for which the search scope is too broad and there is a dependence on the initial values. The processing time was still very poor compared to the backtracking algorithm.

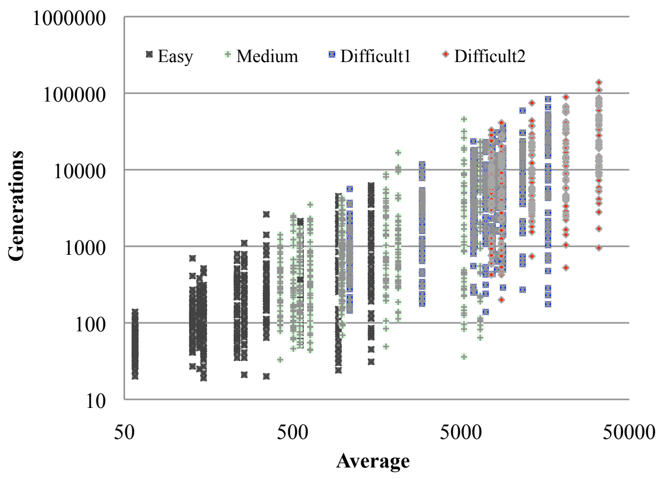


Fig. 7 The difficulty order of tested Sudoku. The minimum and maximum generations needed to solve each Sudoku from 100 test runs as a function of generations needed.

4 Accelerating Evolutionary Computation with Many-core Architecture

4.1 System Architecture for Sudoku Solution

4.1.1 GTX 460 and CUDA Programming

Here we describe parallel processing of a program implemented on the GeForce GTX 460, a commercial GPU from the NVIDIA Corporation that uses the CUDA architecture. The GTX 460 comprises seven streaming multiprocessors (SMs). Each SM has 48 CUDA Cores and comprises up to 48 KB of shared memory. Data reading and writing between SM is accomplished via a large capacity global memory (1 GB). A part of the global memory is a constant read-only memory of 64 KB. The processors within SM can read from and write to the shared memory at high-speed, but the data reading and writing between SM and the global memory is slow. Therefore, the parallelization of evolutionary computation must be implemented with full consideration given to that feature. The basic CUDA operations are broadly grouped into the four classes: (1) reserving GPU memory, (2) data transfer from CPU to GPU, (3) parallel execution on the cores of the GPU, and (4) data transfer from the GPU to the CPU.

CUDA has three units of processing: a thread corresponds to a single process, a block is a number of threads, and a number of blocks of the same size constitute a grid. In CUDA, a thread array of up to three dimensions can be made into a block and a grid can include an array of blocks of up to two dimensions. The unit for the execute instruction from the host is the grid. All of the threads in a grid are executed by the same program, which is called the kernel. The CUDA programming model is a kind of multi-thread model. Each thread is allocated an element in a data array, and the data array serves in the management of the parallel execution of those threads. Threads within the same block share the shared memory inside the SM, so the number of threads within a block is limited to 1024.

4.1.2 System Architecture for Core i7

A conceptual diagram of the homogeneous multi-core processor architecture and system software targeted by this research is shown in Fig. 8. Intel, AMD, and other semiconductor companies have recently been marketing quad-core products for a wide range of computers from PCs to servers. In this research, we target a homogeneous multi-core processor that has recently come to be used in PCs and attempt to speed-up Sudoku puzzle solving on a commonly available system. The OS (Ubuntu 10.04) used in our research provides a POSIX thread interface as an application programming interface (API) and can execute a multi-thread program on a multi-core processor.

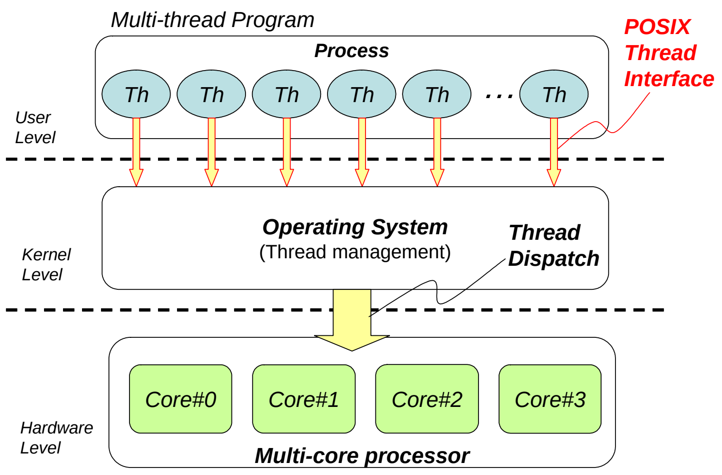


Fig. 8 The system architecture for multi-core processors.

4.2 Parallel GA Model and Implementation for Many-core Architecture

4.2.1 Parallel GA Model and Implementation for GPU Computation

Fig. 9 shows the parallel GA model for GPU computation. Because the grid is the unit of execution for instructions from the host, we conducted experiments with seven blocks in a grid to match the number of SM and with the number of threads in a block equal to three times the number of individuals ($3\times$N) for parallel processing in units of rows or columns that consist of Sudoku region blocks. We allocate the population pools PP and WP, and some working pools to the shared memory of each SM. Here, PP is the population pool to keep individuals of the current population, and WP is a working pool to keep newly generated offspring individuals until they are selected to update PP for the next generation.

The procedure of the parallel GA model for GPU computation is as follows.

1. In the host machine, all individuals are randomly generated and then sent to the global memory of the GPU.
2. Each SM copies the corresponding individuals from global memory to its shared memory, and the generational process is repeated until the termination criteria are satisfied.

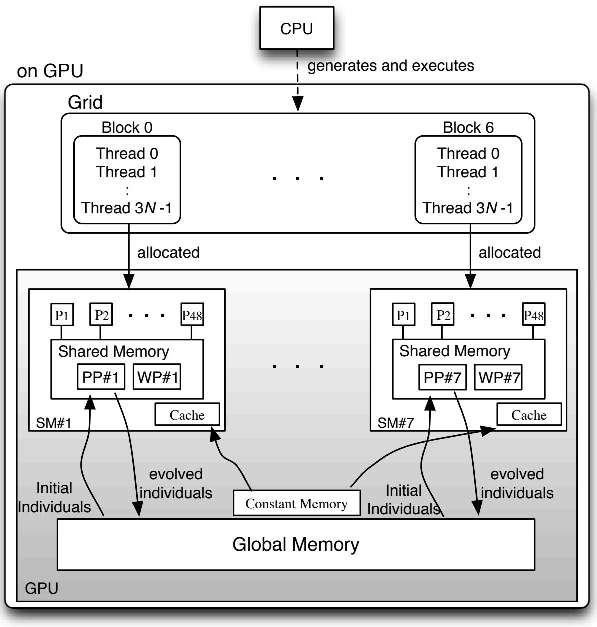


Fig. 9 Parallel GA model for GPU computation.

3. Finally, each SM copies the evolved individuals from its shared memory to global memory.

When applying the GA to the solution of Sudoku, the general procedure is to define an 81-bit one-dimensional chromosome that consists of the joined sub-chromosomes of the various puzzle regions and then perform crossover with the crossover points limited to only the joints between regions. Crossover of this type, however, is believed to easily destroy building blocks. As a one way to solve that problem, we define a two-dimensional chromosome, taking building-block linkage into account. Crossover is then performed by assigning a score to each row or column, each of which consists of region blocks, comparing the scores for the two parent individuals, and then passing the row or columns that have the highest scores on to the child.

Generally, local search functions are effective for constraint satisfaction problems such as Sudoku. Our objective, on the other hand, remains as the solution of Sudoku puzzles in a practical time within the framework of evolutionary computation. Accordingly, we added only a simple local search function in which multiple child candidates are generated when mutation occurs, and the candidate that has the highest score is selected as the child. This is equivalent to the selection function in (μ,λ)-ES and is an operation in the evolutionary computation framework.

Implementation That Takes Measures Against the Initial Value Dependency Problem into Account

From Fig. 7, we can see that when the number of individuals is 150, the number of generations until the optimum solution is found depends on the initial values. If we do not consider parallelization, the processing time is considered to be determined by the product of the number of individuals and the number of generations. Accordingly, from the relation of the proportion of correct solutions obtained to the processing time, we set the number of individuals to 150 on the basis of preliminary experiments. We could conjecture that the effect of parallelization using the GPU would be that increasing the number of individuals would not greatly affect the processing time.

On the other hand, increasing the parallelism requires that the data on individuals and other data required for evolutionary computation be stored in the shared memory of the SM rather than in the global memory, but the shared memory capacity is low and may not hold the data for a sufficient number of individuals. Furthermore, the data transfer speed between SM in the GPU is more than 100 times slower than the communication within SM, so an implementation that requires frequent communication between SM is not suitable. Accordingly, we adopt an implementation method in which each SM runs the same evolutionary computation program, changing only the initial values of the individual data, etc., and whichever SM finds a solution terminates. In other words, the evolutionary computation programs running in the SMs using threads are executed in parallel, and the execution of the same program in each SM with different initial values is considered to serve as a measure against initial value dependency. This can be considered to be SIMD-type parallel processing.

Sparing Use of High-speed Shared Memory

In previous experiments that involved the evaluation of benchmark tests implemented on multi-core processors [10], tables for random number generation were placed on each core to reduce the time required for random number generation. On the other hand, the shared memory of the GTX 460 is small, with a maximum of 48KB, so if random number tables are placed in each block, the required number of individuals cannot be defined. Therefore, the CURAND library function is used instead of the random number tables. Random number generation with CURAND is slower than using random number tables on each core, but it is much faster than generating the random numbers on the host and transferring the values to each SM. Furthermore, because the 64KB read-only constant memory can be accessed at high-speed, the initial arrangement of the Sudoku puzzle (four bytes (int)$\times$81 = 324 bytes) is stored in that cache memory. The data allocated to the shared memory is as follows.

- Area for storing individual data: 1 byte (char) $\times$ 81 $\times$ N $\times$ 2
- Work area for tournament selection: 4 bytes (int) $\times$ N
- Work area for crossover: 4 bytes (float) $\times$ N/2
- Work area for mutation: 1 byte (char) $\times$ 81 $\times$ N

Parallel Processing in Units of Sub-chromosome

Fig. 10 shows an example of the swap mutation within a sub-block and the thread assignment. For a Sudoku puzzle that comprises 3$\times$3 region blocks, the genetic manipulation for each region block is performed in parallel, so nine threads are allocated to the processing for one individual. Because of the limited shared memory capacity, however, we assign here three threads to the processing for one individual. The processing for crossover, mutation or other such purpose for each line or column that consists of three region blocks is accelerated by the parallel processing of three threads.

4.2.2 Parallel GA Model and Implementation for Multi-core Processors

It can be seen from Fig. 7 that the number of generations needed to find an optimal solution for the same Sudoku problem is highly dispersed. This indicates that a Sudoku solution using evolutionary computations is dependent on the initial value when a sufficient number of individuals cannot be set due to insufficient memory capacity or other constraints. With this in mind, we generate the same number of

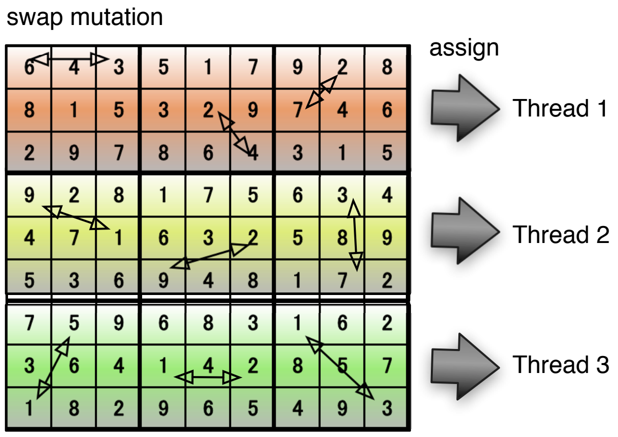


Fig. 10 An example of the swap mutation within a sub-block and the thread assignment.

threads as cores in the target processor and propose a method that executes genetic operations with each core having a different initial value. We also adopt the value of the core that finds a Sudoku solution first.

The procedure of the parallel GA model at each core processor is as follows.

1. All individuals are randomly generated.
2. The generational process is repeated until the termination criteria are satisfied.
3. The core that finds a Sudoku solution first cancels the operations in the other cores.

5 Evaluation experiments

5.1 Execution platform

The specifications of the execution platform used in experiments for Intel Core i7 and GTX 460 are listed in Table 3 and Table 4 respectively. The specifications of the GTX 460 GPU used in these experiments are listed in Table 5.

Table 3 Multi-core processor execution environment.

CPU	Intel Core i7 920 (2.67GHz, 4cores)
OS	Ubuntu 10.04
C compiler	gcc 4.4.3 (optimization “O3”)

Table 4 GPU execution environment.

CPU	Phenom II X4 945 (3GHz, 4 cores)
OS	Ubuntu 10.04
C compiler	gcc 4.4.3 (optimization “O3”)
CUDA Toolkit	3.2 RC

Table 5 GTX 460 specifications.

Board	ELSA GLADIAC GTX 460
#Core	336 (7 SM$\times$48 Core/SM)
Clock	675MHz
Memory	1GB (GDDR5 256 bits)
Shared memory/SM	48KB
#Register/SM	32768
#Thread/SM	1536

5.2 Scalability

The system described here has scalability with respect to the number of cores. Increasing the number of cores in an SM is also considered to improve robustness against the dependence on initial values. For this reason, in the case of Core i7, we varied the number of threads to be executed in parallel from 1 to 8 and surveyed (1) solution rate, (2) average number of generations until the correct solution was obtained, and (3) average execution time. The number of cores in the multi-core processor is 4, but since 2 threads can be executed in parallel in one core by hyper-threading technology, we performed the experiment by executing a maximum of 8 threads in parallel. On the other hand, in the case of GPU, we varied the number of SM ranging from one to seven.

The results are presented in Tables 6, 7, and 8 for Core i7, and in Tables 9, 10, and 11 for GTX 460. The values shown in the results are the averages for 100 experiments that were conducted with 150 individuals and a cut-off of 100,000 generations.

Table 6 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD1).

#Threads	Count [%]	Average Gen.	Exec. time
1	94	32,858	22s 19
2	100	15,268	13s 87
4	100	7,694	13s 11
8	100	3,527	7s 39

Table 7 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD2).

#Threads	Count [%]	Average Gen.	Exec. time
1	82	42,276	28s 41
2	98	25,580	22s 48
4	100	13,261	21s 47
8	100	5,992	12s 12

Table 8 The rate of correct answers, the number of average generations, and the average execution time (Intel Core i7: SD3).

#Threads	Count [%]	Average Gen.	Exec. time
1	69	60,157	39s 88
2	93	46,999	40s 43
4	100	19,982	30s 79
8	100	8,795	17s 13

Table 9 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD1).

#SM	Count [%]	Average Gen.	CPU time
1	62	57,687	16s 728
2	80	40,820	11s 845
4	98	19,020	5s 527
8	100	10,014	2s 906

Table 10 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD2).

#SM	Count [%]	Average Gen.	CPU time
1	50	70,067	20s 199
2	69	58,786	16s 958
4	93	31,254	9s 260
8	97	22,142	6s 391

Table 11 The rate of correct answers, the number of average generations, and the average execution time (GTX 460: SD3).

#SM	Count [%]	Average Gen.	CPU time
1	32	82,742	23s 958
2	59	68,050	19s 722
4	77	47,811	13s 879
8	95	30,107	8s 727

5.3 Experiments on Increasing the Number of Individuals

For a large number of individuals, initial values in a Sudoku solution are highly diverse. To investigate the relationship between diversity in initial values and the results of a Sudoku solution, we varied the number of individuals on Intel Core i7 from 150 to 400 and surveyed (1) solution rate, (2) average number of generations until the correct solution was obtained, (3) average execution time, and (4) minimum number of generations.

The result for SD2 is presented in Table 12. The values shown in the results are the averages for 100 experiments. In these experiments, we limited the number of threads to 4 and 8 and set the search termination point to 100,000 generations.

Table 12 The result on increasing the number of individuals (SD2).

#Individuals	Count [%]	Average Gen.	Exec. time	Best Gen.
100	100	8,641	11s 63	644
150	100	5,992	12s 12	243
200	100	7,115	19s 20	229
300	100	9,441	38s 29	123
400	98	15,441	84s 76	86

5.4 Minimum Number of Generations

To estimate the performance in the case that the initial value dependence problem has been solved, we determined the minimum numbers of generations and the execution times required to solve SD1 through SD3 (Table 13).

Table 13 The minimum numbers of generations and the execution times required to solving SD1 through SD3.

Sudoku	Minimum Gen.	Exec. time
SD1	83	25 ms
SD2	158	47 ms
SD3	198	76 ms

6 Discussion

6.1 Scalability and the Dependence on Initial Values

From Table 6 through Table 8, we can see that increasing the number of threads reduces the execution time and increases the correct solution rate. Furthermore, we can see that the reduction rate of the average number of generations and average execution time with respect to execution by one thread decreases as the number of threads increases. In other words, the problem of initial value dependence tends to be eliminated as the number of threads is increased for both the processing time and the correct solution rate.

On the other hand, in the case of GPU, from Table 9 through Table 11, we can see that increasing the number of SM reduces the execution time and increases the correct solution rate. In other words, the problem of initial value dependence tends to be eliminated as the number of SM is increased for both the processing time and the correct solution rate. From Table 13, increasing the number of SM or any other means of solving the initial value dependence problem makes it fully possible to solve super-difficult Sudoku puzzles within one second in a stable manner by parallelization of evolutionary computation using a GPU.

In other words, we consider that the performance of a multi-core processor is scalable in relation to number of threads and that the performance of a GPU is scalable in relation to the number of SMs. Parallelization using the GTX 460 GPU finds solutions faster than that using Core i7 multi-core processor, but we consider this to be due simply to a difference in number of cores. On the other hand, Core i7 exhibits higher solution rates, which we think are due to the fact that random numbers in GTX 460 are generated using the CURAND library function.

6.2 Setting the Number of Individuals

Generally, we can consider that the effect of parallelization will become large as the number of individuals increases. From Table 12, in case of the Core i7, increasing the total number of individuals increased the number of individuals that came closer to the correct Sudoku solution but also increased the number that deviated from the correct solution. This is considered to be the reason why the average number of generations until the correct solution was obtained also increased. Increasing the number of individuals also increased the processing time for one generation thereby increasing the average execution time until the correct solution was obtained. The value of best generations also decreased. Accordingly, if the number of cores in the processor can be increased and processing performance by parallelization increased in future multi-core processors, we can expect processing to accelerate to the point where it will be possible to derive correct solutions for even super-difficult Sudoku problems in less than a few seconds.

In the processing-acceleration technique by GPU that we have been developing in parallel with the above technique, programming must take into account upper limits such as task parallelization and memory capacity based on the hardware specifications of the GPU to be used. There is therefore a limit as to how far the number of individuals can be increased to solve the problem of initial value dependence. On the other hand, the technique introduced in this paper, while inferior to the GPU technique in terms of parallelization, enables a parallel program to be executed without limitations in number of threads or memory capacity by virtue of using general-purpose multi-core processors. Looking forward, we believe that accelerating evolutionary computations for solving Sudoku puzzles by a highly-parallel system should be effective for either GPUs or multi-core processors while solving the problem of initial value dependence by increasing the number of individuals.

On the other hand, in the case of GTX 460, the amount of data allocated to the shared memory as described in Section 4.2.1 is equal to 249N, so the maximum number of individuals for which data can be stored in the 48 KB shared memory is 192. Furthermore, considering that 192 is a multiple of the number processors within the SM and also a multiple of the CUDA thread processing unit, the execution time and the correct solution rates for when the number of individuals is set to 192 are presented in Table 14.

Table 14 The execution time and the correct solution rates for when the number of individuals is set to 192.

Sudoku	Count [%]	Average Gen.	CPU time
SD1	100	9,072	2s 751
SD2	100	13,481	4s 530
SD3	100	22,799	6s 862

Compared with the case in which the number of individuals is set to 150, the processing time was reduced by approximately 5% while the correct solution rate remained at 100%. For SD2, the correct solution rate increased from 97% to 100% and the processing time decreased by approximately 29%. For SD3, the correct solution rate increased from 90% to 100% and the processing time decreased by approximately 21%. From this data, we can see that setting the number of individuals to an appropriate value for parallel execution of a evolutionary computation program written in C on a GPU accelerates the processing relative to processing on a CPU by a factor of 25.5 for the SD1 problem, by a factor of 19.1 for SD2, and by a factor of 16.2 for SD3. We can also see that a correct solution rate of 100% can be attained, even for problems in all three of the super-hard categories.

It can therefore be seen that super-difficult Sudoku problems can be solved in realistic times by the parallelization of evolutionary computation using the Core i7 multi-core processor commonly used in desktop PCs or the inexpensive, commercially available GTX 460 GPU.

These experiments also show that the GPU can find solutions faster than the multi-core processor by making use of a higher degree of parallelization. As new GPUs with a higher number of SMs and a higher degree of integration come to be developed, we can expect even faster execution times. At the same time, the GPU suffers from limitations such as the need for programming that must consider upper limits in task parallelization and in the memory allocated to each task due to hardware constraints. In other words, it is more difficult to use a GPU than a multi-core processor which can execute programs in parallel without having to worry about limitations in number of threads or memory capacity. Furthermore, when using a library function such as CURAND for random-number generation due to limitations in shared-memory capacity within a SM, problems with the cycle length of random numbers generated in this way must be taken into account.

Thus, when trying to decide whether to use a GPU or multi-core processor when attempting to accelerate evolutionary computations by parallelization, due consideration must be given to the target application problem.

6.3 Assessment of Fine-grained Parallelization on a Sub-chromosome Unit

The effect of acceleration on the proposed crossover and mutation methods using fine-grained parallelization of a sub-chromosome unit was assessed. Since assessment is made on an individual without parallel processing, there is allowance in the number of utilizable threads. Fig. 11 shows the relationship between mutation and thread allocation.

As shown in Fig. 11, in order to increase parallel processing speed, 9 threads were allocated to the processing of 1 individual. Two numbers were randomly chosen within a region, and swapping of the two within the 9 region blocks was processed in parallel.

Fig. 12 shows the relationship between crossovers and thread allocation. Regarding crossovers, to prevent the destruction of the valid part of the solution (building blocks), 3 rows or 3 columns consisting of region blocks were compared and the higher scores were passed onto the offspring. Since each row and column independently undergoes this process three times in parallel, 3 threads can be allocated to each individual and thus, parallel processing is achieved.

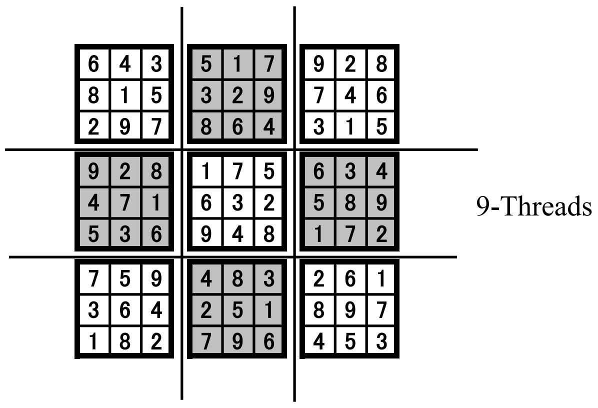


Fig. 11 Relationship between mutation and thread allocation.

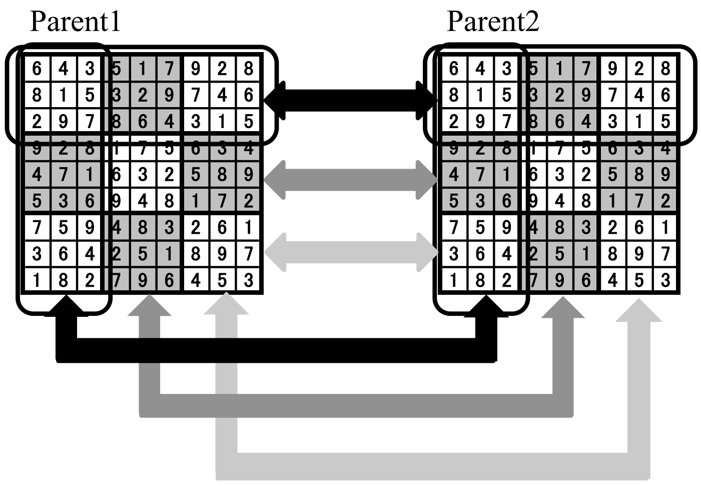


Fig. 12 Relationship between crossovers and thread allocation.

The parameters for the genetic algorithm were the same as those of the previous experiment, with the level of difficulty set at Super Difficult 3. The machine specifications used for the experiment were shown in Table 15.

Table 15 GTX 580 GPU execution environment (2).

OS	Ubuntu 10.04
CUDA Toolkit	4.0
GPU	GeForce GTX 580
CPU	Phenom II X4 945 (3GHz, 4 cores)
Main Memory	DDR2-800 4GB

Table 16 shows the fine-grained parallel processing results. Processing time equals the average generation number required to obtain the answer divided by execution time (sec). In addition, the degree of the increase in processing performance is a value relative to execution without parallel processing of individuals on a CPU.

Table 16 shows that fine-grained parallel processing of the proposed crossover method increased performance by 23%, while fine-grained parallelization of mutations without parallel processing of individuals on a GPU increased process performance by 75% when compared to execution without parallel processing of individuals on a GPU. In addition, process performance increased by approximately 4 times with parallel processing of proposed mutation and crossover methods on a GPU compared to execution without parallel processing on a CPU. The performance increase is lower compared to parallel processing on an individual level because the overhead required for thread generation is large. Further, the increase in process performance is larger with the mutation method compared to the crossover method because the number of threads, that is, the degree of parallel processing, is set to 9 and the number of processes for mutations is large. Performance increased by approximately 3 times for processing on a GPU compared to processing on CPU because a number of the same processes with different initial values are executed by 16 Streaming Multi-processors simultaneously, hence, there is less of a problem of dependence on initial values.

Our results show a modest increase in performance with fine-grained parallel processing of genetic operations such as crossovers and mutations on a sub-chromosome unit compared to parallel processing on an individual level. On the other hand, in an ideal environment, two parallel processes that are run simultaneously may be compounded to produce even faster processing. In our experiment using GTX 460, each performance increase was not compounded. The reason is the problem of computing system resources which are limited by the executable number of threads in each block. Improvement of the GPU endeavors to eliminate the lack of system resources, which will allow the execution of the two parallel processors to be multiplied, resulting in much faster processing.

Table 16 Comparison of processing time with fine-grained parallel processing.

	Processing time [sec]	Processing performance (Generation/sec)	Increase in processing performance
No parallel processing of individuals (CPU)	42.600	1282	1
No parallel processing of individuals (GPU)	17.110	3766	2.93
Parallel processing of crossovers (GPU)	14.313	4054	3.16
Parallel processing of mutations and crossovers (GPU)	10.730	5018	3.91

7 Conclusion

We have used the problem of solving Sudoku puzzles as an actual and practical problem to demonstrate that practical processing time is possible through the use of many-core processors for parallel processing of evolutionary computation. Specifically, we implemented parallel evolutionary computation on the NVIDIA GTX 460, a commercially-available GPU, or the commercially available Core i7 multi-core processor from Intel. Evaluation results showed that execution acceleration factors of from 10 to 25 relative to execution of a C program on a CPU are attained and a correct solution rate of 100% can be achieved, even for super-difficult problems. In short, we showed that the parallelization of evolutionary computation using a multi-core processor commonly used in desktop PCs or a GPU that can be purchased at low cost can be used to solve problems in realistic times, even in the case of problems for which the application of genetic algorithms has not been studied in the past because of excessive processing times.

Furthermore, fine-grained parallelization of genetic operations that take linkage into account accelerated processing by a factor of 4 relative to processing on a CPU. An increase in GPU resources will diminish the conflict of thread usage between coarse-grained parallelization on an individual level and will enable a faster processing speed.

References

1. Gordon, V.S., Whitley, D.: Serial and parallel genetic algorithms as function optimizers. In: Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann (1993) 177–183

2. Mühlenbein, H.: Parallel genetic algorithms, population genetics and combinatorial optimization. In: Proceedings of the 3rd International Conference on Genetic Algorithms. (1989) 416–421
3. Mühlenbein, H.: Evolution in time and space - the parallel genetic algorithm. In: Foundations of Genetic Algorithms, Morgan Kaufmann (1991) 316–337
4. Shonkwiler, R.: Parallel genetic algorithm. In: Proceedings of the 5th International Conference on Genetic Algorithms. (1993) 199–205
5. Cantu-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers (2000)
6. Byun, J.H., Datta, K., Ravindran, A., Mukherjee, A., Joshi, B.: Performance analysis of coarse-grained parallel genetic algorithms on the multi-core sun UltraSPARC T1. In: Southeastcon, 2009. SOUTHEASTCON '09. IEEE. (2009) 301–306
7. Serrano, R., Tapia, J., Montiel, O., Sepúlveda, R., Melin, P.: High performance parallel programming of a ga using multi-core technology. In Castillo, O., Melin, P., Kacprzyk, J., Pedrycz, W., eds.: Soft Computing for Hybrid Intelligent Systems. Volume 154 of Studies in Computational Intelligence. Springer Berlin / Heidelberg (2008) 307–314
8. Tsutsui, S., Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In: GECCO '09: Proc. 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference. (2009) 2523–2530
9. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm. In: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies. PDCAT '09 (2009) 457–462
10. Sato, M., Sato, Y., Namiki, M.: Proposal of a multi-core processor from the viewpoint of evolutionary computation. In: Proceedings of the IEEE Congress on Evolutionary Computation 2010. (July 2010) 3868–3875
11. Wikipedia: Sudoku. Available via WWW: http://en.wikipedia.org/wiki/Sudoku (cited 8.3.2010)
12. Wikipedia: Backtracking. Available via WWW: http://en.wikipedia.org/wiki/Backtracking (cited 1.11.2011)
13. IEEE: ISO/IEC 9945-1 ANSI/IEEE Std 1003.1. (1996)
14. Sato, Y., Inoue, H.: Solving sudoku with genetic operations that preserve building blocks. In: Proceedings of the IEEE COnference on Computational Intelligence in Game. (2010) 23–29
15. Lewis, R.: Metaheuristics can solve sudoku puzzles. Journal of Heuristics **13** (August 2007) 387–401
16. Simonis, H.: Sudoku as a constraint problem. In: Proc. of the 4th Int. Workshop Modelling and Reformulating Constraint Satisfaction Problems International Conference on Genetic Algorithms. (2005) 13–27
17. Lynce, I., Ouaknine, J.: Sudoku as a sat problem. In: Proceedings of the 9 th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale, Springer (2006)
18. Moon, T., Gunther, J.: Multiple constraint satisfaction by belief propagation: An example using sudoku. In: In 2006 IEEE Mountain Workshop on Adaptive and Learning Systems, Springer (2006)
19. Mantere, T., Koljonen, J.: Solving and ranking sudoku puzzles with genetic algorithms. In: Proceedings of the 12th Finnish Artificial Conference STeP 2006. (October 2006) 86–92
20. Mantere, T., Koljnen, J.: Solving, rating and generating sudoku puzzles with ga. In: Proceedings of the IEEE Congress on Evolutionary Computation 2007. (July 2007) 1382–1389
21. Moraglio, A., Togelius, J., Lucas, S.: Product geometric crossover for the sudoku puzzle. In: Proceedings of IEEE Congress on Evolutionary Computation 2006. (July 2006) 470–476
22. Galvan-Lopez, E., O'Neill, M.: On the effects of locality in a permutation problem: The sudoku puzzle. In: Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG 2009). (September 2009) 80–87

23. Goldberg, D.E., Sastry, K.: A practical schema theorem for genetic algorithm design and tuning. In: Proceedings of the 2001 Genetic and Evolutionary Computation Conference. (2001) 328–335
24. Number Place Plaza (eds.): Number place best selection 110. Cosmic mook (December 2008)
25. Super difficult Sudoku's. Available via WWW: http://lipas.uwasa.fi/ timan/sudoku/EA_ht_2008.pdf#search='CT20A6300%20Alternative%20Project%20work%202008'
(cited 8.3.2010)