A Genetic Task Migration Algorithm for Fault Recovery in NoC-based Manycore Systems

JINXIANG WANG*, ZIXU WU, FANGFA FU*
Microelectronics Center
Harbin Institute of Technology
No. 92, Xidazhi Street, Nangang District, Harbin 150001
CHINA
jxwang@hit.edu.cn, fff1984292@hit.edu.cn

Abstract: - Recovery from permanent core faults in NoC-based manycore systems usually requires migrating tasks from faulty cores to fault-free cores, where balanced workloads are commonly desired. Finding optimal migration destinations for tasks, however, is a challenging issue due to the time complexity of the search process. To cope with this, in this paper, a genetic algorithm based task migration algorithm is proposed, where the adaptive crossover (AC) scheme and the An chaotic mapping disturbance (AD) scheme are incorporated to improve the search efficiency. Experiments show that the modifications to the standard genetic algorithm (SGA) are effective, and the proposed task migration algorithm outperforms two PSO-based algorithms, the SGA and a deterministic algorithm (DA) in finding more balanced migration solutions. The average improvements of the proposed algorithm over the DA, PSO, DPSO, and SGA algorithms are 86%, 88%, 37%, and 16%, respectively.

Key-Words: - Task migration, genetic algorithm, fault recovery, NoC, manycore system

1 Introduction

The CMOS technology today has enabled hundreds of processor cores to be fabricated on a single chip [1,2] and promises the integration of thousands of cores in the near future [3]. With such large number of processor cores on chip, the traditional bus-based architecture is no longer feasible due to its lack of scalability. Typically, these chips employ the network-on-chip (NoC) communication architecture and are known as the NoC-based manycore systems. Although such systems can provide significant performance enhancement, growing power density accelerates the current-related and thermo-related failures [4]. Thus, besides the cause of manufacturing defects, permanent faults may also occur in processor cores at runtime due to the accelerated aging, which poses a great challenge for fault tolerance in manycore systems [5,6,7].

To ensure correct system functionality, faulty processor cores are usually eliminated from the system logically by migrating tasks to other fault-free cores and not assigning new tasks to the faulty cores any more. Although final migration decisions differ according to different specific application requirements, it is generally preferred that the workload on each core should be balanced before and after migration, since an unbalanced workload would cause thermal hotspots and thermal stress, making the cores age differently and resulting in shorter lifetime for the whole system [8]. Moreover, since a balanced workload is beneficial to the parallel execution of application programs, it helps to alleviate the throughput degradation due to the decrease in available processor cores. Therefore, this work focuses on finding a balanced migration solution for core failures in manycore systems. The tasks targeted in this paper are compute-intensive tasks, which have negligible communications and can be considered as independent tasks.

To find a balanced workload distribution, it seems attractive to remap all the tasks in the system [9] since this would produce the most balanced solution for the faulty system. The communication cost of remapping all tasks, however, could be extremely large, especially for manycore systems with multiple applications, and the tasks on fault-free cores might have to be halted for remapping, which causes unnecessary performance degradation. Meanwhile, the effort required to search for the best solution could grow sharply with the increasing number of processors cores [9]. Hence, this strategy is effective only when the number of tasks and processor cores is small.

Besides remapping all tasks, an alternative approach is to remap the tasks being affected by faulty cores. Although finding the most balanced workload distribution cannot be guaranteed at this time, the solution space and the searching effort are

reduced dramatically. Therefore, this approach is actually more feasible than the former one to address the task migration problem for fault tolerance in manycore systems.

Selecting destination cores for tasks to be migrated is basically a task assignment problem, which is NP-complete [10]. Hence, it is infeasible to search for the optimal exhaustively due to its time complexity. To obtain a migration solution with balanced workloads, in this paper, we present a task migration algorithm by adopting and improving the Genetic Algorithm (GA). By modifying the constant crossover probability to a parameter adjusting itself adaptively over iterations and by adding a chaotic disturbance to the best individual, our **A**daptive **C**rossover **A**n chaotic mapping **D**isturbed **G**enetic **M**igration algorithm, namely ACAD-GM, is able to generate a near-optimal migration solution efficiently. The rest of the paper is organized as follows. Section 2 presents a brief review of some relevant work. Section 3 demonstrates the targeted architecture and formulates the task migration problem based on the manycore architecture. In Section 4, the genetic task migration algorithm is described in detail. Then, in Section 5, the experiment results are discussed. Finally, Section 6 concludes the paper.

2 Related Work

As mentioned in the previous section, choosing optimal destinations for tasks is NP-complete. For the problem in this work, to assign n tasks to m processor cores, there would be m^n possible solutions. To obtain a near-optimal solution or even the optimal solution in reasonable time, efficient algorithms have been introduced in the literature.

Genetic Algorithm (GA), which is a well-known population-based stochastic search algorithm, has been adopted to solve the problem of task scheduling and mapping both in traditional parallel computing systems and in multicore and manycore systems [11,12,13]. It iteratively searches the solution space with a set of individuals and updates the population by selection, crossover and mutation operations. Because it searches multiple points simultaneously and has the capability of exploring new space, GA has a low probability to be trapped at local optima, which is essential to be used in the many-peaked search space. To guide the searching process efficiently, parameter adaptation schemes have been proposed. In [14], the difference between the average and the maximum fitnesses is used as a representation of detecting the convergence of GA. The probabilities of crossover and mutation for each individual are calculated separately according to the difference and the fitness value of the solution. Similarly, in [15], the probabilities of crossover and mutation are also designed to be changed to maintain population diversity. However, the measurement of the genetic diversity used is the ratio between the mean and the maximum values of the fitness function at each generation. Other parameters such as the mutation rate, mutation range and number of crossovers are also important to the performance of GA. A simultaneous adaptation scheme of tuning these parameters dynamically is presented in [16]. For all these GA variations, the main effort is to achieve better tradeoff between accelerating algorithm convergence and maintaining exploitation capability to new space. Although the search process can be controlled more precisely with more parameters, the cost of extra computation is increased, which should be maintained as low as possible.

Besides GA, the Particle Swarm Optimization (PSO) is another population-based algorithm [17]. It has been successfully employed to solve the complex optimization problems. It should be noted that the standard PSO operates in continuous space. Although it has been shown that solving the task assignment problem with PSO is feasible [10], the characteristic of discrete space differs a lot from that of the continuous space. To extend the capability of PSO in solving discrete optimization problems, a binary version of PSO is proposed in [18]. Although the binary PSO is effective and has been widely adopted, multi-valued numbers have to be converted to binary representations before using the approach, which requires extra processing power. To reduce the conversion cost, a discrete PSO (DPSO) is presented in [19] for the job scheduling problem. It introduces a direct encoding scheme to represent the positions of particles and derives a method for updating positions of particles without using the sigmoid transformation as in the binary PSO. The search process of PSOs is generally simpler than that of GAs, but PSOs also faces the problem of leveraging the exploration and the exploitation capacities. In addition, since parameters have great influence on the performance of PSOs, parameters must be selected carefully for efficient optimization.

3 Problem Formulation

This work focuses on providing a task migration solution when cores fail in a NoC-based manycore system. The targeted system architecture hardware consists of $M \times N$ homogeneous processor cores connected by an $M \times N$ 2D mesh NoC via on-chip

routers, as shown in Fig.1. Tasks are executed on each processor core.

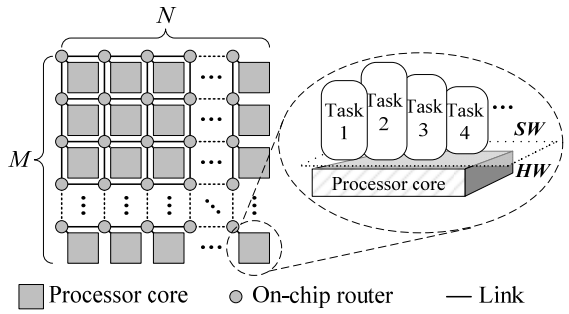

Fig.1 NoC-based manycore system architecture

Without loss of generality, the workload on each core before failures is assumed to be different to capture the execution process of tasks entering and leaving the system dynamically. Let $w(x)$ denotes the workload of x, where x is either a processor core or a task. Then the total workload on core c_i after task t_j being migrated to it can be expressed as:

$$w'(c_i) = w(c_i) + w(t_j), \quad (1)$$

where $w'(c_i)$ and $w(c_i)$ represent the workload on c_i after and before migration, respectively. Let $T=\{t_1, t_2, \ldots, t_m\}$ denotes the task set on faulty cores, where m is the total number of tasks on all faulty cores. And let $C=\{c_1, c_2, \ldots, c_n\}$ denotes the set of fault-free cores, where n is the total number of all faulty-free cores. Then the task migration problem is to find a mapping $f:T\rightarrow C$, so that each element in T is mapped to an element in C. If let ΔW_i be the extra workload introduced by the migrated tasks on c_i, then

$$\Delta W_i = \sum_{j=1}^{m} d_j \cdot w(t_j), \quad (2)$$

$$d_j = \begin{cases} 1, & f(t_j) = c_i \\ 0, & f(t_j) \neq c_i \end{cases}, \quad j \in [1, m]. \quad (3)$$

Thus, Equation (1) can be extended to

$$w'(c_i) = w(c_i) + \Delta W_i. \quad (4)$$

To obtain a balanced workload distribution on all fault-free cores after the migration, Equation (5) is calculated.

$$E = \sum_{i=1}^{n} \left[w'(c_i) - \overline{w} \right]^2, \quad (5)$$

where,

$$\overline{w} = \frac{\sum_{j=1}^{M \times N} w(c_j)}{n}. \quad (6)$$

The reason why Equation (5) is used instead of the standard deviation of workloads is that it requires less computation.

Therefore, the task migration problem in this work can be finally expressed as

Find: a solution $S \in C^m$ for $f:T\rightarrow C$,

Such that: E is minimized.

4 Genetic Migration Algorithm

In this section, a genetic migration algorithm based on the improved Genetic Algorithm is presented to solve the task migration problem.

4.1 Encoding scheme and fitness function

To solve a problem with GA, the solution of the problem must be encoded as a chromosome. For the migration problem, the chromosome is constructed as follows.

Solution S:	s_1	s_2	s_3	...	s_m
	t_1	t_2	t_3		t_m
Chromosome:	2	5	2	...	10

Mapping: $t_1 \rightarrow c_2$, $t_2 \rightarrow c_5$, $t_3 \rightarrow c_2$, $\cdots$, $t_m \rightarrow c_{10}$

Fig.2 Chromosome representation

As illustrated in Fig.2, a chromosome with m tasks to be migrated has m elements, which has the same length as a solution S to f. Similar to the meaning of each element in S, the value of each element in a chromosome represents the destination core number the task is mapped. Thus, for the chromosome exemplified in Fig.2, it can be obtained that tasks t_1 and t_3 are mapped to core c_2, while tasks t_2 and t_m are mapped to core c_5 and c_{10}, respectively.

The main objective of the task migration presented in this paper is to minimize the workload differences among all fault-free cores after failures. Since the chromosome providing smaller function value calculated with Equation (5) produces more balanced workload distribution, Equation (5) is used as the fitness function to evaluate the fitness value for each individual (i.e. each chromosome).

4.2 ACAD-GM algorithm

With the above migration solution representation and the fitness function, a near-optimal solution can be obtained by a standard GA. To improve the searching efficiency, the ACAD-GM algorithm is proposed, where the adaptive crossover (AC) scheme and the An chaotic mapping disturbance (AD) scheme are introduced. The pseudo code of the ACAD-GM algorithm is illustrated in Fig.3.

Algorithm: ACAD-GM
Input: task workload set $w(T)$;
core workload set $w(C)$
Output: migration solution
1: initialize population (N_{pop} individuals)
2: evaluate population using equation (5)
3: $g = 0$
4: **while** $g < G_{max}$ **do**
5: $g = g+1$
6: select N_{sel} individuals from the whole population using the SUS method
7: apply crossover operations to the selected N_{sel} individuals using the AC scheme
8: apply mutation operations to the individuals after crossover
9: evaluate new population using equation (5)
10: apply chaos disturbance to the individual with the highest fitness (AD scheme)
11: **return** migration solution

Fig.3 Pseudo code of ACAD-GM algorithm

To achieve a balanced workload distribution, the workload of each task in task set T and the total workload on each faulty-free core in set C before migration are provided as the input to the ACAD-GM algorithm. In the first step, N_{pop} individuals are generated randomly to form an initial population. Then, Equation (5) is used to evaluate the fitness value for each individual. For the termination criteria of GA, a maximum number of generations G_{max} is defined. Thus, the algorithm would not stop even if no better solution could be found for many consecutive generations. This helps to present the best capability that the algorithm can provide within G_{max} iterations. To enable the evolution process, some of the individuals need to be selected for mating. The selection method used in this paper is the Stochastic Universal Sampling (SUS), as it has lower computation complexity than the Roulette Wheel Selection [20] and it could provide better results [21]. The number of individuals to be selected (i.e., the N_{sel} in Fig.3) can be controlled by defining a selection rate, also known as the generation gap [22]. After the selection, the AC scheme (described in detail in Section 4.2.1) and the mutation operations are applied to the N_{sel} individuals. The newly generated N_{sel} individuals, together with the ones that have not been selected in the previous selection step, form a new population. This new population is then evaluated with the fitness function. To improve the search ability of the standard GA in the solution space, a chaotic disturbance (i.e., the AD scheme, described in detail in Section 4.2.2) has been added to the best solution found in the current generation. After G_{max} generations, the ACAD-GM algorithm produces the final migration solution.

The AC and the AD schemes are presented in the following two sections.

4.2.1 AC scheme

Crossover is an important operator and has a great influence on the performance of GAs. Since crossover creates new individuals with information from the parent individuals, high crossover rate generally yields fast population convergence, while low crossover rate maintains population diversity. In addition, the number of crossover points also affects the search process.

To improve the search ability of the standard GA, we first combine a decreased crossover rate with the adaptive crossover probability approach adopted from [14]. The combined crossover rate p_{comb} is calculated by Equation (7).

$$p_{comb} = \frac{G_{max} - g}{G_{max}} + \frac{g}{G_{max}} \cdot p_{adap}, \quad g = 1,2,...,G_{max} \quad (7)$$

where, g denotes the current number of generations, and p_{adap} is the adopted crossover probability. The construction of Equation (7) is to accelerate the convergence of the adopted approach at early generations. Then, the influence of p_{adap} becomes dominant gradually when the population evolves. The p_{adap} is given by the following equation:

$$p_{adap} = \begin{cases} \dfrac{f_{max} - f_p}{f_{max} - f_{avg}}, & f_p \geq f_{avg} \\ 1, & f_p < f_{avg} \end{cases}, \quad (8)$$

where, f_{max} and f_{avg} are the maximum and average ranked fitness values of all parents (i.e., the N_{sel} individuals), respectively. And f_p is the larger of the ranked fitness values of two parents to be crossed. The ranked fitness value of an individual here is calculated by sorting individuals of the population according to their original fitness values obtained with Equation (5) and reassigning a new fitness value for each one, so that the individual which provides better solution could have higher ranked fitness value and these ranked fitness values are limited to a fixed range throughout all iterations.

With the combined scheme, the convergence of the population is accelerated. This, however, also increases the probability of premature convergence to local optima. To overcome this problem, the number of crossover points is designed to vary with the evolution process as well. At early generations, multi-point crossover is employed to encourage the exploration of new solution space, which mitigates

the problem of premature convergence. Then, the number of crossover points decreases gradually to one as the population evolves, which helps to maintain better local exploitation, as single-point crossover is less disruptive. Since too many crossover points induce excessive computation, in this paper, 3-point crossover is chosen as the initial condition. The number selection of crossover points is described by Equation (9):

$$N_{\text{point}} = \begin{cases} 3, & g > \lfloor \frac{2}{3} \cdot G_{\max} \rfloor \\ 2, & \lfloor \frac{1}{3} \cdot G_{\max} \rfloor < g \le \lfloor \frac{2}{3} \cdot G_{\max} \rfloor \\ 1, & g \le \lfloor \frac{1}{3} \cdot G_{\max} \rfloor \end{cases}, \quad (9)$$

where, N_{point} is the number of crossover points, and g is the current number of generations as defined in Equation (7).

4.2.2 AD scheme

Recent studies have shown that chaos optimization algorithms have high efficiency in searching for the global optima [23,24]. To further improve the search ability of GA, a chaotic disturbance is added to the best individual of the current generation. The disturbance is basically a change of some values of the elements in the chromosome. Since the main operations in GA have provided an effective search method, this disturbance is designed to be as little as possible to avoid disrupting the main search process.

The An chaotic map is adopted in this paper, as it exhibits a random ergodic behavior with decreasing probabilities of producing values from 0 to 1, which defers from other chaotic maps such as the Cat map, the Logistic map and the Tent map [23]. This characteristic is helpful for improving the capability of a local exploitation since it introduces slight disturbance for most cases while providing a chance to explore further points in the solution space at the same time. The An mapping function is shown in Equation (10).

$$x_{n+1} = \begin{cases} \frac{3}{2} x_n + \frac{1}{4}, & x \in [0, 0.5) \\ \frac{1}{2} x_n - \frac{1}{4}, & x \in [0.5, 1] \end{cases}, \quad (10)$$

where x_n is the value of variable x in the n-th iteration. This equation can be used to map a uniformly distributed random variable to a non-uniformly distributed one if sufficient iterations have been performed to ensure that the sequence enters the chaotic state. And a random number r between 0 and 1 can, thus, be obtained with the An map.

To avoid disrupting the main search process of the standard GA, in our scheme, the r/10 is used to determine the percentage of elements in the best chromosome to be disturbed. Since the position of a better chromosome is not known a priori, which elements would be disturbed are chosen randomly in our scheme. For the same reason, the value of an element in the chromosome is also chosen randomly in the core set C.

After applying the disturbance, the fitness value of the new chromosome is calculated. To prevent corrupting the best solution found by the population, the best chromosome is replaced by the new one only if the new chromosome could produce smaller value of Equation (5) than that of the best one.

5 Experimental Results

In this section, the effectiveness of the proposed modifications to the standard GA (SGA) [11] is evaluated in Experiment I. Then we evaluate the proposed algorithm by comparing the workload balancing capability, the maxspan after migration and the execution time of algorithm with a simple deterministic algorithm (DA), the PSO [10], the DPSO [19] and the SGA in Experiment II. Maxspan, as defined in [12], is the largest task completion time among all the processors in the system. Thus, in our experiment, the maxspan can be determined by finding the largest workload among all fault-free cores after migration, which can be calculated according to Equation (4). The test cases used throughout the two experiments are described in Section 5.1.

5.1 General experiment setup

To evaluate the performance of the algorithms, three selection probabilities are utilized to construct different workload scenarios based on the telecomm benchmarks from the Embedded System Synthesis Benchmarks Suite (E3S) [25], so that the tasks in the system can follow a desired distribution. These scenarios are considered as the initial condition of the system before cores fail. The processor core, in this experiment, is assumed to be the IBM PowerPC 750CX-500MHz. Hence, the required execution time of each benchmark task can be calculated accordingly.

The selection probabilities used are the uniform distribution, normal distribution and inverse normal distribution. Specifically, the uniform distribution evenly selects a desired number of tasks from the 16 benchmark tasks according to the required execution time. The normal distribution generates a set of tasks with more of the medium tasks and less of the heavy and light tasks. While the inverse normal

distribution selects less of the medium tasks and more of the other tasks.

For the first experiment, an 8×8 2D mesh NoC-based manycore system is targeted. With the above selection probabilities, 18 cases are generated where 20 and 40 tasks are selected respectively for each core. For each scenario, 10%, 40% and 80% of cores in the manycore system are chosen randomly to be faulty to simulate different faulty conditions of the system. For clarity, these faulty cases are categorized into six workload distribution cases, as shown in Table 1.

Table 1 Workload distribution cases

Number of tasks	Uniform distribution	Normal distribution	Inverse normal distribution
20	Workload I	Workload II	Workload III
40	Workload IV	Workload V	Workload VI

All the algorithms are implemented using MATLAB 7.11 and tested on a computer with Pentium Dual-Core CPU E5200 operating at 2.5GHz, 3GB memory and Windows XP operating system.

5.2 Experiment I

To evaluate the optimization performance of the proposed modifications to the SGA, in this experiment, 2000 iterations (G_{max} =2000) for both SGA and ACAD-GM are performed, and the two algorithms run till G_{max} iterations before exit. A population size of 20 for both algorithms is used. Other parameters for SGA are selected as follows: generation gap is 0.95, crossover rate is 0.9, and mutation rate is 0.01. Final results for each case are averaged over 50 independent trials and shown in Table 2.

Table 2 Comparison between SGA and ACAD-GM on 18 test cases

Workload cases	Faulty percentage	Fitness value			Standard deviation			Execution time		
		SGA	ACAD-GM	Improv.	SGA	ACAD-GM	Improv.	SGA	ACAD-GM	Degrad.
I	10%	5.57E+06	5.57E+06	0.04%	4.48E+04	3.90E+04	13.1%	3.79E+00	6.38E+00	68.4%
	40%	2.06E+05	1.10E+05	46.4%	4.93E+04	2.80E+04	43.2%	8.04E+00	1.26E+01	56.5%
	80%	1.26E+05	5.44E+04	56.7%	3.25E+04	1.95E+04	40.0%	1.21E+01	1.92E+01	59.1%
II	10%	4.82E+05	4.80E+05	0.6%	2.05E+04	2.24E+04	-8.9%	3.67E+00	6.27E+00	70.9%
	40%	1.15E+05	7.48E+04	35.1%	2.20E+04	1.59E+04	27.8%	7.96E+00	1.26E+01	58.1%
	80%	6.70E+04	3.32E+04	50.4%	1.74E+04	1.15E+04	33.7%	1.23E+01	1.92E+01	56.5%
III	10%	3.98E+06	3.97E+06	0.24%	5.24E+04	5.36E+04	-2.3%	3.82E+00	6.36E+00	66.2%
	40%	2.64E+05	1.18E+05	55.4%	5.56E+04	3.63E+04	34.7%	8.06E+00	1.28E+01	59.2%
	80%	1.48E+05	6.10E+04	58.8%	4.63E+04	2.21E+04	52.3%	1.23E+01	2.32E+01	89.6%
IV	10%	1.57E+06	1.56E+06	0.76%	4.80E+04	4.18E+04	12.8%	5.00E+00	8.35E+00	66.9%
	40%	6.83E+05	4.07E+05	40.4%	1.15E+05	7.27E+04	36.9%	1.23E+01	1.96E+01	59.8%
	80%	3.62E+05	1.54E+05	57.5%	8.07E+04	6.29E+04	22.1%	2.25E+01	3.49E+01	55.2%
V	10%	4.68E+05	4.45E+05	4.8%	2.85E+04	3.31E+04	-16.0%	4.79E+00	8.43E+00	76.2%
	40%	6.48E+05	3.84E+05	40.8%	1.03E+05	7.04E+04	31.5%	1.24E+01	1.98E+01	59.6%
	80%	2.91E+05	1.22E+05	58.2%	7.45E+04	3.79E+04	49.1%	2.22E+01	3.51E+01	57.7%
VI	10%	9.67E+06	9.63E+06	0.38%	4.82E+04	3.47E+04	28.1%	4.98E+00	8.40E+00	68.6%
	40%	8.23E+05	4.54E+05	44.8%	1.51E+05	8.26E+04	45.4%	1.24E+01	1.99E+01	60.3%
	80%	4.24E+05	1.76E+05	58.4%	1.27E+05	7.07E+04	44.6%	2.18E+01	3.44E+01	57.7%

Note: the minus sign "-" means degradation

It can be seen from the results illustrated in Table 2 that ACAD-GM achieves better fitness values than SGA in all 18 cases and has lower standard deviations than SGA in most cases. When the number of faulty cores increases in the system, the advantage of ACAD-GM in finding better solutions becomes clearer regardless of the workload distribution. The average improvements on the fitness value and the standard deviation are 33.9% and 27.1%, respectively. Therefore, the proposed schemes are efficient in aiding SGA to find better solutions. These improvements, however, are at the cost of averagely 63.7% more execution time of the SGA. This overhead seems to be very large. We will show, however, in the second experiment that the time overhead depends actually on the iterations the algorithm takes.

To further study the effect of the proposed AC and AD schemes on the optimization behavior, the evolution process in searching for the optimal solution is compared. Since the improvements of ACAD-GM on the 10% faults and the 80% faults are two extreme cases for each workload distribution, where evolution curves of the two

algorithms tend to be overlapped and separate for the 10% faults and 80% faults cases, respectively, the six distribution cases with medium faulty percentage (i.e., 40%) are illustrated as follows.

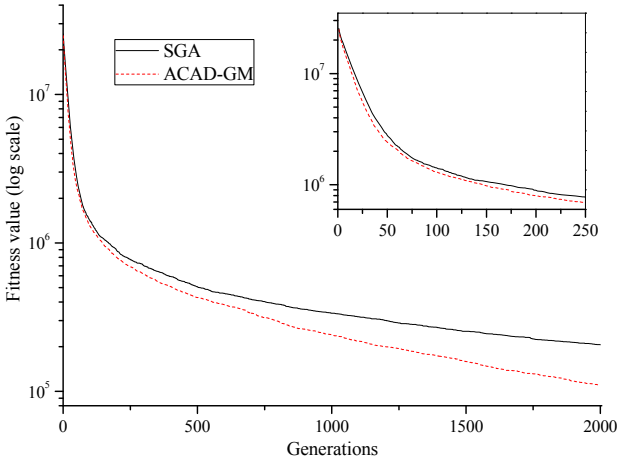

Fig.4 Evolution process comparison under Workload I

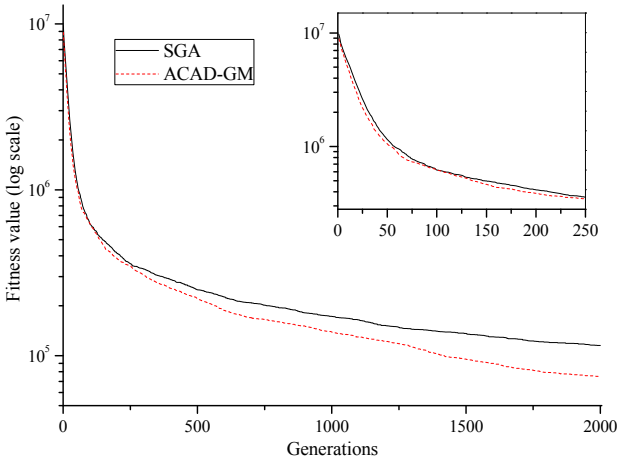

Fig.5 Evolution process comparison under Workload II

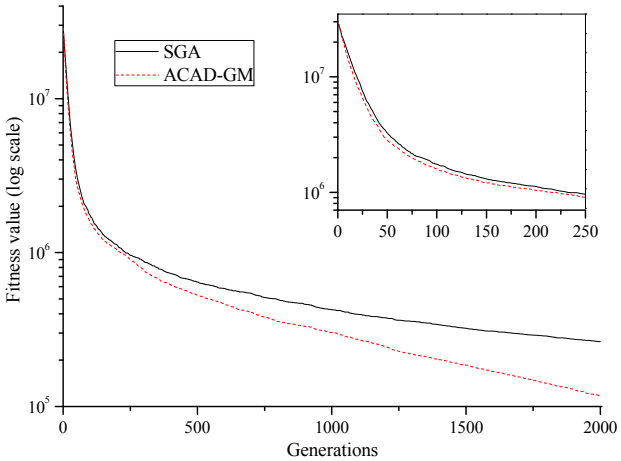

Fig.6 Evolution process comparison under Workload III

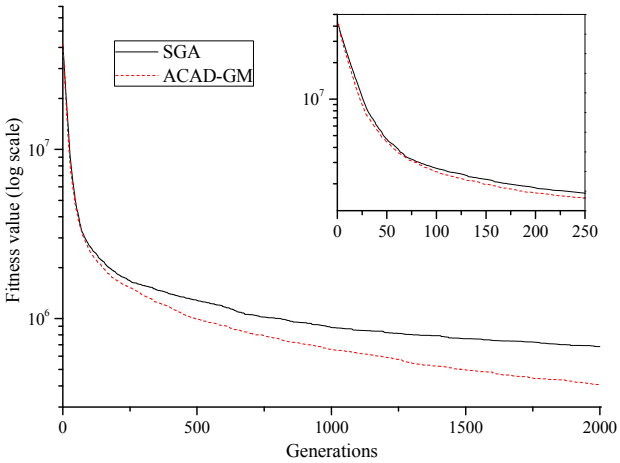

Fig.7 Evolution process comparison under Workload IV

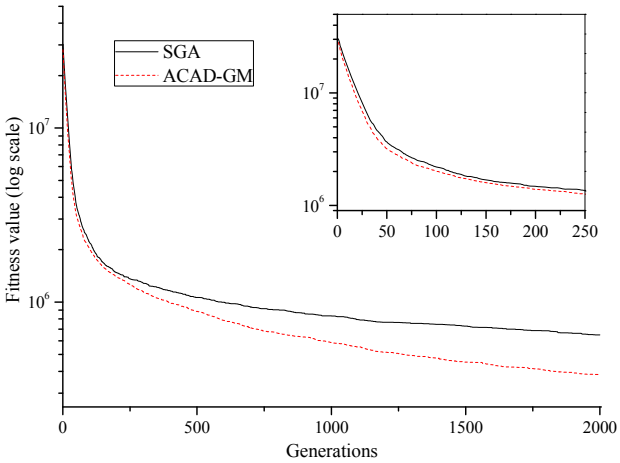

Fig.8 Evolution process comparison under Workload V

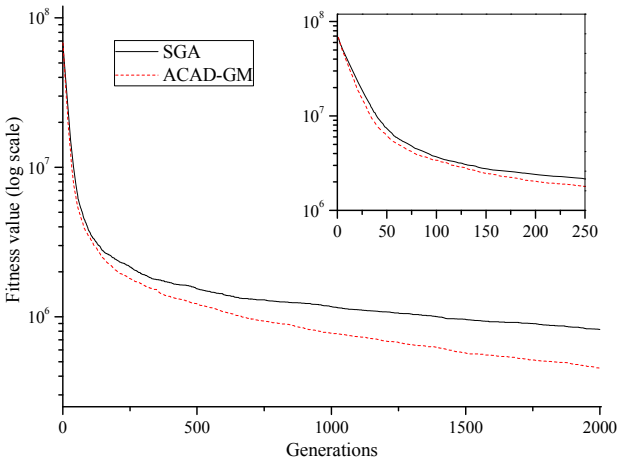

Fig.9 Evolution process comparison under Workload VI

As seen from Fig.4 to Fig.9, although the advantages of ACAD-GM over SGA are not clear for the first 250 generations under Workload II and Workload IV as demonstrated in Fig.5 and Fig.7, ACAD-GM converges generally faster than SGA

for all six workload distributions. This verifies the effectiveness of the proposed AC and AD schemes. As for the least improvement case on the fitness value listed in Table 2, ACAD-GM only reduces the fitness value by 0.04% compared with the SGA. The comparison of evolution curves for this case is illustrated in Fig.10 for the sake of completeness. It can be observed that the advantage of ACAD-GM is clear for the first 250 generations. And although it is hard for both algorithms to find better solutions after about 1200 generations due to limited task mapping choices, ACAD-GM still produces solutions with lower fitness values. Thus, this also justifies the effectiveness of the AC and AD schemes.

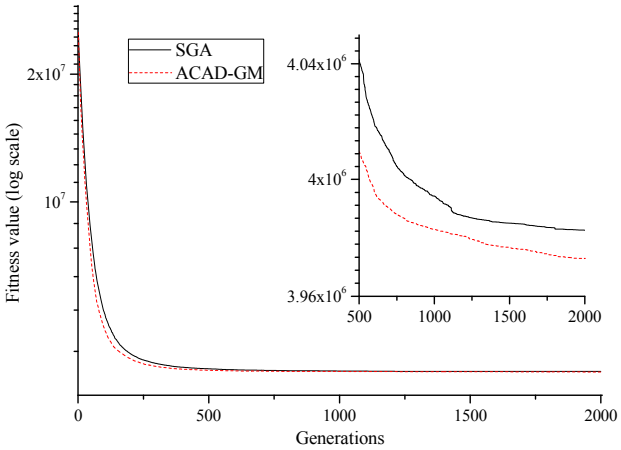

Fig.10 Evolution process comparison under Workload I with 10% faults

5.3 Experiment II

In this experiment, the ACAD-GM algorithm is compared with the DA, the PSO, the DPSO and the SGA algorithms in terms of workload balancing capability, execution time and maxspan. Before presenting the setup for this experiment and the details of the results, the main search process of DA is briefly described in Section 5.3.1 for clarity.

5.3.1 A baseline algorithm

The idea of presenting the DA here is to provide a baseline for the comparison, since the solutions DA generated does not vary with different trials.

As illustrated in Fig.11, DA generates solutions by repeatedly sorting fault-free cores by the workload on them and adding the heaviest remaining task to the lightest fault-free core after the sort. Although this would not guarantee a solution with the lowest objective function value (i.e., the value of Equation (5)) to be found, it provides a better baseline algorithm than a random assignment policy.

Algorithm: DA
Input: task workload set $w(T)$;
core workload set $w(C)$
Output: migration solution S
1: sort tasks in T by $w(T)$ in descending order to get T_{sorted}
2: $i = 0$
3: **for** $i <$ total number of elements in T_{sorted} **do**
4: sort fault-free cores in C by $w(C)$ in ascending order to get C_{sorted}
5: $S(i) = C_{\text{sorted}}(0)$ (i.e., the core with the lightest workload in C_{sorted})
6: $w(C_{\text{sorted}}(0)) = w(C_{\text{sorted}}(0)) + w(T_{\text{sorted}}(i))$
7: $i = i+1$
8: **return** S

Fig.11 Pseudo code of DA

5.3.2 Setup for experiment II

In this experiment, totally 30 types of workload scenarios are generated, where 20 and 40 tasks are selected respectively for each core with different network sizes varying from 4×4 to 12×12. For each scenario, 10%, 20%, 40%, 60% and 80% of cores in the manycore system are chosen to be faulty randomly. Consequently, there are totally 150 faulty cases for each algorithm. To present the results clearly, these test cases are also categorized into six groups according to the workload distribution, as shown in Table 1.

Table 3 Parameters for algorithms

Algorithm	Parameters
PSO	c_1=c_2=1.7, W=0.6, $V_{\max}$=$X_{\max}$/2, *PopSize*=20, $G_{\max}$=200
DPSO	c_1=c_2=2.0, $V_{\max}$=40, *PopSize*=20, $G_{\max}$=200
SGA	generation gap g_{gap}=0.95, crossover rate p_c=0.9, mutation rate p_m=0.01, *PopSize*=20, $G_{\max}$=200
ACAD-GM	generation gap g_{gap}=0.95, *PopSize*=20, $G_{\max}$=200

The parameter settings of the algorithms are listed in Table 3, where DA is not included as it does not require any parameter. The parameters of PSO are chosen as recommended in [26], as it is reported to have higher convergence rate than the typical parameter set recommended by [27]. The maximum range that a particle can fly (i.e., $X_{\max}$), however, is limited to the total number of elements in core set C, and the maximum velocity $V_{\max}$ is limited to $X_{\max}/2$, which is different from the usage

presented in [26]. The reason for putting these limitations to X_{max} and V_{max} is that assigning tasks to cores outside of core set C is meaningless in our experiment. For each stochastic algorithm, the population size (*PopSize*) and the maximum iteration number (G_{max}) are set to 20 and 200, respectively. And all four stochastic algorithms run till G_{max} iterations before exit. Their final results are averaged over 100 trials.

5.3.3 Comparison of workload balancing capability and execution time

In this section, workload balancing capabilities of the algorithms are compared by considering Equation (5) as the objective function. Therefore, the algorithm that generates the lowest function value is preferred. The value of the objective function is referred to as the fitness value in this experiment, and the results are normalized to DA. Since it is not proper to display all the 150 results for each algorithm due to the length of the paper, only 10%, 40% and 80% of faults (i.e., totally 90 cases) for each algorithm are chosen to be illustrated from Fig.12 to Fig.17.

The corresponding 90 execution time results for each algorithm are listed in Table 4. The complete 750 results of the fitness value and 750 results of the execution time are used for calculation only. Table 5 is a scoreboard summarized from the complete results of the execution time. It shows the number of times that an algorithm obtains a certain rank when sorted by execution time, among all the 150 results of the algorithm.

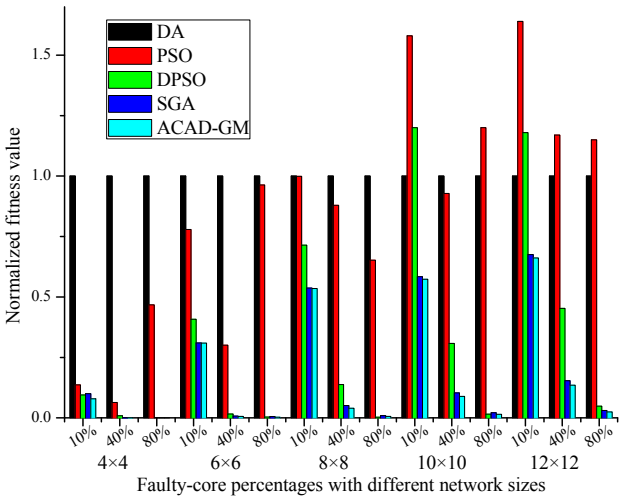

Fig.12 Workload balancing capability comparison under Workload I

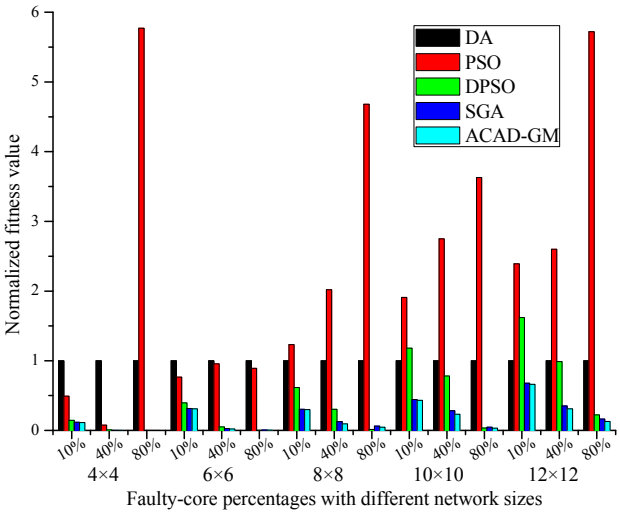

Fig.13 Workload balancing capability comparison under Workload II

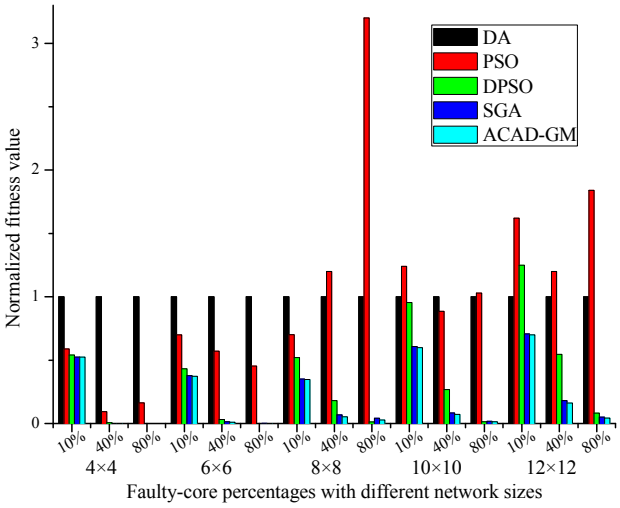

Fig.14 Workload balancing capability comparison under Workload III

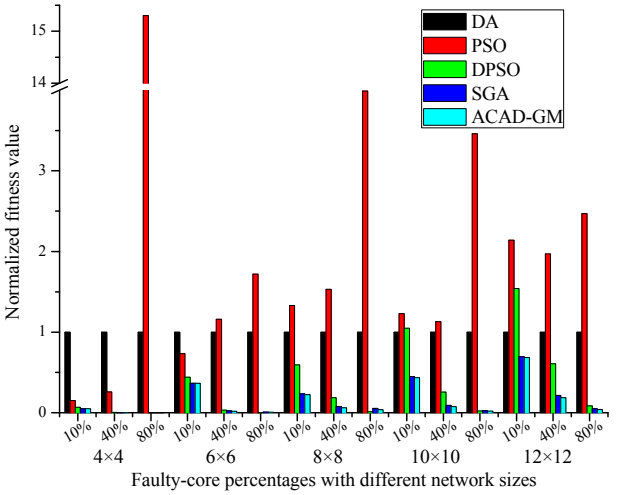

Fig.15 Workload balancing capability comparison under Workload IV

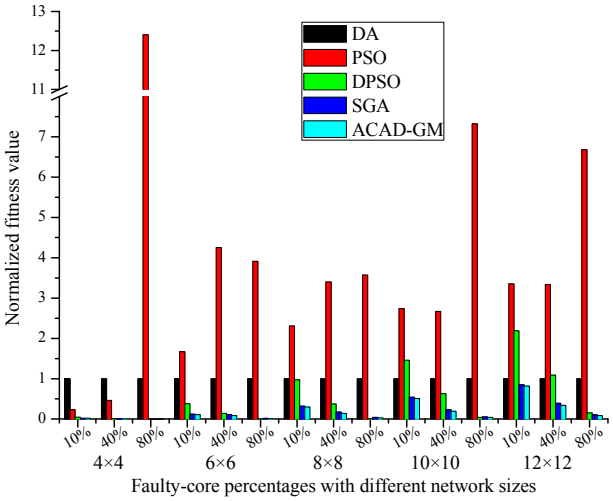

Fig.16 Workload balancing capability comparison under Workload V

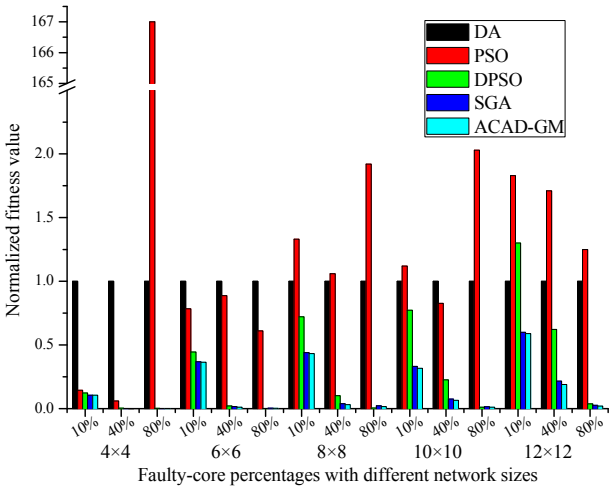

Fig.17 Workload balancing capability comparison under Workload VI

Table 4 Execution time of algorithms corresponding to the 90 workload cases shown from Fig.12 to Fig.17

Workload cases	Network size	Faulty percentage	Execution time for each algorithm (s)				
			DA	PSO	DPSO	SGA	ACAD-GM
I	4×4	10%	2.19E-03	2.15E-01	9.43E-01	2.80E-01	3.02E-01
		40%	6.29E-03	2.79E-01	2.47E+00	3.64E-01	4.01E-01
		80%	1.36E-02	4.13E-01	4.86E+00	5.02E-01	5.52E-01
	6×6	10%	4.45E-03	2.46E-01	1.89E+00	3.25E-01	3.48E-01
		40%	1.52E-02	4.12E-01	5.93E+00	5.39E-01	5.81E-01
		80%	3.09E-02	5.99E-01	1.11E+01	8.20E-01	8.51E-01
	8×8	10%	7.07E-03	2.81E-01	2.98E+00	3.66E-01	3.91E-01
		40%	3.04E-02	7.18E-01	1.20E+01	7.98E-01	8.31E-01
		80%	5.37E-02	6.02E-01	2.02E+01	1.22E+00	1.28E+00
	10×10	10%	1.23E-02	3.84E-01	5.19E+00	4.53E-01	4.88E-01
		40%	4.65E-02	1.04E+00	1.90E+01	1.02E+00	1.07E+00
		80%	8.55E-02	1.48E+00	3.31E+01	1.79E+00	1.93E+00
	12×12	10%	1.78E-02	4.29E-01	8.15E+00	5.46E-01	5.77E-01
		40%	7.08E-02	1.19E+00	2.89E+01	1.37E+00	1.46E+00
		80%	1.24E-01	1.98E+00	4.93E+01	2.65E+00	2.64E+00
II	4×4	10%	2.20E-03	2.15E-01	9.38E-01	2.78E-01	3.00E-01
		40%	6.35E-03	2.79E-01	2.48E+00	3.65E-01	3.91E-01
		80%	2.77E-02	4.32E-01	4.86E+00	4.99E-01	5.47E-01
	6×6	10%	4.45E-03	2.46E-01	1.89E+00	3.25E-01	3.48E-01
		40%	1.51E-02	4.17E-01	5.94E+00	5.39E-01	5.84E-01
		80%	3.02E-02	6.26E-01	1.11E+01	8.25E-01	8.59E-01
	8×8	10%	7.09E-03	2.80E-01	2.96E+00	3.66E-01	4.02E-01
		40%	2.88E-02	6.80E-01	1.19E+01	7.76E-01	8.30E-01
		80%	5.44E-02	6.38E-01	2.02E+01	1.22E+00	1.29E+00
	10×10	10%	1.25E-02	4.14E-01	5.17E+00	4.56E-01	4.91E-01
		40%	4.66E-02	1.01E+00	1.91E+01	1.02E+00	1.07E+00
		80%	8.62E-02	1.30E+00	3.28E+01	1.84E+00	1.92E+00
	12×12	10%	1.87E-02	4.47E-01	8.07E+00	5.46E-01	5.87E-01
		40%	7.02E-02	1.19E+00	2.87E+01	1.39E+00	1.45E+00
		80%	1.24E-01	2.18E+00	4.89E+01	2.50E+00	2.86E+00
III	4×4	10%	2.28E-03	2.20E-01	9.43E-01	2.83E-01	3.11E-01
		40%	6.59E-03	2.85E-01	2.47E+00	3.62E-01	3.90E-01
		80%	1.34E-02	4.28E-01	4.82E+00	5.22E-01	5.37E-01
	6×6	10%	4.44E-03	2.50E-01	1.90E+00	3.25E-01	3.49E-01
		40%	1.51E-02	4.53E-01	5.93E+00	5.16E-01	5.79E-01
		80%	3.04E-02	6.01E-01	1.12E+01	8.09E-01	8.69E-01

Table 4 (Continued)

Workload cases	Network size	Faulty percentage	Execution time for each algorithm (s)				
			DA	PSO	DPSO	SGA	ACAD-GM
III	8×8	10%	7.25E-03	2.85E-01	2.97E+00	3.65E-01	4.06E-01
		40%	2.89E-02	6.92E-01	1.20E+01	8.00E-01	8.25E-01
		80%	5.35E-02	5.90E-01	2.04E+01	1.22E+00	1.28E+00
	10×10	10%	1.22E-02	3.50E-01	5.19E+00	4.56E-01	4.85E-01
		40%	4.67E-02	9.21E-01	1.92E+01	1.02E+00	1.07E+00
		80%	8.48E-02	1.44E+00	3.31E+01	1.83E+00	1.89E+00
	12×12	10%	1.79E-02	4.47E-01	8.10E+00	5.46E-01	5.74E-01
		40%	7.14E-02	1.19E+00	2.88E+01	1.39E+00	1.47E+00
		80%	1.25E-01	2.14E+00	4.94E+01	2.50E+00	2.86E+00
IV	4×4	10%	4.40E-03	2.47E-01	1.73E+00	3.24E-01	3.50E-01
		40%	1.26E-02	3.72E-01	4.75E+00	4.78E-01	5.15E-01
		80%	2.68E-02	4.16E-01	9.50E+00	8.05E-01	8.27E-01
	6×6	10%	8.88E-03	3.13E-01	3.66E+00	3.95E-01	4.29E-01
		40%	2.98E-02	5.83E-01	1.18E+01	8.10E-01	8.44E-01
		80%	6.05E-02	8.97E-01	2.21E+01	1.38E+00	1.45E+00
	8×8	10%	1.42E-02	3.77E-01	5.92E+00	4.96E-01	5.36E-01
		40%	5.78E-02	1.07E+00	2.37E+01	1.23E+00	1.34E+00
		80%	1.08E-01	1.16E+00	4.05E+01	2.20E+00	2.35E+00
	10×10	10%	2.46E-02	5.04E-01	1.02E+01	6.58E-01	7.16E-01
		40%	9.47E-02	1.64E+00	3.81E+01	1.78E+00	1.91E+00
		80%	1.71E-01	2.16E+00	6.62E+01	4.06E+00	4.09E+00
	12×12	10%	3.61E-02	6.36E-01	1.60E+01	8.45E-01	8.47E-01
		40%	1.40E-01	2.47E+00	5.81E+01	2.63E+00	2.85E+00
		80%	2.47E-01	6.33E+00	1.00E+02	6.42E+00	6.47E+00
V	4×4	10%	4.29E-03	2.47E-01	1.73E+00	3.24E-01	3.49E-01
		40%	1.29E-02	3.60E-01	4.76E+00	4.79E-01	5.31E-01
		80%	2.70E-02	3.75E-01	9.54E+00	8.03E-01	8.23E-01
	6×6	10%	8.83E-03	3.13E-01	3.65E+00	3.94E-01	4.31E-01
		40%	3.01E-02	5.50E-01	1.17E+01	7.84E-01	8.21E-01
		80%	6.02E-02	9.38E-01	2.20E+01	1.37E+00	1.43E+00
	8×8	10%	1.42E-02	3.74E-01	5.82E+00	4.92E-01	5.23E-01
		40%	5.78E-02	9.96E-01	2.35E+01	1.27E+00	1.32E+00
		80%	1.07E-01	1.17E+00	4.06E+01	2.24E+00	2.35E+00
	10×10	10%	2.47E-02	5.03E-01	1.02E+01	6.75E-01	6.98E-01
		40%	9.26E-02	1.59E+00	3.81E+01	1.78E+00	1.90E+00
		80%	1.70E-01	2.14E+00	6.61E+01	3.58E+00	4.08E+00
	12×12	10%	3.69E-02	6.37E-01	1.61E+01	8.16E-01	8.34E-01
		40%	1.41E-01	2.50E+00	5.80E+01	2.55E+00	2.69E+00
		80%	2.47E-01	6.54E+00	9.98E+01	6.33E+00	6.45E+00
VI	4×4	10%	4.27E-03	2.47E-01	1.73E+00	3.24E-01	3.51E-01
		40%	1.27E-02	4.39E-01	4.79E+00	4.98E-01	5.22E-01
		80%	2.69E-02	5.85E-01	9.56E+00	8.08E-01	8.45E-01
	6×6	10%	8.93E-03	3.35E-01	3.68E+00	4.10E-01	4.31E-01
		40%	2.98E-02	5.69E-01	1.18E+01	7.96E-01	8.33E-01
		80%	6.14E-02	9.60E-01	2.22E+01	1.37E+00	1.45E+00
	8×8	10%	1.42E-02	4.08E-01	5.91E+00	4.95E-01	5.37E-01
		40%	5.77E-02	1.04E+00	2.38E+01	1.26E+00	1.33E+00
		80%	1.08E-01	1.39E+00	4.05E+01	2.29E+00	2.35E+00
	10×10	10%	2.44E-02	5.96E-01	1.02E+01	6.66E-01	7.05E-01
		40%	9.30E-02	1.85E+00	3.82E+01	1.79E+00	1.89E+00
		80%	1.70E-01	2.25E+00	6.62E+01	3.56E+00	3.75E+00
	12×12	10%	3.58E-02	6.37E-01	1.60E+01	7.83E-01	8.22E-01
		40%	1.43E-01	2.49E+00	5.82E+01	2.58E+00	2.70E+00
		80%	2.49E-01	6.27E+00	9.96E+01	6.17E+00	6.38E+00

Table 5 Algorithm ranking based on execution time

Algorithm	Rank (rank 1 is the fastest)				
	1	2	3	4	5
DA	150	0	0	0	0
PSO	0	142	6	2	0
DPSO	0	0	0	0	150
SGA	0	8	140	2	0
ACAD-GM	0	0	4	146	0

Generally, it can be seen from Fig.12 to Fig.17 that, for all test cases, PSO and DPSO could not guarantee better workload balancing capabilities than DA, and PSO cannot generate better solutions than DPSO. When the number of tasks executed on a single core increases from 20 to 40, PSO has higher probability to be trapped in the local optima, resulting in more unbalanced workload distribution, while the results of DPSO does not vary largely. For example, in a 16-core system with 13 faulty cores (i.e., the case of 80% fault and 4×4 network size), PSO handles well for Workload I and Workload III but not for other workload distributions. Especially for Workload VI, it generates solutions with the fitness value almost 166 times larger than DA. While for DPSO, the fitness value it achieved in the worst case is only about two times the value for DA. DPSO produces smaller fitness values as the number of faulty cores increases for a given network size. On the other hand, as shown in Table 4 and Table 5, PSO requires the second shortest execution time in most cases (142 cases to be exact), while DPSO has the longest execution time in all cases.

Therefore, it can be concluded from the above results that although PSO operates fast and could generate good solutions in some cases, it is generally not as effective as DPSO in solving discrete problems. Meanwhile, although DPSO outperforms DA in generating solutions with lower fitness values in most cases, it requires the longest execution time.

As for SGA and ACAD-GM, it can be observed that both the algorithms produce solutions better than DA in all cases and their execution time are in the same order of magnitude as PSO, which are generally one or two orders of magnitude lower than DPSO. The figures also shows that for a given network size, SGA and ACAD-GM produce more balanced workload distribution as the number of faulty cores increases, which is similar to the behavior of DPSO. The main reason for this phenomenon is that when more cores fail, more combinations can be tried to find an optimal solution. Meanwhile, more execution time is required (as shown in Table 4).

A more detailed comparison between ACAD-GM and other algorithms has been performed based on the exact 750 fitness values (not included here due to the length of the paper). And we found that, in all 150 faulty cases, ACAD-GM achieves on average 86%, 88%, 37% and 16% improvements on the fitness value (i.e., lower fitness value) compared with DA, PSO, DPSO and SGA, respectively. A detailed comparison based on the complete 750 results of the execution time shows that ACAD-GM requires on average 35%, 36.3% and 6.2% more execution time than DA, PSO and SGA, respectively, while reduces 90.9% execution time compared with DPSO.

5.3.4 Comparison of maxspan

The results illustrated from Fig.18 to Fig.23 are the maxspans generated by each algorithm under six workload distribution cases.

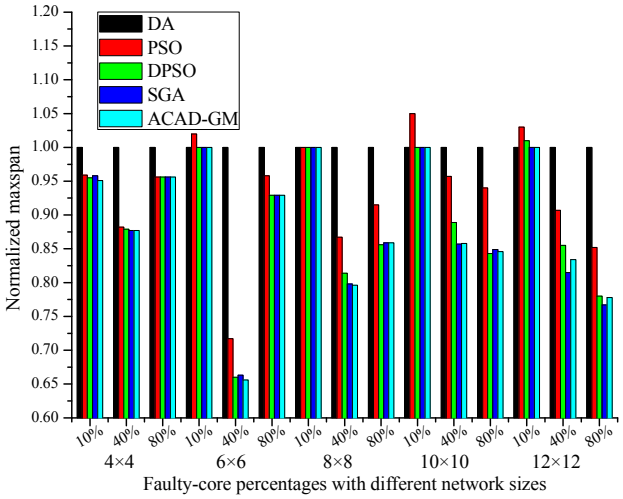

Fig.18 Maxspan comparison under Workload I

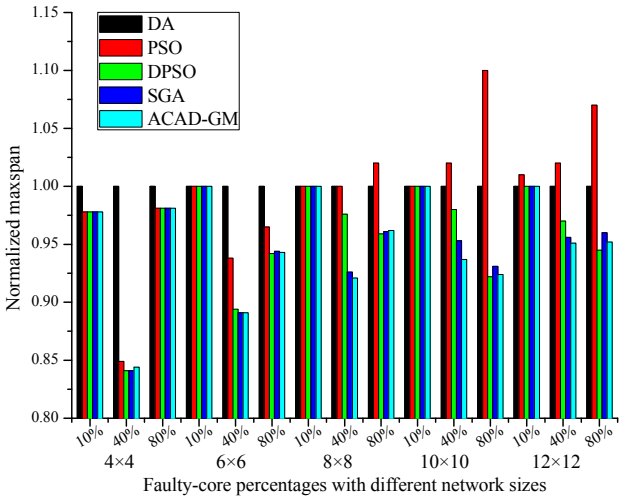

Fig.19 Maxspan comparison under Workload II

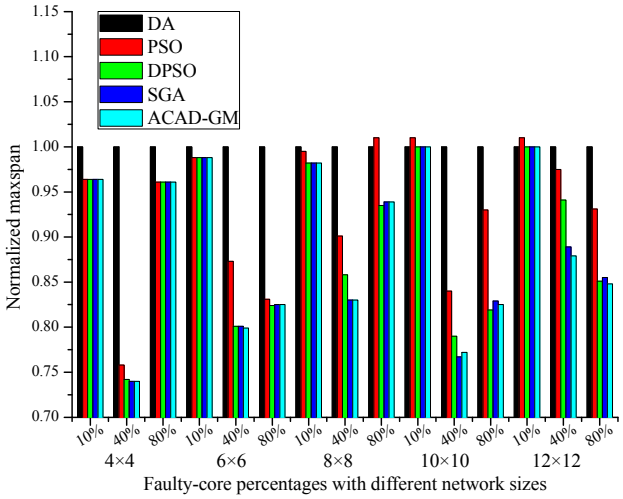

Fig.20 Maxspan comparison under Workload III

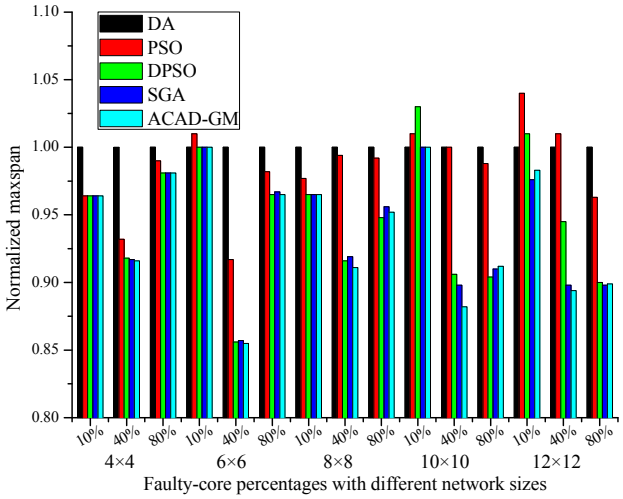

Fig.21 Maxspan comparison under Workload IV

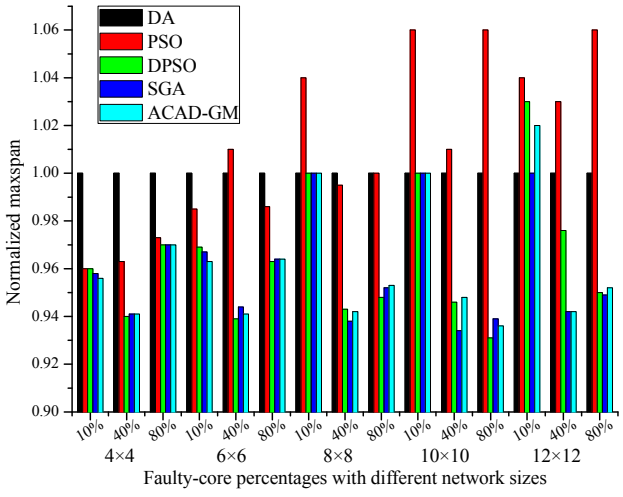

Fig.22 Maxspan comparison under Workload V

It can be discovered generally from Fig.18 to Fig.23 that DPSO, SGA and ACAD-GM produce smaller maxspans than DA and PSO in most cases. And the worst maxspan of PSO is only 10% larger than that of DA, as shown in Fig.19. Comparing

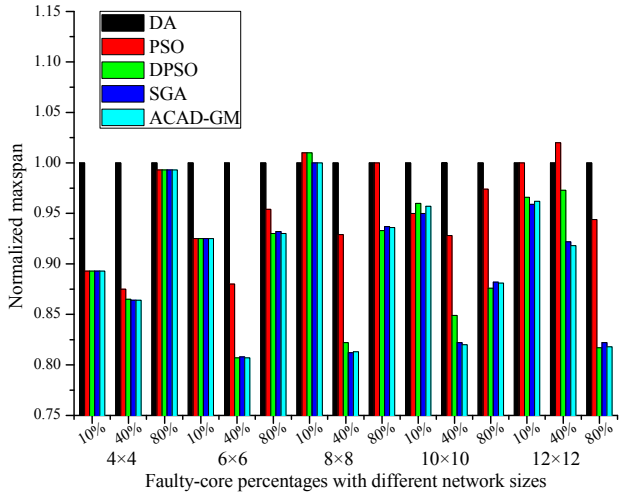

Fig.23 Maxspan comparison under Workload VI

with the results from Fig.12 to Fig.17, it can be observed that a more balanced workload distribution does not necessarily guarantee a smaller maxspan. This is because the maxspan only depends on the maximum workload in the system. And if the workload on one core is much larger than any of the workloads on other cores before migration, the algorithms would map more tasks onto the light-loaded cores first due to the restriction of Equation (5). Thus if the workload on any of the other cores does not exceed the workload on the originally heaviest-loaded core, then the maxspan after migration remains the same. A detailed analysis based on the complete 750 results of the maxspan shows that ACAD-GM achieves on average 8.67%, 5.13%, 0.98% and 0.14% improvements on the maxspan compared with DA, PSO, DPSO and SGA, respectively.

Table 6 Improvements of ACAD-GM summarized from Experiment II

Aspect	Algorithms being compared			
	DA	PSO	DPSO	SGA
Fitness value	86%	88%	37%	16%
Execution time	-35%	-36.3%	90.9%	-6.2%
Maxspan	8.67%	5.13%	0.98%	0.14%

Note: the minus sign "-" means degradation

For clarity, the improvements of the proposed ACAD-GM algorithm on fitness value, execution time and maxspan compared with the other four algorithms are summarized in Table 6. Together with the above analysis, it can be seen from Table 6 that although ACAD-GM has little advantage in shortening the maxspan, it can provide the best-balanced workload distribution after task migration, which is important to the lifetime of the whole

manycore system. Moreover, the added schemes in ACAD-GM improve the workload balancing capability of SGA by 16% while only adds 6.2% of the execution time. Since the extra computations introduced by the AC and AD schemes in each generation are constant for a given faulty case, this time overhead varies almost linearly as the maximum generation changes from 2000 to 200, resulting in approximately 9 times smaller than the ratio 63.7% shown in Experiment I.

6 Conclusion

In this paper, a genetic task migration algorithm, namely ACAD-GM, is proposed towards workload balancing for fault recovery in NoC-based manycore systems. It incorporates an adaptive crossover scheme and a chaotic disturbance scheme with the standard genetic algorithm to improve the searching efficiency in solving the task migration problem. Experiments verify the effectiveness of the proposed schemes and shows that the ACAD-GM has better workload balancing capability compared with four relevant algorithms. The time overhead caused by the AC and the AD schemes varies almost linearly with the number of search iterations.

Acknowledgements

This research was supported by the Harbin Applied Technology Research and Development Project (Young Talents for Scientific and Technological Innovation) (Grant No. 2013RFQXJ095) and the Fundamental Research Funds for the Central Universities (Grant No. HIT.NSRIF.2014039). The authors would also like to thank the anonymous reviewers for their valuable suggestions.

References:

[1] http://www.tilera.com.

[2] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge et al., A 167-processor computational platform in 65 nm CMOS, *IEEE Journal of Solid-State Circuits*, Vol.44, No.4, 2009, pp. 1130-1144.

[3] Z. Lu and A. Jantsch, Trends of terascale computing Chips in the next ten years, *in Proc. of IEEE 8th International Conference on ASIC*, 2009, pp. 62-66.

[4] http://www.jedec.org.

[5] R. Marculescu, U. Y. Ogras, L. S. Peh, N. E. Jerger, and Y. Hoskote, Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.28, No.1, 2009, pp. 3-21.

[6] S. Borkar, N. P. Jouppi, and P. Stenstrom, Microprocessors in the era of terascale integration, *in Proc. of the conference on Design, automation and test in Europe, EDA Consortium*, 2007, pp. 237-242.

[7] S. Mitra, K. Brelsford, Y.M. Kim, H.H.K. Lee, and Y. Li, Robust system design to overcome CMOS reliability challenges, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol.1, No.1, 2011, pp. 30-41.

[8] L. Huang, F. Yuan, and Q. Xu, Lifetime reliability-aware task allocation and scheduling for MPSoC platforms, *in Proc. of the Conference on Design, Automation and Test in Europe*, 2009, pp. 51-56.

[9] C. Lee, H. Kim, H. W. Park, S. Kim, H. Oh, and S. Ha, A task remapping technique for reliable multi-core embedded systems, *in Proc. of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2010, pp. 307-316.

[10] A. Salman, I. Ahmad, and S. Al-Madani, Particle swarm optimization for task assignment problem, *Microprocessors and Microsystems*, Vol.26, No.8, 2002, pp. 363-371.

[11] F. A. Omara and M. M. Arafa, Genetic algorithms for task scheduling problem, *Journal of Parallel and Distributed Computing*, Vol.70, No.1, 2010, pp. 13-22.

[12] A. Y. Zomaya and Y. H. Teh, Observations on using genetic algorithms for dynamic load-balancing, *IEEE Transactions on Parallel and Distributed Systems*, Vol.12, No.9 , 2001, pp. 899-911.

[13] P. Mesidis and L. S. Indrusiak, Genetic mapping of hard real-time applications onto NoC-based MPSoCs - A first approach, *in Proc. of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2011, pp.1,6, pp. 20-22.

[14] M. Srinivas and L. M. Patnaik, Adaptive probabilities of crossover and mutation in genetic algorithms, *IEEE Transactions on Systems, Man and Cybernetics*, Vol.24, No.4, 1994, pp. 656-667.

[15] J. A. Vasconcelos, J. A. Ramirez, R. H. C. Takahashi and R. R. Saldanha, Improvements in genetic algorithms, *IEEE Transactions on Magnetics*, Vol.37, No.5, 2001, pp. 3414-3417.

[16] D. W. Boeringer, D. H. Werner, and D. W. Machuga, A simultaneous parameter adaptation

scheme for genetic algorithms with application to phased array synthesis, *IEEE Transactions on Antennas and Propagation*, Vol.53, No.1, 2005, pp. 356-371.

[17] R. C. Eberhart and J. Kennedy, A new optimizer using particle swarm theory, *in Proc. of the sixth international symposium on micro machine and human science*, 1995, pp. 39-43.

[18] J. Kennedy and R. C. Eberhart, A discrete binary version of the particle swarm algorithm, *in Proc. of 1997 IEEE International Conference on Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation*., 1997, pp. 4104-4108.

[19] H. Izakian, B. T. Ladani, A. Abraham, and V. Snasel, A discrete particle swarm optimization approach for grid job scheduling, *International Journal of Innovative Computing, Information and Control*, Vol.6, No.9, 2010, pp. 4219-4233.

[20] A. J. Chipperfield, P. Fleming, and H. Pohlheim, *Genetic Algorithm Toolbox: For Use with MATLAB User's Guide (version 1.2)*, University of Sheffield, Department of Automatic Control and Systems Engineering, 1994.

[21] W. Abdulal and S. Ramachandram, Reliability-aware genetic scheduling algorithm in grid environment, *2011 International Conference on Communication Systems and Network Technologies*, 2011, pp. 673-677.

[22] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, A genetic algorithm framework for test generation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.16, No.9, 1997, pp. 1034-1044.

[23] M. W. Li, W. C. Hong, and H. G. Kang, Urban traffic flow forecasting using Gauss-SVR with cat mapping, cloud model and PSO hybrid algorithm, *Neurocomputing*, Vol.99, 2013, pp. 230-240.

[24] D. Yang, Z. Liu, and J. Zhou. Chaos optimization algorithms based on chaotic maps with different probability distribution and search speed for global optimization, *Communications in Nonlinear Science and Numerical Simulation*, Vol.19, No.4, 2014, pp. 1229-1246.

[25] R. Dick, Embedded system synthesis benchmarks suites, 2008.

[26] I. C. Trelea, The particle swarm optimization algorithm: convergence analysis and parameter selection, Information processing letters, Vol.85, No.6, 2003, pp. 317-325.

[27] M. Clerc, The swarm and the queen: towards a deterministic and adaptive particle swarm optimization, *in Proc. of the 1999 Congress on Evolutionary Computation*, 1999, pp. 1951-1957.