2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

Parallelization and fault-tolerance of evolutionary computation on many-core processors

Yuji Sato
Faculty of Computer and Information Sciences
Hosei University
Tokyo, Japan
yuji@k.hosei.ac.jp

Mikiko Sato
Graduate School of Engineering
Tokyo University of Agriculture and Technology
Tokyo, Japan
mikiko@namikilab.tuat.ac.jp

***Abstract*—We report on fault-tolerant technology for use with high-speed parallel evolutionary computation on many-core processors. In particular, for distributed GA models which communicate between islands, we propose a method where an island's ID number is added to the header of data transferred by this island for use in fault detection, and we evaluate this method using Deceptive functions and Sudoku puzzles. As a result, we show that it is possible to detect single stuck-at faults with practically negligible overheads in applications where the time spent performing genetic operations is large compared with the data transfer speed between islands. We also show that it is still possible to obtain an optimal solution when a single stuck-at fault is assumed to have occurred, and that increasing the number of parallel threads has the effect of making the system less susceptible to faults and more sustainable.**

Keywords—Evolutionary Computation; Genetic Algorithms; Many-core Processors; Fault-tolerance; Parallelization;

I. INTRODUCTION

As an approach to speeding up evolutionary computation, the use of evolutionary computation methods that run on massively parallel computers has been actively researched since the 1990s [1, 2]. On the other hand, a recent trend has been towards the growing use of ordinary PCs with inexpensive multi-core processors (MCPs) aimed at small-scale parallelization using several cores or several tens of cores [3–5]. Research has also started on accelerating ordinary programs by using graphics processing units (GPUs) developed for the purpose of accelerating the processing of computer graphics. Against this background, the study of parallelizing evolutionary computation through the use of multi-core processors and many-core processors such as GPUs is getting under way [6–10].

On the other hand, with the growing popularity of GPU devices and the use of these devices in fields where high reliability is required, such as scientific computing and data centers, NVIDIA has developed the Tesla series of GPUs with ECC memory (error checking and correction memory) functions [11]. Since Tesla GPUs are expensive, we also propose the idea of implementing error checking functions in software based on inexpensive GPUs such as the GeForce series [12] that do not support fault-tolerant technology.

In this ECC memory, additional information such as an 8-bit parity code must be added to each 64-bit data word. This means that over 10% of the installed memory must be allocated for the storage of parity codes. Although there is no problem allocating the parity codes to part of a high-capacity memory region such as the global memory (GM), this approach is unsuitable for the allocation of smaller memory regions such as registers or shared memory inside a streaming multiprocessor (SM). We can classify faults into two types: stuck-at faults and transient faults. Transient faults lack repeatability and are caused by factors such as noise. They usually cause errors in roughly one or two bits. Stuck-at faults are caused by improper initialization or deterioration, and often result in failures that extend across multiple bits. Here, the ECC memory can correct single-bit errors, and can also detect errors of two or more bits but without the ability to identify the locations of these errors. Therefore, ECC memory is effective for transient faults, but not always sufficient for stuck-at faults.

Furthermore, modern supercomputers are often configured by CPUs in conjunction with GPUs such as Titan [13] or TSUBAME 2.0 [14] in large-scale networks due to the advantages of such architectures, including their very high cost/performance ratios and low power consumption. On the other hand, while traditional supercomputers presumes a structure with a regular arrangement of modules with identical specifications, a many-core processor has a hierarchical structure and is configured as a system with a different architecture at each level. For example, a GPU consists of multiple SMs, each comprising a number of core processors. This results in a hierarchical structure where the architecture inside an SM differs from the architecture between SMs. Against this background, our aim in this study is to clarify the fault-tolerant performance achieved when evolutionary computation is performed by fast parallel processing on many-core processors such as GPUs, and to propose a fault-tolerant technology for increasing the durability of application programs.

In section 2 of this paper, we present an overview of the fault-tolerant abilities of the independent competition model. In section 3, we propose an island-model fault-tolerant technology that takes communication into consideration. In section 4, we perform evaluation tests of a single stuck-at fault, and finally we conclude with a discussion and summary.

II. FAULT-TOLERANCEE OF THE INDEPENDENT COMPETITION MODEL

A. Independent competition model

As an example, Figure 1 illustrates the basic architecture of Nvidia's GTX460 GPU, and the method used to implement the parallel evolutionary computation model. The GTX460 consists of 7 SMs and a large (1 GB) global memory. Each SM has 48 core processors and a small (48 KB) shared memory. In each SM, it is possible to define up to 1536 threads.

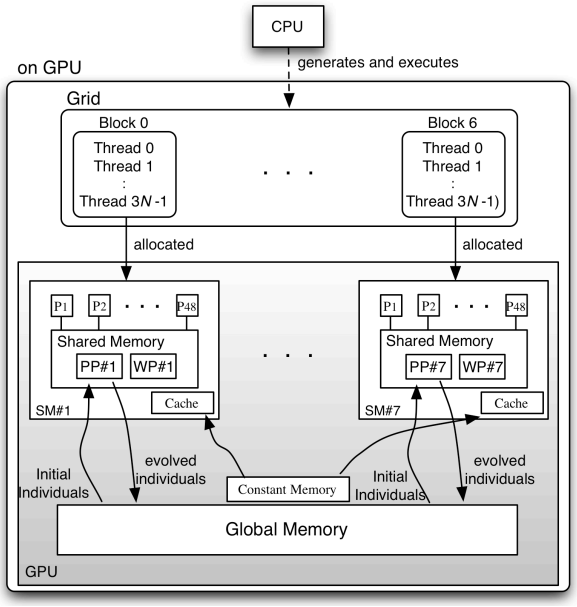

Fig. 1. GTX460 configuration and implementation of the parallel processing model for evolutionary computation.

When distributed GA is implemented on a GPU, it is generally done using an island model where an SM corresponds to a single island. In the island model, the alternation of generations is repeated by performing genetic operations on each island, and co-evolution is performed while exchanging elite individuals and other data between the islands at fixed intervals. However, although communication inside an SM can be performed at high speed, communication between SMs is performed via global memory and is about 100 times slower. Other reports have therefore discussed an "independent competition model" where data is not exchanged between SMs. In this model where the same GA is run independently in each SM by changing the random initial value, and the program is terminated when an SM that has obtained a solution is found.

B. Comparative evaluation of the independent competition model and the conventional technique [15]

A number of fault-tolerant and enhanced reliability technologies have hitherto been proposed [16-19], principally for applications such as computers and LSIs, and there are also a good number of practical examples such as using multiplexing techniques to make computers more reliable, or using redundancy techniques to improve the yield of semiconductor memory devices. We will consider an example of a multiplexing scheme and a stand-by redundancy system here [16]. Because of these conventional techniques, most of the ones that use redundancy are centered on techniques that presume a multiple module configuration, and are considered suitable for installation on many-core environments comprising multiple core processors. We performed an evaluation in which stuck-at faults and transient faults were assumed to occur.

1) Evaluation method: We performed an evaluation in which stuck-at faults and transient faults were assumed to occur. For stuck-at faults, the GTX460 was used to simulate a physical fault such as a fault in the shared memory inside the SM or in the path connecting a core processor to the shared memory. In the SM where a fault has occurred, it is considered that correct genetic operations are prevented from running, and a correct solution cannot be found. With an SM regarded as a single module, we investigated the reliability and performance (execution time needed to obtain a correct solution) of three methods — multiplexing, stand-by redundancy, and parallel evolutionary computing. The reliability comparisons were performed by comparing the relationships between time and reliability based on Equations (1), (2) and (3), with the number of modules fixed at N.

Eq. (1), (2) and (3) indicate the reliability R_{nmr} of the N-Modular Redundancy (NMR) system, the reliability R_{sb} of a stand-by redundancy system, and the reliability R_p of the independent competitive model, respectively. Here, R_m, R_v and R_s represent the reliability of a single module, the reliability of the voting element, and the reliability of the switching circuit, respectively. if it is assumed that the early failure period has elapsed and the system has entered the period of fixed failure rate, then the failure rate R_m of a single module is given by $R_m=e^{-\lambda}$, where λ is the fixed failure rate.

$$R_{nmr} = R_v \sum_{i=0}^{(N-1)/2} \binom{N}{i} \left(1 - R_m\right)^i R_m^{(N-i)} \tag{1}$$

$$R_{sb} = \left\{1 - \left(1 - R_m\right)^N\right\} R_s \tag{2}$$

$$R_p = \left\{1 - \left(1 - R_m\right)^N\right\} \tag{3}$$

The performance comparisons were made by investigating the execution time taken to obtain a correct solution while varying the number of SMs used in the experiment from 1 to 7 (i.e., while varying the number of faulty SMs from 6 down to 0). The evaluation was performed using a Sudoku solver program based on evolutionary computation [20].

For transient faults, it is assumed that the faults consist of temporary non-repeating changes to data values inside each SM due to the effects of noise and the like. In the evaluation, these faults were assumed to manifest as errors whereby the IDs of parent individuals are randomly switched during the

selection phase. The experiment was performed using an Intel Core i7 processor, with the error frequency varied as a parameter. Evaluations were performed using the knapsack problem which restricts the number 40 and Rosenbrock function (4) below, which was chosen from the five types of function minimization problems shown by De Jong. We used the tournament selection and the parameters used for genetic manipulation in these evaluation tests are shown in Tables I.

$$F_2 = 100\left(x_1^2 - x_2\right)^2 + \left(1 - x_1\right)^2 \quad (4)$$

TABLE I. PARAMETERS FOR GENETIC MANIPULATION

Pop. size	Max. generation	Tournament size	Cross. rate	Mutation rate
100	30,000	4	0.7	0.01

2) Evaluation of results for stuck-at faults: We assume single stuck-at fault model here.

a) Desktop evaluation of reliability: Figure 2 shows the relationship between reliability and time t for a single module, multiplexing, stand-by redundancy and the evolutionary computation model. Although this figure shows the results for a single module failure rate of λ=0.001, there is no change in the overall trends for different values of λ. The reliability R_{sb} of stand-by redundancy is calculated by assuming a fixed value of 0.9 for the switching circuit reliability R_s. In fact, the initial value of R_s is closer to 1, but considering that there is the possibility of a fault occurring in the fault detector circuit and that the value of R_s also decays over time, we performed these calculations with a fixed value of 0.9 for the sake of convenience. The reliability R_p of the parallelized evolutionary computation model maintains the highest value throughout the entire period, and in terms of reliability it seems that this is an effective approach to achieving fault-tolerance in many-core architectures such as GPUs. It also has the advantage of making it unnecessary to add extra hardware such as voting circuits and switching circuits.

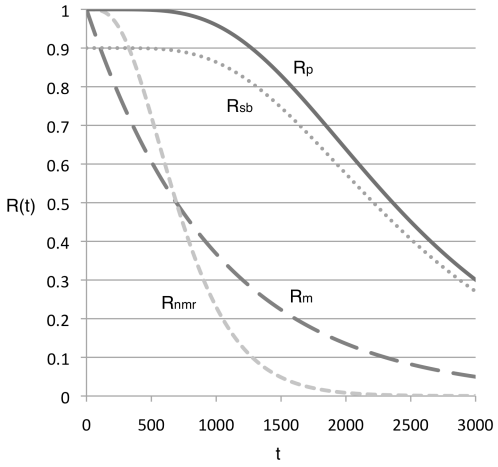

Fig. 2. Variation of the reliability of fault-tolerant techniques with time t.

b) Comparative evaluation of performance: Table 2 shows the relationship between the number of SMs and the performance achieved when evolutionary computation for solving Sudoku puzzles [21] is performed in parallel on a GTX460 GPU. In Table 2, while varying the number of SMs used, we calculated the number of times a correct solution was obtained out of 100 attempts where the processes truncated at 100,000 generations (Count), the average number of generations needed to obtain a correct solution, and the computation time. However, with no set truncation point, a correct answer would have been obtained in all cases, even with just one SM. Also, Fig. 3 shows how the execution time varies with the number of SMs in the multiplexing scheme and the parallelized evolutionary computation model. The execution times of the evolutionary computation model are the measured values shown in Table II, and the execution times of the multiplexing scheme are the theoretical (lower bound) values for the ideal case where the time taken up by the voting logic is ignored. The execution time of the stand-by redundancy scheme was more or less the same as for the multiplexing scheme, and was constant as expected. The difference in execution times between the multiplexing and stand-by redundancy schemes corresponds to the difference in time needed to perform fault detection and switching and the time needed for the voting logic, and their relative merits depend on how they are implemented. In the multiplexing and stand-by redundancy schemes, the execution time is constant regardless of the number of SMs, whereas the parallelized evolutionary computation model has the advantage that the execution time needed to search for a solution decreases as the number of SMs increases.

TABLE II. THE NUMBER OF GENERATIONS UNTIL THE CORRECT SOLUTION WAS OBTAINED, THE EXECUTION TIME, AND THE RATE OF CORRECT ANSWERS (SD1 [19], GIVENS: 24)

	Average Gen.	Execution time	Count [%]
#SM: 1	57,687	16s 728	62
#SM: 2	40,820	11s 845	80
#SM: 3	30,053	8s 733	89
#SM: 4	19,020	5s 527	98
#SM: 5	13,718	3s 985	97
#SM: 6	12,037	3s 495	99
#SM: 7	10,014	2s 906	100

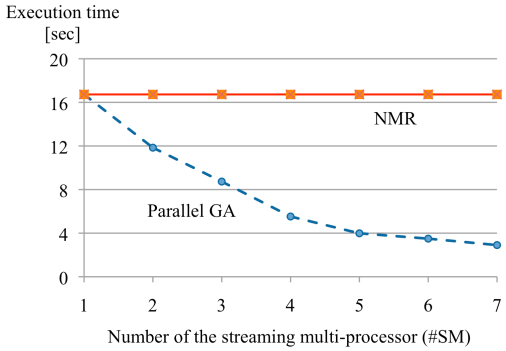

Fig. 3. Variation of execution time with number of SMs.

3) Evaluation of results for transient faults: Figure 4 shows how the average number of generations needed to search for the minimum value of the function shown in Equation (4) varies with the number of threads. Figure 5 shows how the average number of generations needed to search for the solution of the knapsack problem which restricts the number 40. Here, it can be seen that the average number of generations it takes to find a solution tends to increase gradually as the transient fault probability increases, regardless of how many threads are being used. However, in each case a solution was still obtained despite the addition of transient faults. Transient faults can also be thought to play a role in increasing diversity in the GA search process, and GA is thought to be intrinsically less susceptible to the adverse effects of transient errors. Also, regardless of the probability of transient errors, it can be seen that the number of generations needed to find a solution decreases as the number of threads is increased (i.e., with increasing parallelism). It can thus be seen that parallelization of evolutionary computation in a many-core environment is not only robust against stuck-at faults but also against transient faults.

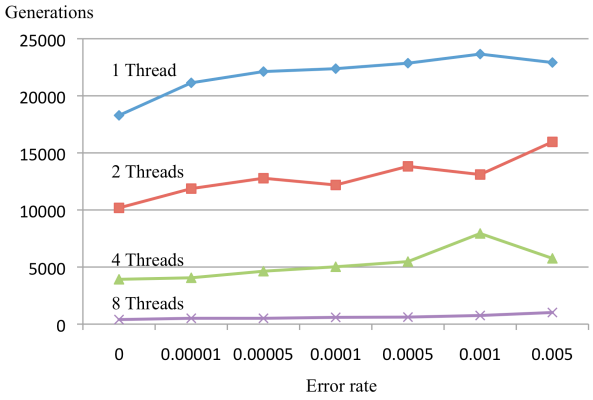

Fig. 4. Average number of generations needed to find the minimum value of Rosenbrock function.

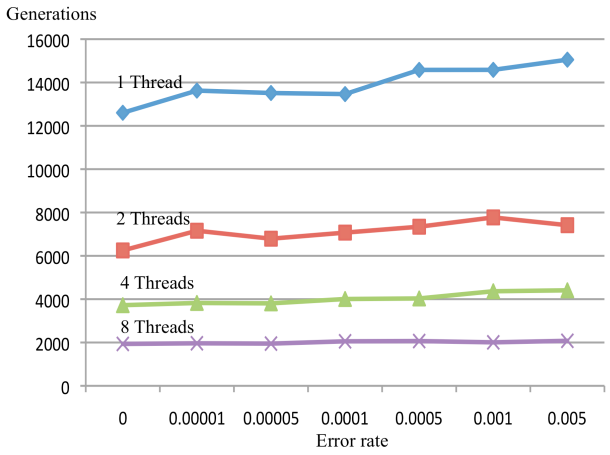

Fig. 5. Average number of generations needed to find the solution of the knapsack problem.

4) Discussion for the independent competition model:

From the above, when parallel evolutionary computation is performed in many-core processors using a scheme based on independent competition, it seems that benefits such as higher reliability and lower susceptibility to transient errors can be achieved compared with when using conventional fault-tolerant techniques. There is also no need for extra hardware such as voting circuits or switching circuits. It is also thought to be effective at improving the performance as the degree of multiplicity or redundancy is increased. Although the performance decreases as the number of faulty physical SMs increases, a correct result can still be obtained even when there is only one functioning SM remaining, so it is though that this approach offers enhanced sustainability by increasing the performance of application programs running on the GPU and allowing application programs to continue running due to the increased fault tolerance.

On the other hand, in the ordinary island model, it is impossible to guarantee a correct result when a stuck-at fault occurs in an interconnection (path) between a GM and an SM used for communication between islands. Therefore, in the following section we propose a fault detection method for island models that takes communication into consideration.

III. An Island-Model Fault-Tolerant Technique That Takes Communication Into Consideration

A. Basic idea

In a GPU, the transfer of data between SMs is generally specified as being performed in byte units. On the other hand, the data to be exchanged does not normally correspond to an exact number of byte units, so the GPU system pads out the data with random bits to form a whole number of bytes for transfer between the SMs. Our basic idea is to perform error detection by making using these left over bits when data is transferred between SMs.

B. Proposed implementation

Figure 6 shows an example of how this idea can be implemented. In an island model, it is usually determined in advance which island will transfer data to which island. Here is an example of an implementation where a single island corresponds to a single SM. In this example, data is transferred between SMs with an added fault detection header that uses the bits that are left over when the data is converted into byte units, and each SM stores a cyclic list showing the predetermined order in which data is sent to it from the other SMs. Faults can then be detected by checking whether or not the header information in the received data matches the information recorded in the cyclic list. When a fault is detected, the genetic operations are continued without accepting the received data. Figure 6 shows an example where data is transferred from SM #000 to SM #001. When sending the data "data0" in SM #000 to SM #001, the bits that are left over when it is converted into byte units are used to transmit the data with an added header "000". In SM #001, the header information of this data is compared with the pre-recorded number of the SM from which

data is transferred in order to check whether or not a fault has occurred.

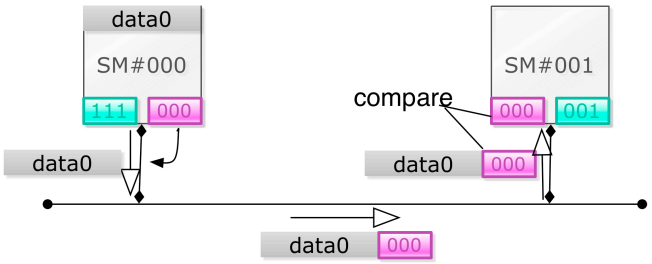

Fig. 6. Conceptual illustration of the fault-tolerant technique for island models.

The target of fault detection is assumed to be single stuck-at faults in the SMs or on the network. When there is a fault in the SM on the transmitting side, errors occur in the transferred header data so that it fails to match the information in the cyclic list registered in the receiving SM. As a result, the fault can be detected and the number of the SM where this fault occurred can be reported. When there is a fault in the network, this can be detected when a fault has occurred at the header insertion location. It is thus possible to detect a single fault at any location on the network by shifting the header insertion position each time data is transferred.

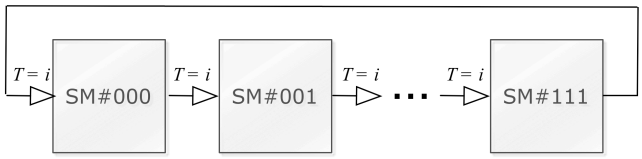

Fig. 7. Example of a method where sequential data is transferred between SMs on adjacent islands.

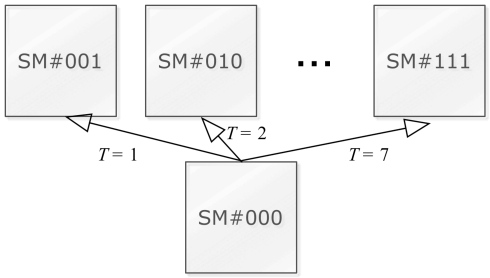

Fig. 8. Example of a method where the destination to which data is transferred is changed at each transfer so as to make a full circuit.

Next, we will consider a method for producing the pre-prepared cyclic list in each SM's shared memory and constant memory. Figures 7 and 8 show an example of a communication method in an island model. Figure 7 shows the method for transferring sequential data to neighboring islands (SM), whereby data is transmitted simultaneously from each SM to the neighboring SM. Figure 8 shows the method whereby data is sent by changing the transfer destination at each transfer so as to make a full circuit. In these methods, the sequence set by the designed is recorded in the SMs as a cyclic list. On the other hand, the transfer destination SMs can be randomized by having the host CPU generate random numbers and transfer a list of random numbers in advance to each SM where it is stored as a cyclic list.

The amount of data transferred does not increase if information is inserted in the regions left over after conversion into byte units. However, when there is insufficient space for header insertion, an extra byte has to be added, resulting in an increased data transfer overhead. In the following experiment, we performed a computer simulation to evaluate how the processing time is affected by the proposed method.

IV. Evaluation Experiment

A. Evaluation using Sudoku puzzles

1) Evaluation method: We investigated the overhead for fault detection using an AMD Opteron Quad Core 2380 multi-core processor. Experiments were performed in a system configured with 8 core processors by connecting two Opteron Quad Core 2380 processors together. Evaluations were performed with a single island corresponding to a single thread, and with up to 8 threads. We investigated the overheads obtained while varying the frequency of transfers and the number of threads used, and we studied whether or not an optimal solution could be obtained in the event of a single stuck-at fault and, if so, whether or not there was an increase in the number of generations needed to converge on a solution. We considered two different transfer methods — one in which sequential data was transferred to neighboring islands (threads), and another in which the transfer destinations were changed at each transfer so as to make a complete circuit. As problems for evaluation, we used a Sudoku solution method [10] as an example of a practical problem.

2) Experimental results:

a) Investigation of overheads: Figures 9 to 11 show the relationship between the number of generations between data transfers, the number of generations needed to reach an optimal solution, and the processing performance per generation for solving Sudoku puzzles [21]. The bar graph shows the number of generations until convergence, and the values are shown on the left side. The line graph shows the processing performance, with the number of operations per second shown on the right side. Figures 9 to 11 show the results obtained with 2, 4, and 8 threads respectively. The two bar graphs for each transfer interval represent ordinary migration from left to right, and migration where fault detection headers are inserted. Both sets of results were obtained in experiments with the same initial population. The tests were run 100 times and the averages were compared.

As these Figures show, the overheads incurred by implementing the proposed idea are negligibly small, regardless of the data transfer interval, and for any number of threads.

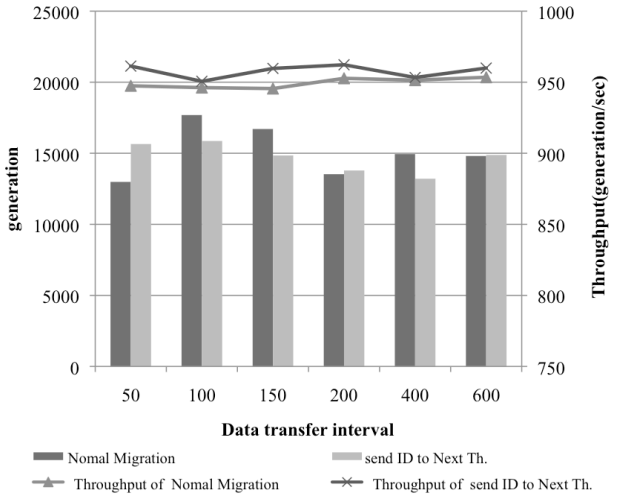

Fig. 9. Data transfer interval (number of generations) vs. number of generations needed to reach an optimal solution and processing performance per generation (number of threads: 2).

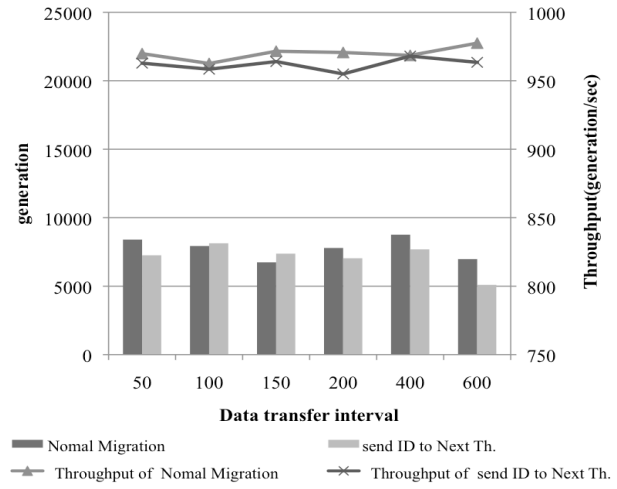

Fig. 10. Data transfer interval (number of generations) vs. number of generations needed to reach an optimal solution and processing performance per generation (number of threads: 4).

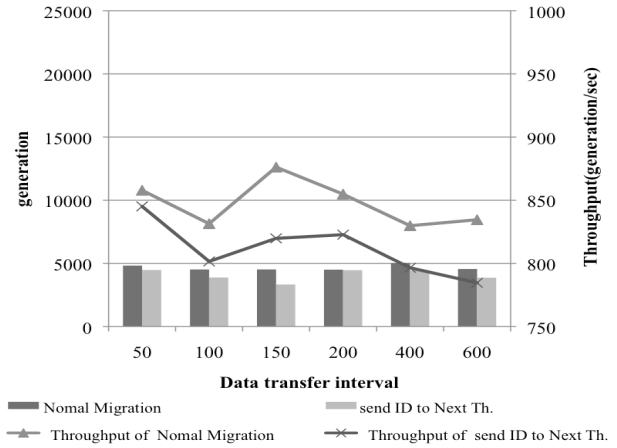

Fig. 11. Data transfer interval (number of generations) vs. number of generations needed to reach an optimal solution and processing performance per generation (number of threads: 8).

b) Evaluation assuming a single stuck-at fault: Figures 12 and 13 show the relationship between the number of threads and the number of generations needed to obtain an optimal solution when assuming a single stuck-at fault. The three bar graphs for each thread correspond to (from left to right) ordinary migration, migration with fault detection headers inserted, and migration with fault detection headers and a single stuck-at fault. Figure 12 shows the results of a method where sequential data is transferred to a neighboring thread ("Next" method), and Fig. 13 shows the results of a method where the transfer destination is changed at each transfer so as to make a complete circuit ("Round" method). In both methods, the results obtained with a data transfer interval of 200 generations are shown. The tests were run 100 times and the averages of the results were compared.

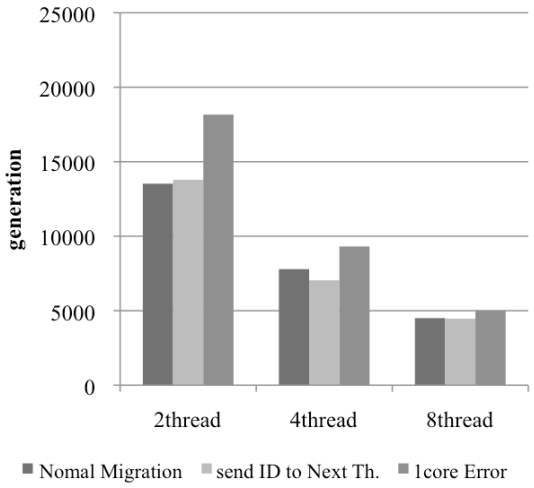

Fig. 12. Relationship between number of threads and number of generations needed to find a solution when assuming a single stuck-at fault (1) ("Next" method, data transfer interval: 200 generations).

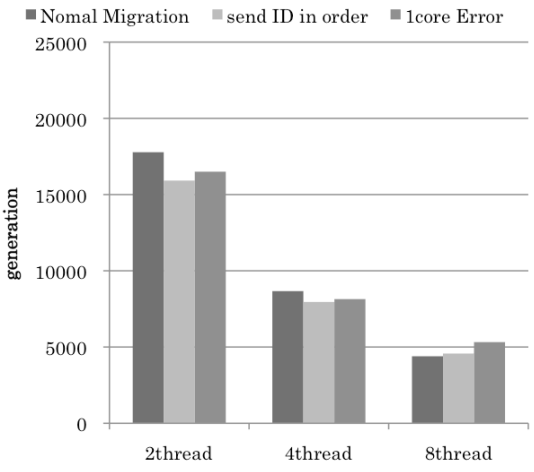

Fig. 13. Relationship between number of threads and number of generations needed to find a solution when assuming a single stuck-at fault (2) ("Round" method, data transfer interval: 200 generations).

In both Figs. 12 and 13, when there are two threads and we assume a fault in one thread, then the number of generations needed for convergence increases but it is still possible to obtain an optimal solution. With four or eight threads, the assumption of a fault in one thread has hardly any effect on the ability to find a solution.

B. Evaluation using Deceptive functions

1) Evaluation method: We investigated the overhead for fault detection using a Nvidia's Geforce GTX 680 GPU. Experiments were performed in a system configured with 8 SMs and evaluations were performed with a single island corresponding to a SM. We investigated the overheads obtained when an extra byte has to be added for header insertion, and we studied whether or not an optimal solution could be obtained in the event of a single stuck-at fault and, if so, whether or not there was an increase in the execution time needed to converge on a solution. We considered the transfer method in which sequential data was transferred to neighboring islands (SMs). We used s simple GA with inversion mutation and the parameters used for genetic manipulation in these evaluation tests are shown in Tables III.

TABLE III. PARAMETERS FOR GENETIC MANIPULATION

Pop. Size	Max. gen.	Crossover rate	Mutation rate	Data transfer interval
50/SM	2,000	0.7	0.03	25

2) Deceptive Functions: As problems for evaluation, 16-copies of 4-bit fully deceptive trap functions [22] shown in Equation (5) was used.

$$f(X) = \sum_{i=1}^{16} f_1(xI_i) \tag{5}$$

where the function f of the string X is taken as the sum of the sub-function $f_1()$ which is the 4-bit trap function shown in Figure 14.

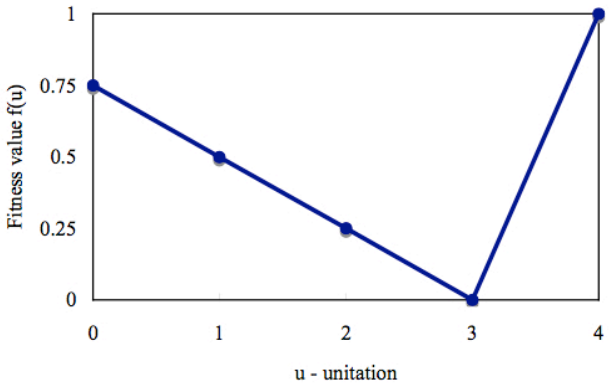

Fig. 14. Single 4-bit trap function used as a building block for the 64-bit test function.

3) Experimental results: In this experiment, the original data length is 64-bits (8-bytes) and there is no space for header insertion, an extra byte has to be added. Table IV shows the overheads obtained when an extra bits has to be added for header insertion. The three data correspond to (from left to right) ordinary migration, migration with fault detection headers (3-bits) and a single stuck-at fault, and migration with fault detection headers (8-bits) and a single stuck-at fault. Whole sets of results were obtained in experiments with the same initial population.

TABLE IV. HEADER LENGTH AND ECESUTION TIME

Header length	0-bits	3-bits	8-bits
Execution time [sec]	0.46	0.52	0.53

As this Table shows, the overheads incurred by implementing the proposed idea are about 13%, for any bit length of fault detection headers, but it is still possible to obtain an optimal solution.

C. Discussion

From Figures 9 to 11, we did not observe any pronounced drop in performance when implementing the proposed idea, regardless of the data transfer interval. This is thought to be because the amount of data transferred does not increase when fault detection header is inserted in the regions left over after conversion into byte units and the processing time taken up by the proposed idea is negligible compared to the variation of convergence times obtained when using random numbers for processes such as genetic operations. On the other hand, Table IV shows the execution time was about 13% increased with fault detection headers and a single stuck-at fault. It is therefore considered that in applications where the processing time dedicated to genetic operations is large relative to the speed of data transfers between islands, the processing time of the proposed idea is practically negligible.

Figures 12, 13 and Table IV also indicate the proposed idea allows an optimal solution to be obtained even when a single stuck-at fault is assumed, and that the system becomes less susceptible to the effects of a fault as the number of parallel threads increases. Therefore, the proposed method is thought to offer real benefits to sustainable systems where there is a low likelihood of applications being interrupted when a physical fault occurs.

As related previous work, Y. Gong and A.S. Fukunaga presented the effects of communication failures in grid-based distributed genetic algorithms [23]. Their paper investigated the effects of communication failures in loosely coupled systems. More exactly, they assumed communications between processes in a GA are asynchronous and a persistent failures on account of broken wires. Then, they proposed two recovery methods "retry" and "reroute" for the node that couldn't receive a prospected data. On the other hand, we investigated the sustainability of parallelized genetic algorithms on tightly coupled systems such as many-core

processors. We assumed communications between processes in a GA are synchronous and stuck-at faults (Individual signals are assumed to be stuck at Logical '1' or '0'). This means that all of nodes have a possibility to receive the faulty data. Therefore, we have proposed a method where an island's ID number is added to the header of data transferred by this island for use in fault detection. In the future, it will be necessary to evaluate large-scale systems where multiple GPUs are networked together.

V. CONCLUSION

In this paper, we have investigated fault-tolerant technology for cases where evolutionary computation is performed at high speed in parallel on many-core processors. In particular, for an island model where communication is performed between islands, we proposed a method where the ID number of an island is added to the header of data transferred by the island for use in fault detection, and we evaluated this method using deceptive functions and Sudoku puzzles. As a result, we have shown that the processing time of the proposed idea is practically negligible in applications where the processing time used for genetic operations is large compared with the speed of data transfer between islands. We have also shown that an optimal solution can be obtained even with a single stuck-at fault, and that increasing the number of parallel threads makes the system less susceptible to faults.

ACKNOWLEDGMENT

This research is partly supported by the collaborative research program 2012, Information Initiative Center, Hokkaido University.

REFERENCES

[1] Mühlenbein, H.: Evolution in time and space - the parallel genetic algorithm. In Foundations of Genetic Algorithms, pp. 316–337. Morgan Kaufmann (1991).

[2] Shonkwiler, R.: Parallel genetic algorithm. In Proc. of the 5th International Conference on Genetic Algorithms, pp. 199–205 (1993).

[3] Pham, D., Asano, S., Bolliger, M., Day, M. N., Hofstee, H. P., Johns, C., Kahle, J., Kameyama, A, Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., and Yazawa, K.: The design and implementation of a first-generation CELL processor. In 2005 IEEE International Solid- State Circuits Conference, vol. 1, pp. 184–592 (2005).

[4] Shiota, T., Kawasaki, K., Kawabe, Y., Shibamoto, W., Sato, A., Hashimoto, T., Hayakawa, F., Tago, S., Okano, H., Nakamura, Y., Miyake, H., Suga, A., and Takahashi, H.: A 51.2 gops 1.0 gb/sdma single-chip multi-processor integrating quadruple 8-way vliw processors. In 2005 IEEE International Solid-State Circuits Conference, vol. 1, pp. 194–593 (2005).

[5] Torii, S., et al.: A 600mips 120mw 70ua leakage triple-cpu mobile application processor chip. In the IEEE ISSCC Digest of Technical Papers, pp. 136–137 (2005).

[6] Byun, J.-H., Datta, K., Ravindran, A., Mukherjee, A., and Joshi, B.: Performance analysis of coarse-grained parallel genetic algorithms on the multi-core sun UltraSPARC T1. In SOUTHEASTCON'09. IEEE, pp. 301–306 (2009).

[7] Serrano, R., Tapia, J., Montiel, O., Sep´ulveda, R., and Melin, P.: High performance parallel programming of a GA using multi-core technology. In Soft Computing for Hybrid Intelligent Systems, pp. 307–314 (2008).

[8] Tsutsui, S., and Fujimoto, N.: Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In Proceedings of the 2009 ACM/SIGEVO Genetic and Evolutionary Computation Conference, pp. 2523–2530 (2009).

[9] Sato, M., Sato, Y., and Namiki, M.: Proposal of a multi-core processor from the viewpoint of evolutionary computation. In Proceedings of the 2010 IEEE Congress on Evolutionary Computation, CD-ROM (2010).

[10] Sato, Y., Hasegawa, N., and Sato, M.: GPU Acceleration for Sudoku Solution with Genetic Operations. In Proceedings of the 2011 IEEE Congress on Evolutionary Computation, CD-ROM (2011).

[11] Tesla. Available via WWW: http://www.nvidia.com/object/tesla-supercomputing-solutions.html (cited 10. 2. 2013)

[12] Tesla GeForce. Available via WWW: http://www.nvidia.com/page/home.html (cited 10. 2. 2013)

[13] Titan (supercomputer) Available via WWW: http://en.wikipedia.org/wiki/Titan_(supercomputer) (cited 28.4.2013)

[14] Shimokawabe, T., Aoki, T., Takaki, T., Yamanaka, A., Nukada, A., Endo, T., Maruyama, N., Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011/11, 1-11. (ACM)

[15] Lara, P. K.: Fault Tolerant and Fault Testable Hardware Design. Prentice-Hall International Ltd (1985).

[16] Sato, Y.: Parallelization of Genetic Algorithms and Sustainability on Many-core Processors. In Advances in Intelligent Systems and Computing, 202, pp. 175-187, Springer (2012)

[17] Toy, W. N., and Gallaher, L. E.: Overview and architecture of 3B20D processor. Bell Syst. Tech. J., Vol. 62, No. 1, pt. 2, pp. 181-19 (1983).

[18] Bartlet, F.: The Tandem 16; A "NonStop" operating system. In The Theory and Practice of Reliable System Design (Ed. By D. P. Siewiorek and R. S. Searz), pp. 453-460 (1982).

[19] Siewiorek, D. P., et al.: A case study of C.mmp, Cm and C.Vmp: Part 1 – Experience with fault-tolerance in multiprocessor systems. ibid., pp. 1178-1199 (1978).

[20] Sato, Y., and Inoue, H.: Solving Sudoku with Genetic Operations that Preserve Building Blocks. In Procceding of the 2010 IEEE Conference on Computational Intelligence and Games, pp. 23-29 (2010).

[21] Super difficult Sudoku-1. Available via WWW: http://lipas.uwasa.fi/~timan/sudoku/EA_ht_2008.pdf#search='CT20A6300%20AIterative%20Project20work%202008' (cited 10. 2. 2013)

[22] Deb, K., and Goldberg, D. E.: Analyzing deception in trap function. In Whitley, L. D. (Ed.), Foundations of Genetic Algorithms 2, Morgan Kaufmann Publishers, pp. 93-108. (1993).

[23] Yiyuan, G., Fukunaga, A.S.: Fault tolerance in distributed genetic algorithms with tree topologies. In Proc. of the 2009 IEEE congress on Evolutionary Computation, pp. 968- 975 (2009).