Parallelization of Genetic Algorithms and Sustainability on Many-core Processors

Yuji Sato

Faculty of Computer and Information Sciences, Hosei University

3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan

yuji@k.hosei.ac.jp

Abstract. In this paper, we study and evaluate fault-tolerant technology for use in the parallel acceleration of evolutionary computation on many-core processors. Specifically, we show running evolutionary computation in parallel on a GPU results in a system that not only performs better as the number of processor cores increases, but is also robust against any physical faults (e.g., stuck-at faults) and transient faults (e.g., faults caused by noise), and makes it less likely that the application program will be interrupted while running. That is, we show that this approach is beneficial for the implementation of systems with sustainability.

Keywords: Genetic Algorithm, Many-core Processors, Fault-tolerance, Sustainability

1 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 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]. A many-core processor contains multiple core processors of the same specifications. It should therefore be possible to achieve improved fault tolerance and reliability by effectively exploiting this feature. A number of fault-tolerant and enhanced reliability technologies have hitherto been proposed, 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. 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.

On the other hand, while conventional redundant technology 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 streaming multi-processors (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. Also, for example, the GPU GTX590 consist of two GPU (GTX580) are networked together. Accordingly, there is thought to be a need for new fault-tolerant techniques for many-core processors to replace these existing techniques.

In section 2 of this paper, we present some typical conventional fault-tolerant techniques. In section 3, we investigate the relationship between fault-tolerant techniques and the parallelization of evolutionary computation on a GPU. In section 4, we experimentally evaluate situations where stuck-at faults and transient faults are assumed to occur, and finally we conclude with a summary.

2 Typical Fault-tolerant Techniques

2.1 Multiplexing

Static redundancy [11] involves using additional components to allow the effects of faults to be completely hidden (this is called "fault masking"). Typical techniques of this sort are module multiplexing and error correcting schemes, but when a fault can cause arbitrary errors to appear at the output, the fault can only be masked by a multiplexing scheme. We will consider an example of a multiplexing scheme here. To illustrate the basic concept of a multiplexing scheme, Fig. 1 shows the basic configuration of an *N*-Modular Redundancy (NMR) scheme. In this figure, the boxes labeled with M represent identical modules, from which the final output is produced via a majority voting element V.

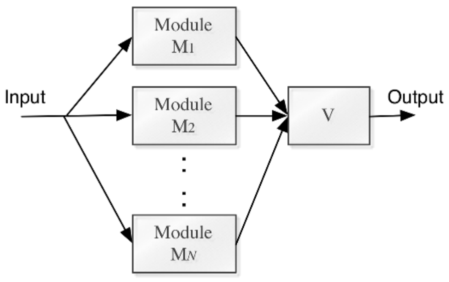

Fig. 1. Basic configuration of the multiplexing scheme

In an NMR scheme, it is possible to mask faults in up to $n = (N-1)/2$ modules. Thus, if the system reliability $R(t)$ is defined as the probability that no fault will occur before time t on condition that there are no faults in the system at time 0, then the reliability R_{nmr} of the NMR system is as follows:

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

Here, R_m and R_v represent the reliability of a single module and the reliability of the voting element, respectively. For example, when it is assumed that there is no fault in the voting element, the reliability R_{3mr} of triple modular redundancy is given by the following formula:

$$R_{3mr} = 3R_m^2 - 2R_m^3 \tag{2}$$

For each constituent module of an NMR system, 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. Substituting this value of R_m into Equation (2) yields the following formula:

$$R_{3mr} = 3e^{-2\lambda t} - 2e^{-3\lambda t} \tag{3}$$

2.2 Stand-by redundancy

In static redundancy schemes such as multiplexing, faults are masked by using redundancy, and the faults themselves continue to exist within the system. Therefore, the number of hidden faults gradually increases as the system continues to operate for a longer period of time. When the number of faults exceeds $(N-1)/2$, errors will occur in the voting output and the system will fail. To deal with this problem, dynamic redundant systems [11] have been proposed. These consist of a fault detection means and a system reconfiguration means. For example, Fig. 2 shows a conceptual diagram of a stand-by redundancy system with a similar configuration to that of the above multiplex system. The system in this figure comprises a single operating module, $N - 1$ spare modules, a fault detection means, and a switching function. When a fault is detected in the operating module, it is replaced with one of the spare modules on stand-by.

Here, assuming the ideal case of a 100% fault detection rate, the reliability R_{sb} of a stand-by redundancy system can be expressed in terms of the reliability R_m of a single module and the reliability R_s of the switching circuit as follows:

$$R_{sb} = \left\{1 - \left(1 - R_m\right)^N\right\} R_s \tag{4}$$

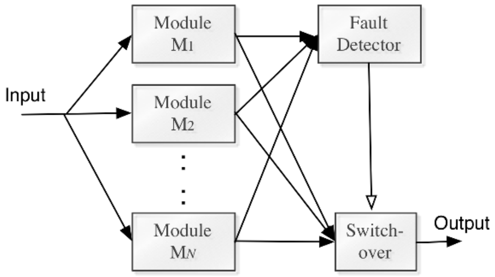

Fig. 2. Conceptual diagram of a stand-by redundancy system

3 Parallelization and Fault-tolerant Technology for EC

3.1 The problems of conventional methods

In general, multiplexing and stand-by redundancy incur costs that rise in proportion with the number of modules, so the application of these methods to real systems has chiefly involved duplex or triplex architectures. For example, the 3B20D processor developed by AT&T for electronic exchange networks [12] used duplex technology for the CPU, memory and I/O disk system. Another example is the Tandem 16 system developed by Tandem Computers for processing online transactions by organizations such as banks [13]. This was a reconfigurable multiprocessor system that achieved greater reliability by duplexing the disk devices and the buses between processors. C.vmp [14] was a multiprocessor system that could function correctly even with a mixture of permanent faults and temporary faults in the hardware. It achieved this by using a triplex configuration. These technologies are all geared towards architectures where identical modules are connected according to fixed rules. For example, if an SM is regarded as a single module, then it is suitable for implementation on a GPU.

On the other hand, in multiplexing schemes, the slowness of communication via global memory causes problems when voting circuits are implemented at the CPU end in consideration of regularity, and it is necessary to investigate how to implement voting circuits. Furthermore, it is not possible to guarantee correct results when there are faults in more than half the SMs. A problem with implementing stand-by redundancy on a GPU is the low utilization rate of the SMs that occurs as a result. Complex technology is needed to implement online processing of switching with redundant parts. When implemented with offline processing, this raises the problem of having to temporarily halt the execution of application programs. Extra hardware is needed to perform switching with redundant parts, and it is also conceivable that faults may occur in the fault detector circuits or switching circuits. Also, since a GPU has a hierarchical structure while an SM internally comprises multiple multi-core processors, the architectures inside SMs and between SMs are configured differently. We can also consider large-scale systems where multiple GPUs are networked together. Accordingly, in large-scale systems based on many-core architectures, it may become necessary to investigate new fault-tolerant technology as an alternative to conventional schemes.

3.2 Proposal of fault-tolerant technology based on parallelization of EC

The purpose of this study is to propose technology for improving the performance of active application programs and achieve greater fault tolerance by performing evolutionary computation in parallel on a GPU. As a first step, we investigate an example where the evolutionary computation of earlier studies is implemented on a GPU. Figure 3 illustrates the basic architecture of Nvidia's GTX460 GPU, and the method used to implement the parallel evolutionary computation model.

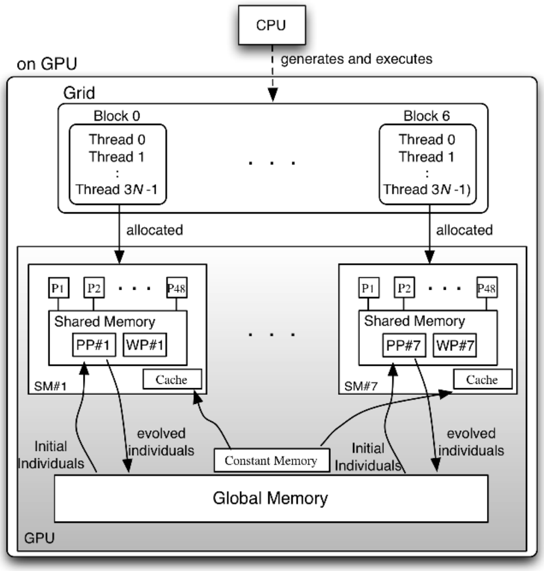

Fig. 3. GTX460 architecture and implementation of the parallel evolutionary computation model

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. Communication inside the SMs can be performed at high speed. But communication between SMs is performed via the global memory and is about 100 times slower. We therefore consider a 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. We will refer to this as an "independent competition model".

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

(1) In the host machine, all individuals (population size/SM × #SM) 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 genetic manipulation process is repeated until the termination criteria are satisfied.

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

Since there is only a small amount of shared memory inside the SM, the number of individuals that can be stored inside the SM is limited. Also, since the number of core processors in an SM is at most a dozen or so, the search performance of a single SM is not necessarily very high. However, the total population size per GPU is same as the number in the host, therefore the number of generations required to get a feasible solution will be maintained. In an independent competition model, acceleration is achieved due to the effects of parallelization.

In conventional design, the date stored in memory is the date for the fixed calculation or transaction, so the date at a faulty location will cause errors to appear at the output. Therefore, multiplexing and stand-by redundancy will be effective for the fault masking. On the other hand, in evolutionary computation, most of the data stored in memory is the genetic information of individuals, so although a gene stored at a faulty location will no longer search effectively for a solution, the populations in SMs where there are no faults will still be able to search for a solution. Therefore, since a solution can be found as long as there is one SM still operating normally, the reliability R_p of the independent competitive model is given by the following formula, where R_m is the reliability of the SM when considered as a module:

$$R_p = \left\{1-\left(1-R_m\right)^N\right\} \tag{5}$$

Compared with conventional multiplexing where it is impossible to guarantee correct results when there are faults in half or more of the SMs, this approach has higher fault tolerance and can find a solution even if there is only one SM functioning normally. Also, compared with a stand-by redundancy system, there is no need for equipment for the switching of redundant parts, thereby increasing the reliability by a factor of $1/R_s$. There is also no need for extra hardware such as voting circuits, fault detectors or switching circuits.

4 Evaluation Experiments

4.1 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), (4) and (5), with the number of modules fixed at 7. 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.

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 Equation (6) below, which was chosen from the five types of function minimization problems shown by De Jong.

$$F_2 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 \tag{6}$$

We used the tournament selection and the parameters used for genetic manipulation in these evaluation tests are shown in Tables 1.

Table 1. The parameters for genetic manipulation

Population size	Maximum generation	Tournament size	Crossover rate	Mutation rate
100	30,000	4	0.7	0.01

4.2 Experimental results and discussion

4.2.1 Evaluation of results for stuck-at faults

Comparative evaluation of reliability. Figure 4 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 $\lambda=0.001$, there is no change in the overall trends for different values of λ.

From this figure, it can be seen that the reliability R_{nmr} of multiplexing is larger than the reliability R_m of a single module up to a certain time T. The value of R_m has been observed to reverse for sufficiently large values of t, but since the reliability after a sufficiently long time had elapsed is not of major importance, it can be considered that multiplexing is effective for practical purposes. 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 value of R_{sb} starts off close to 1, and although there is slight degradation in the reliability R_{nmr} of the multiplexing scheme during the initial stage, the reliability greater than R_{nmr} tends to be maintained as the system operation time becomes longer and the number of hidden faults gradually increases. 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.

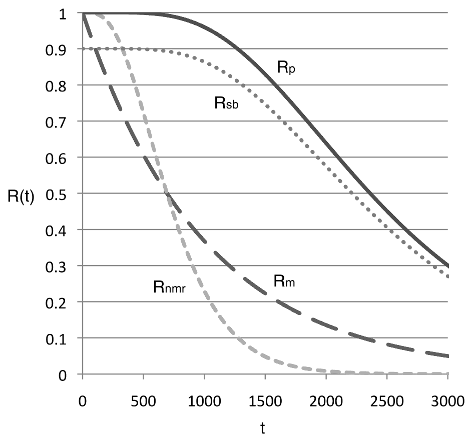

Fig. 4. Variation of the reliability of fault-tolerant techniques with time t.

Comparative evaluation of performance. Table 1 shows the relationship between the number of SMs and the performance achieved when evolutionary computation for solving Sudoku puzzles is performed in parallel on a GTX460 GPU. In Table 1, 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. 5 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 2, 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 2. The number of generations until the correct solution was obtained, the execution time, and the rate of correct answers (SD1, Givens: 24)

	Count [%]	Average Gen.	Execution time
#SM: 1	62	57,687	16s 728
#SM: 2	80	40,820	11s 845
#SM: 4	98	19,020	5s 527
#SM: 7	100	10,014	2s 906

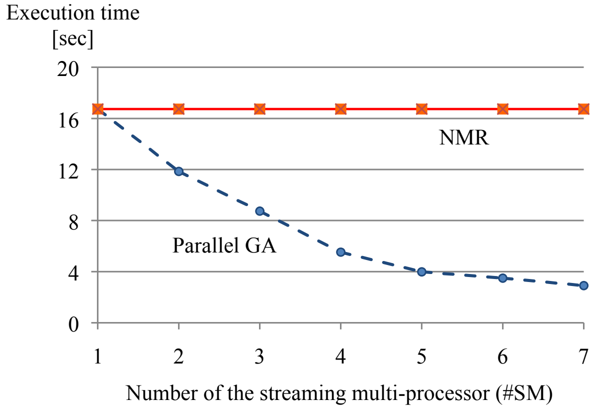

Fig. 5. Variation of execution time with number of SMs

4.2.2 Evaluation of results for transient faults

Figure 6 shows how the average number of generations needed to search for the minimum value of the function shown in Equation (6) varies with the number of threads. Figure 7 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.

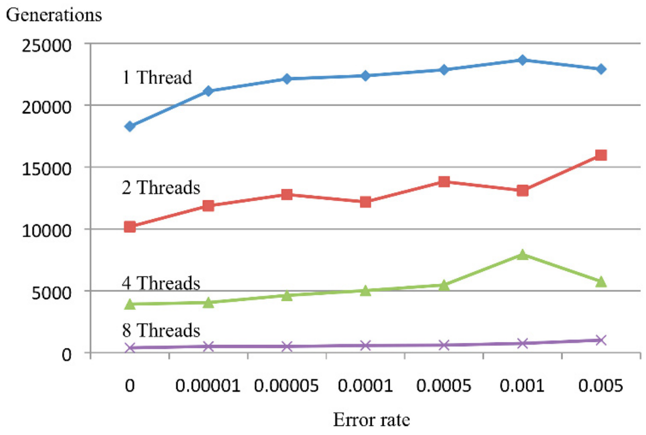

Fig. 6. Average number of generations needed to find the minimum value of a function shown in eq. (6).

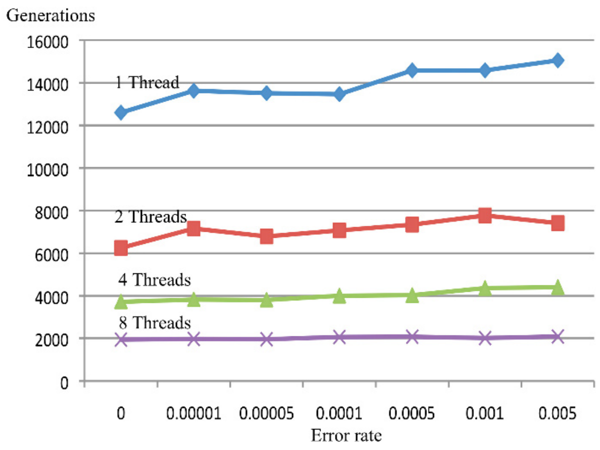

Fig. 7. Average number of generations needed to find the solution of the knapsack problem.

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.

In the future, it will be necessary to investigate the fault-tolerant performance in island models where modules communicate with each other as a parallelized evolutionary computation method using many-core processors. With regard to transient errors, it will be necessary to investigate in more detail whether there are any differences in trends due to the type of transient error or the problem dependencies. There is also a need for further research on the fault tolerance achieved when core processors inside the SMs are regarded as separate modules or when working with modules of two different types (core processors and SMs), and the fault tolerance of large-scale systems with multiple GPUs connected by a network.

5 Conclusion

In this paper, we have studied and evaluated fault-tolerant technology for use in the parallel acceleration of evolutionary computation in a many-core environment. This not only offers the benefits of parallel execution, but also acts as a fault-tolerant technology. Specifically, we compared the “independent competition model” with two conventional models, “multiplexing” and “stand-by redundancy”. As the number of physical fault locations increases, the performance declines but the functionality is maintained. Accordingly, there is less likelihood that running applications will be halted when a physical fault occurs, which is advantageous for the realization of systems with sustainability.

Acknowledgment

This research is partly supported by the collaborative research program 2012, Information Initiative Center, Hokkaido University, and a grant from the Institute for Sustainability Research and Education of Hosei University 2012

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

[12] 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)

[13] 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)

[14] 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)