International Journal of Smart Computing and Artificial Intelligence
International Institute of Applied Informatics
2019, Vol. 3, No. 2, –

The effect of speculative computation on combinatorial optimization problems

Yasuki Iizuka *, Akira Hamada *, Yosuke Suzuki *

Abstract

In recent years, multicore or many-core processors have gained significant attention as they enable computation with a large degree of parallelism on desktop computers. However, conventional parallel processing methods often cannot easily achieve parallel effects due to various factors. In this study, we evaluated the effect of applying MultiStart-based speculative parallel computation to combinatorial optimization problems. Using probability theory, we performed theoretical verification to determine whether speculative computation is more effective than conventional parallel computation methods. In addition, we conducted experiments and compared the result with those of conventional parallel processing. In this paper, we report the results of the theoretical verification and experiments, and we show that speculative computation is more effective than conventional parallel processing.

Keywords: speculative computing, parallel processing, molecular simulation

1 Introduction

In recent years, CPU performance has been greatly improved. With the miniaturization of the CPU manufacturing process, the CPU is changing from the Multi-core to the Many-core era. At the same time, the demands for software complexity are also increasing. Parallel processing can effectively accelerate the computation of problems with large complexity. However, even if m parallel processes (or threads) are executed, the execution time will not be reduced to $1/m$ because (i) only some parts of the program can be parallelized, (ii) the process generates overhead, and (iii) some processes must be synchronized. Therefore, the capacity factor of parallel computing resources will not be 100%.

Parallelization can also be performed using speculative computation. Speculative computation is to pre-execute computations that may not use computation results in the future. For example, Multilisp[1] or MultiStart [2] executes speculative computation simultaneously (or in pseudo parallel). The purpose of this research is to investigate the effect of speculative computation on combinatorial optimization problems and to develop algorithms that use parallel computing resources efficiently. In order to use parallel computing for combinatorial optimization problems, the algorithm itself must be

* Graduate School of Science, Tokai University Hiratsuka, Japan

considered, and it is not easy. In this paper, we analyze the effect of simply executing one algorithm independently using multiple parallel threads with a different random number seed or different initial value, and select the best solution. Such speculative computation can realize parallel computation without changing existing algorithms. The speculative method uses a stochastic algorithm, such as the SA algorithm or Tabu search, as the base program.

We show the results of advanced theoretical estimations using probability theory. In addition, we performed an experiment to computationally solve an actual problem using the speculative method. We have been studying the effect of speculative computation by theoretical investigation and pseudo-parallel experiments [3, 4]. In this paper, we report the experimental results of its effect with a many-core processor. The results demonstrate that speculative computation is more effective than conventional parallel computing.

2 Parallel Processing

Many parallel programs are developed from sequential programs to reduce computation time, i.e., some parts of sequential programs can be parallelized. An image of the iterative improvement algorithm for the combinatorial optimization problem is shown in Fig.1. Continue to improve the solution while repeating the loop of line 3 to line 9. Suppose that the loop of lines 5 to 7 can be parallelized in this program. Note that parallelizable parts do not necessarily exist in the program of combinatorial optimization problem. Fig.2 shows an example of a parallelized program. Such programs incur process generation and synchronization overheads. When synchronization is required, completed processes must wait for running processes (line 17). Processing cannot continue during this waiting state; thus, computational resources are not used efficiently. Increasing the number of processes does not always increase processing speed. For example, if a part of a program is executed using 10 processes, the execution time does not necessarily increase by a factor of 10.

```
1: Algorithm II
2: Initialize();
3: while(condition) {
4:     ...
5:     for(){
6:         ...
7:     }
8:     ...
9: }
10: return minimumValue;
```

Figure 1. An image of iterative improvement algorithm

```
11: Algorithm CPP
12: Initialize();
13: while(condition) {
14:     ...
15:     parallel-for(){
16:         ...
17:     }
18:     ...
19: }
20: return minimumValue;
```

Figure 2. An example of conventional parallel processing

The ratio of the part of a program that can be parallelized relative to the entire program (line 5 - 7 in Fig.1) is assumed to be α ($0 \leq \alpha \leq 1$). When this part is parallelized by m processes (or threads), the computation time t_p is expressed using α as follows:

$$t_p = (\alpha/m + (1 - \alpha))t_s + \beta, \tag{1}$$

where β is the process generation time and synchronization wait time, t_s is the computation time by sequential processing. This formula is based on Amdahl's argument (Amdahl's low) [5] about the limit of parallelism. When α is sufficiently close to 1, the computation

time decreases to $t_s/m + \beta$. In many cases, α does not attain such an ideal value. Therefore, even if m is increased, the effect of parallelization is limited. In addition, there are also two upper limits of m, CPU resources and data parallelism.

Studies reporting the process to parallelize SA [6, 7] and Tabu search [8] were published more than 20 years ago. However, in these studies, synchronization was required in all iterations. This means that these have a large β.

In speculative parallel computation, a MultiStart [2] procedure executes several computations speculatively and simultaneously, including computations that may not be used. In theoretical analyses using the state transition matrix of the Markov process, it has been suggested that MultiStart can realize superlinearity [9, 10]. However, in these analyses, superlinearity was evaluated using theoretical comparisons based on a simple random search. To the best of our knowledge, experiments targeting practical problems other than benchmarks have not been performed. Note that it is difficult to realize superlinearity by comparing efficient algorithms using various heuristics.

Therefore, various speculative computation methods have been proposed [11, 12, 13]. The line-up competition algorithm (LCA) [14] is a speculative parallel computation algorithm that attempts to obtain the best result after executing many programs. LCA compares solutions among process groups, which are referred to as a family, at every iteration that yields an improvement and modifies the program parameters that are dynamically based on a comparison of the results. In LCA, this comparison is performed after each iteration; consequently, frequent synchronization is required. The synchronization cost of the speculative computation method used in our experiment is nearly zero; moreover, it can parallelize the existing algorithm without modification. In addition, wait time is not required; thus, nearly 100% of the computational resources can be used.

The algorithm portfolio method has also been proposed [15] as a type of promising speculative computation method for difficult problems. In the algorithm portfolio, some (one or more) algorithms are executed simultaneously; then, the algorithm with the best performance is selected. In our study, only one algorithm was executed in parallel; this may be a precise analysis of a special example of an algorithm portfolio. However, assuming only one algorithm, theoretical calculation of the expected value becomes possible. Furthermore, in molecular simulations, multiple algorithms cannot be used and parallel execution of a homogeneous algorithm is required to maintain the validity of the simulation.

In previous studies, the effect of speculative computation was theoretically analyzed using the state-transition matrix eigenvalues of the Markov process [9, 10]. However, in an actual problem, it is difficult to obtain an accurate state-transition matrix. Therefore, only the lower or upper bound can be considered. In this study, we attempt to analyze the effect of the speculative method as a simple probability problem without using a state-transition matrix and derive parameters that can be observed experimentally; i.e., we attempt to analyze the application of this method to a practical problem.

3 Effect of Speculative Computation

3.1 Algorithm

Various successful metaheuristics such as SA, Tabu search, and genetic algorithms (GA) have been proposed for combinatorial optimization problems. Such metaheuristics use the stochastic iterative improvement algorithm as the base. In the case of minimization problems, when the algorithm is run, the average value of the obtained solutions decreases

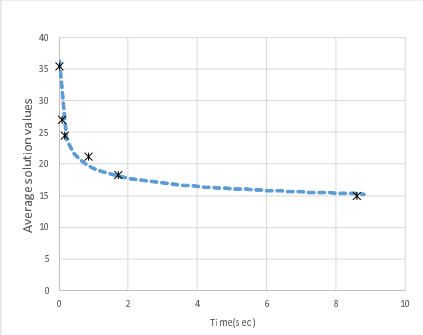

Figure 3. Relation between calculation time and obtained solution, and its approximate curve.

rapidly, but it becomes moderate immediately. If these algorithms run for a very long time, the obtained solution will approach the global optimal solution [16]. Fig.3 is an example of this situation, which is part of the experimental results in section 4. The horizontal axis is time, and the vertical axis is the obtained solution. The approximate curve is added to the figure in the form of Eq.(2).

$$\mu = a \cdot t^{-b} + \epsilon. \tag{2}$$

Here, let μ denote the average of obtained solution, t denote the time, ϵ denote the optimal solution, and a and b denote constants.

Since the Eq.(2) can be approximated better than polynomial approximation or log approximation in our experiment, we decided to use this approximation in this study. However, note that Eq.(2) is just an approximation because there is no mathematical proof.

In this study, we attempt to analyze the effect of the speculative computation method as a probability problem.

When the results of a trial S conform to a specific probability distribution, if the trial is repeated and the minimum value of the repeated trials is selected as trial M, the latter conforms to a probability distribution that differs from trial S. For example, if the scores are determined by a dice roll, the probability distribution becomes 1/6 each of $X = \{1, 2, 3, 4, 5, 6\}$ with an expected value of 3.5 for μ_s. If m dice are thrown simultaneously and the scores resulting from the minimum value are considered to be trial M, the probability distribution will be nonuniform; moreover, as the value of m increases, the expected value μ_m will become closer to 1.

This is the principle of speculative computation. In other words, if a stochastic algorithm is simultaneously executed m-parallel and the minimum value is selected as the result, it may be possible to obtain better computational performance.

```
21: Algorithm SSpeC
22: parallel-for(0 ≤ th < m) {
23:     ret[th] = S();
24: }
25: return selectBest(ret);
```

Figure 4. An example of simple speculative computing algorithm

In this study, a stochastic algorithm is used as the base and the following extremely simple speculative method is adopted. The algorithm is shown in Fig.4. A stochastic

algorithm is executed independently using multiple parallel processes (or threads), where a di erent initial value or di erent random number seed is used for each process (line 22-24). All parallel executions are terminated after a fixed time or fixed number of iterations, then, the best solution is selected from the obtained solutions (line 25).

The solution obtained by this method varies stochastically with parameter m. This method represents a simple Multi-Start, which we refer to as Stochastic Speculative Computation (SSpeC). SSpeC has the following features.

1. Process generation is performed once, and the process generation cost is low.
2. Synchronization is performed once, and the wait time is short.
3. The program utilizes nearly 100% of the available computational resources.
4. The program can be parallelized without modifying the original program.

3.2 Effect of improving a solution

It is assumed that the solution to a stochastic algorithm $\mathcal{S}$–follows the probability distribution of the distribution function $F_s(y)$–and the probability density function $f_s(y)$, where the minimum solution (or the optimal solution) is denoted as ϵ. $\mathcal{S}$–is executed in an m-parallel manner speculatively, and the minimum solution is selected from the obtained solutions. Note that m-parallel execution is assumed to be independent trial. In this case, the probability that solution y can be obtained via m-parallel execution follows the minimum value distribution of the original probability distribution according to the extreme value theory. The probability distribution function $F_m(y)$–and probability density function $f_m(y)$–· are expressed as follows:

$$F_m(y) = 1 - (1 - F_s(y))^m, \tag{3}$$

$$f_m(y) = m(1 - F_s(y))^{(m-1)} f_s(y). \tag{4}$$

To analyze the effect of speculative computation, we would like to express the expected value of this probability distribution as a function of m. Eq.(4) is an extreme value distribution with a lower limit (optimal solution). Therefore, when the approximation is performed using the type 3 asymptotic minimum value distribution, the probability distribution becomes the Weibull distribution and the expected value can be expressed as follows[3, 17].

$$\mu_m(m) = E(Y_m) = \epsilon + \frac{\delta}{m^{1/k}}\Gamma(1 + \frac{1}{k}). \tag{5}$$

Where ϵ is the minimum value, that is, the optimal solution, $\Gamma(\ )$ is a gamma function and δ and k are parameters that depend on the shape of the original probability distribution $f_s(y)$. If the expected values of the probability distribution $\mathcal{S}$ is μ_s, since $\mu_m(1) = \mu_s$, we simplify Eq.(5) as follows.

$$\mu_s = \epsilon + \frac{\delta}{1^{(1/k)}}\Gamma(1 + \frac{1}{k})$$

$$\mu_s - \epsilon = \delta\Gamma(1 + \frac{1}{k}). \tag{6}$$

We then substitute this into Eq.(5), and substitute $1/k$ for h ($h = 1/k$) to obtain the following:

$$\mu_m(m) = \epsilon + m^{-h}(\mu_s - \epsilon). \tag{7}$$

Similarly, the distribution $\sigma_m^2(m)$ is expressed as follows:

$$\sigma_m^2(m) = m^{-2h}\sigma_s^2. \tag{8}$$

Here, h is an index of the speculative computation effects, and a greater value of h results in greater speculative computation effects. From Eq.(7), the effects of speculative computation can be expressed as m^{-h} (the power function of m). The index of the effects of the speculative computation h is the inverse of k; therefore, the distribution will follow the shape of the original probability distribution. When the parallel number is $m \to \infty$, the expected value μ_m approaches ϵ asymptotically.

The results of our numerical analysis indicate that when the original probability distribution is a bell curve (similar to a normal distribution), h is in the range $0 < h < 1$. When the original probability distribution has a long tail (similar to geometric or exponential distributions), $h > 1$ may be observed. If $h > 1$, the effect of speculative computation is superlinear. In other words, h becomes large when the distribution of the original probability distributions is large. If the original probability distribution $F_s(x)$ is distributed uniformly, the distribution $F_m(x)$ of the speculative computation becomes a beta distribution and $h = 1$.

Generally, when a problem is difficult, the distribution of the solution of a stochastic algorithm becomes large, i.e., the effect of speculative computation is significant for difficult problems. Further, it is important that h, an index of the effect of speculative computation, is observable from experimental results.

3.3 Effect of reducing the execution time

When using a stochastic algorithm, it is assumed that the expected value of the computation time and solutions have a relation as that described in Eq.(2). In conventional parallel computing, it is assumed that computation time is reduced by Eq.(1). With the same computation time t_0, the conventional parallel computing is possible to calculate for a time longer by t_s/t_p times than non-parallel computing. Therefore, by substituting this into Eq.(2), the expected value of a solution can be expressed as follows:

$$\mu_p = a \cdot \left(\frac{t_s}{t_p} \cdot t_0\right)^{-b} + \epsilon. \tag{9}$$

On the other hand, with SSpeC, the expected value μ_m (Eq.(7)) of a solution is obtained using the same computation time t_0. For speculative computation to be more effective than the conventional parallelization method, μ_m must be smaller than μ_p.

$$a \cdot \left(\frac{t_s}{t_p} \cdot t_0\right)^{-b} + \epsilon \quad > \quad \epsilon + m^{-h}(\mu_s - \epsilon). \tag{10}$$

By substituting Eq.(1) and Eq.(2),

$$a \cdot \left(\frac{t_s \cdot t_0}{(\alpha/m + 1 - \alpha)t_s + \beta}\right)^{-b} + \epsilon \quad > \quad \epsilon + m^{-h}((a \cdot t_0^{-b} + \epsilon) - \epsilon). \tag{11}$$

We assume that conventional parallelization can be parallelized in an ideal state. In other words, all parts can be parallelized and the overhead is 0. Under ideal conditions, $\alpha = 1$,

$\beta = 0$, and $\epsilon = 0$ for simplicity, Eq.(11) can be transformed as follows:

$$a \cdot (mt)^{-b} \quad > \quad a \cdot t^{-b} \cdot m^{-h}. \tag{12}$$

Thus, the required condition is

$$b < h. \tag{13}$$

When Eq.(13) is satisfied, it is expected that the speculative computation result will be better than that obtained with conventional parallel computing. Note that b and h depend on the problem and algorithm combinations. However, b and h can be easily observed experimentally.

4 Experiment with a Combinatorial Optimization Problem

4.1 Experiment of comparison with conventional parallel processing

To quantify the effect of the speculative computation, we conducted experiments comparing the results with those of conventional parallel processing. We used the weighted constraint satisfaction problem (cost minimization), which is a kind of graph coloring problem. When nodes connected by constraints are assigned the same color, the weight of the constraint is added as a cost. It is a problem of searching for color assignment that minimizes the total cost. Here, the topology of the graph is random, the number of nodes is 100, the number of edges is 300, the weight is 1~5, and the number of colors is 3.

```
31: Algorithm S1
32: foreach(v_p[i]) { v_p[i] = randomInitialValue() }
33: minimumCost = eval(v_p);
34: for(N times) {
35:     foreach(v[i]){ /* parallelizable */
36:         cv = localEval(i,v_p[i], v_p);
37:         x = newValue();
38:         if(cv > 0 ){
39:             if(((localEval(i,x, v_p) < cv) && p(p_1)) || p(p_2)){
40:                 v_n[i] = x
41:             }
42:         }
43:     }
44:     nowcost = eval(v_n);
45:     if(nowcost < minimumCost){
46:         minimumCost = nowcost; memory(v_n);
47:     }
48:     v_p = v_n.clone()
49: }
50: return (minimumCost, restore()))
```

Figure 5. Simple parallel iterative improvement algorithm $\mathcal{S}1$

The base algorithm we used in the first experiment is shown in Fig.5. Basically it is an iterative improvement algorithm. Variable assignments are stored in array v_p (line 2). This algorithm improves each variables independently (line 35-43). The function localEval(i, x,

v_p) (line 36) calculates the local cost around the i-th variable, under the conditions that the variable assignment is v_p and the value of the i-th variable is x. The function newValue() returns a new assignment with a random number (line 37). The function p(x) is true with probability x. If i-th variable has a constraint violation (line 38), and it is improved with the new assignment, change the value with the probability p_1 (line 39-41). Or if this variable has a constraint violation (line 38), change its value with the probability p_2 (line 39-41). In this experiment, we used $p_1 = 0.3$ and $p_2 = 0.02$. When the minimum value of the evaluation result is updated, the minimum value and the assignment at that time are stored (line 45 - 47), finally return its minimum value and assignment (line 50). The changed value is saved in array v_n, copying v_n to v_p for each iteration (line 48), makes it possible to execute the loop of lines 35 - 43 in parallel. This algorithm will be referred to as $\mathcal{S}1$.

In the speculative computation, simply execute the base algorithm $\mathcal{S}1$ in parallel as shown in Fig.4, and select the best value. In conventional parallel processing, the loop of lines 35 - 43 of Fig.5 was executed in parallel.

The CPU used for this experiment is Intel Core i9 processor, 10 core 20 thread (Hyper threading). In the case of conventional parallel processing, the upper limit of the parallel number is logically the number of variables (= 100). However, in this experiment, the upper limit is 20 from the processing capacity of the processor.

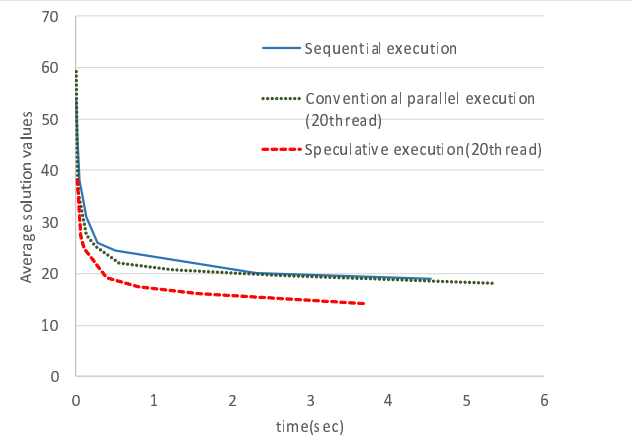

Figure 6. Comparison between conventional parallel processing and speculative computing

The experiment results are shown in Fig.6. The horizontal axis of this figure is the time and the vertical axis is the obtained solution values. The figure shows the results of sequential execution, conventional parallel execution with 20 thread, and speculative execution with 20 thread. Since this is a minimization problem, it is better that the curve is located downward in the figure. These results of Fig.6 confirm that, in this case, speculative computing produces better results than conventional parallel processing.

By CPU time analysis using a program profiler, α of the Eq.(1) is 0.65, when the number of N (line 34) is set to $100,000$.

Next, we calculate parameters h and b from the experimental results. Since the Eq.(7) for the parameter h is the relationship between the number of threads m and the obtained solution, h can be calculated by observing this relationship. Fig.7 shows the result of observing this relationship at the fixed $N = 1000$ iteration point. The parameter $h = 0.268$ in Eq.(7) was obtained from the approximate curve shown in Fig.7 by least squares method

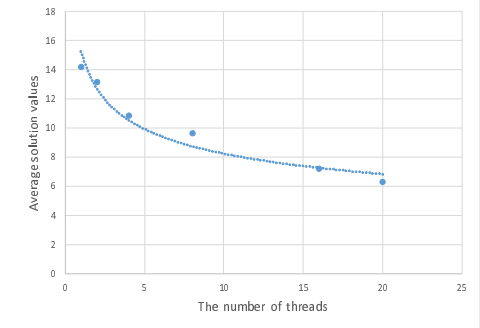

Figure 7. Relationship between degree of parallelism and solution in speculative computation

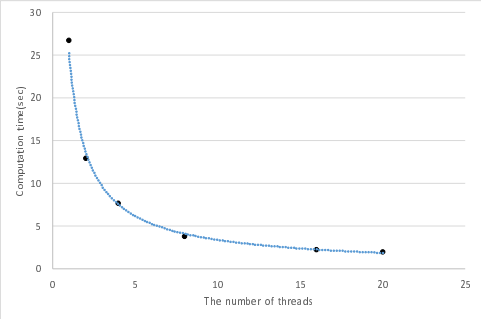

Figure 8. Relationship between parallel degree and computation time in speculative computation

($r^2 = 0.9534$). Since the parameter b in Eq.(2) is the relationship between the calculation time and the obtained solution when executing with a single thread, it can be obtained from the result of the Fig.6. From Fig.6, $b = 0.260$ ($r^2 = 0.9922$) was obtained. In this experimental result, $b \fallingdotseq h$. The reason why the speculative computation was better than the conventional parallel processing, as shown in Fig.6, although it is not $b < h$ (Eq.(13)), is that α and β were not ideal conditions.

In the above, we were paying attention to the effect of improving the solution obtained with the same calculation time. Next, we investigated the effect of shortening the time until the target solution was obtained. We set the solution target to 15, executed the program and stopped the execution when the solution $x : x \leq 15$ is obtained, then checked the computation time up to that point. The result is shown in Fig.8. The horizontal axis is the parallel number, and the vertical axis is the calculation time. From this curve, $h = 0.892$ was obtained. As for the time reduction, its effect was very great, but super linear was not realized.

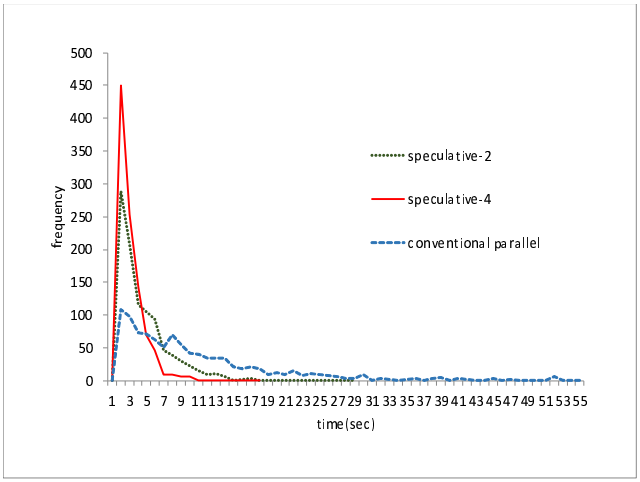

Figure 9. Distribution of computation time

Similarly, Fig.9 shows the distribution of the computation time until the target solution 15 is obtained. The horizontal axis is the computation time and the vertical axis is the

frequency. In conventional parallelization, the distribution has a long tail, on the other hand in speculative computation, increasing the degree of parallelism reduces the variance. This is the effect of Eq.(8).

In this experiment, although sufficient effect was not obtained by conventional parallelization, the effect of parallel execution was confirmed in speculative computation. These results are considered to depend on the performance of the base algorithm. Therefore, these results are just one example. However, it can be said that speculative computation can improve performance even for base algorithms with poor performance.

4.2 Difference in effect by algorithm

The effect of parallelization depends on the algorithm. Next, we conducted an experiment with an algorithm with fewer parallelizing elements. The base algorithm is shown in Fig.10.

```
61: Algorithm S2
62: foreach(v[i]) { v[i] = randomInitialValue() }
63: prevcost = mincost = eval(v);   /* partially parallelizable */
64: for(N times) {
65:     i = randomSelect()
66:     old = v[i];
67:     v[i] = newValue();
68:     nowcost = eval(v);   /* partially parallelizable */
69:     if((nowcost < prevcost) || p(p2) ){
70:         prevcost = nowcost;
71:         if(nowcost < mincost){
72:             mincost = nowcost; memory(v);
73:         }
71:     }
72:     else { v[i] = old }
73: }
74: return (mincost, restore())
```

Figure 10. Another simple iterative improvement algorithm $\mathcal{S}2$

This algorithm randomly selects one variable (line 65), change the value (line 67) if the evaluation is improved, restore it if it is not improved (line 72). Also change the value with probability p_2 (line 69) even if evaluation is not improved. This algorithm is widely used as the basis of the iterative improvement algorithm. The algorithm for Monte Carlo molecular simulation that we will deal with in the later section is also close to this algorithm. This algorithm will be referred to as $\mathcal{S}2$. The $\mathcal{S}2$ algorithm has few parallel executable parts by conventional parallelization. The function eval() (line 63, 68) is the only part that is parallelizable.

Experimental results using this algorithm are shown in Fig.11. In this experiment, if parallel execution is performed by conventional parallelization, the computation time becomes longer than sequential execution.

α of this algorithm is 0.74 (when $N = 1,000,000$). This is bigger than the $\mathcal{S}1$ algorithm. If α is large, it will be advantageous for conventional parallel processing. However, such

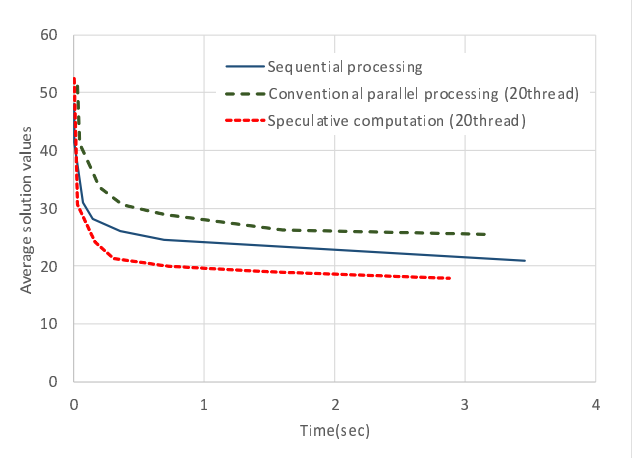

Figure 11. Comparison between conventional parallel processing and speculative computation using another algorithm

results were not obtained. Since this algorithm is simple, the ratio of the parallelizable part is relatively large. However, since the granularity of parallel execution is small, the overhead becomes relatively large, and the parallel effect seems to be reduced. A better solution was obtained by using speculative computation, even with such an algorithm.

4.3 Experiment of speculative computation using GPU

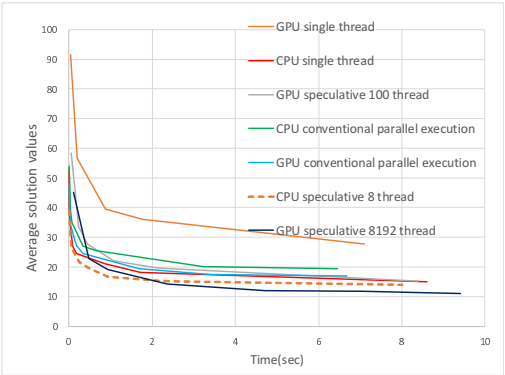

Figure 12. Effect of speculative computation using GPU

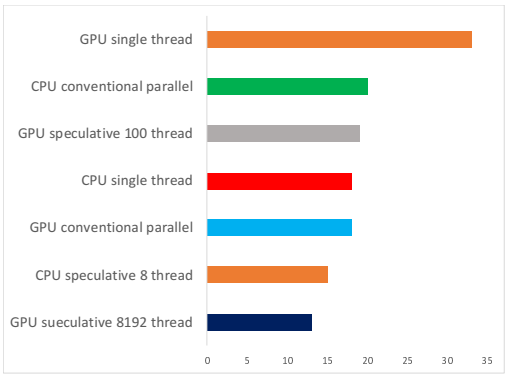

Figure 13. Comparison of solutions obtained at 4 seconds for each method

The GPU has many computation cores and can execute numerical calculations with high degree of parallelism. In this experiment, we tried speculative computation by placing the $\mathcal{S}1$ algorithm on each core of the GPU.

For this experiment, we used NVIDIA GeForce 1080Ti GPU with Core i7-3770 (4 core 8 thread) CPU. In the experiment, we compared CPU sequential execution, CPU speculative 8 thread execution, single thread execution using just GPU 1 core, GPU speculative 100 threads execution and 8192 thread execution, and conventional parallel processing using CPU (8 thread) and GPU (100 thread), Although the constraints from the number of computation

cores have been relaxed by using GPU, the degree of parallelism for conventional parallel processing is limited to 100 due to data parallelism constraint. From the number of CPU cores, the maximum number of CPU threads is limited to 8.

The result is shown in Fig.12. Fig.13 is a comparison of the obtained solutions at 4 seconds by each method read from Fig.12, smaller is better. From these figures, it is confirmed that computation of only one core of GPU is much slower than CPU, and it is not suitable for general purpose calculations like this. From this result, in conventional parallel processing, it was better to use GPU than to use only CPU. Probably because it was able to use GPU numerical computation and threads effectively. Compared to this, speculative computation was more efficient when 8 threads were executed on the CPU than 100 threads on the GPU. This is because speculative computation performed general calculations other than numerical calculations on the GPU. By fully using GPU computing resources and executing 8192 threads, we finally exceeded the 8-thread speculative computation on the CPU.

However, when using many threads, we must pay attention to memory usage. In speculative computation, each thread is independent, and synchronization is not needed. It also means that each thread needs an independent memory space. In the case of algorithms with large memory usage, limits may occur in the memory space rather than in the number of computational cores. Because this experiment used a simple program, the number of threads could be increased to 8192. In actual problems, it may be impossible to increase the number of threads to such a size. GPU core is fast for numerical calculations but slow for general calculations. In order to use it for the combinatorial optimization problem, that is our target, it is necessary to choose an algorithm carefully. If there are memory constraints, the effect of simple speculative computation using the GPU will be limited.

5 Conclusion

In previous studies [9, 10], the effect of speculative computation was calculated theoretically using the state-transition matrix of the Markov process. However, in practical problems, it is difficult to accurately obtain the state-transition matrix. Thus, in this paper, we investigated the effect of the speculative computation method theoretically based on probability theory. And we conducted experiments and compared the results with those of conventional parallel processing. We also performed experiments on GPU. In this experiment, we were able to confirm the existence of the case that the speculative computation is more effective than the conventional parallel processing. Conventional parallel processing has limitations on elements α and β in Eq.(1), and there are also limitations on number of data parallelism. It is difficult to prove these upper or lower bounds. Therefore, this experimental result does not prove the superiority of speculative computation. However, it can be said that it is easier to tune speculative computation than tuning conventional parallel processing. Tuning speculative computation is just increasing the number of threads to the limit of computational resources. Speculative computation should be reexamined now that many-core processors are becoming common in desktop computing. In the future, we would like to proceed with research on speculative algorithms that use CPU and GPU cooperatively based on this result.

References

[1] R. B. Osborne, Speculative computation in multilisp, pp. 103–137. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990.

[2] R. Martí, Multi-Start Methods, pp. 355–368. Boston, MA: Springer US, 2003.

[3] Y. Iizuka, A. Hamada, and Y. Suzuki, "Stochastic speculative computation method and its application to monte carlo molecular simulation," in Proceedings of Hawaii International Conference on System Sciences 2018, pp. 1660–1668, 2018.

[4] Y. Suzuki, A. Hamada, and Y. Iizuka, "Stochastic speculative computation method on general purpose graphics processing units," in 2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 1049–1050, July 2017.

[5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[6] A. Sohn, Z. Wu, and X. Jin, "Parallel simulated annealing by generalized speculative computation," in Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on, pp. 416–419, Dec 1993.

[7] K. L. Wong and A. G. Constantinides, "Speculative parallel simulated annealing with acceptance prediction," IEE Proceedings - Computers and Digital Techniques, vol. 143, pp. 219–223, Jul 1996.

[8] I. D. Falco, R. D. Balio, E. Tarantino, and R. Vaccaro, "Improving search by incorporating evolution principles in parallel tabu search," in 1994 IEEE Conference on Evolutionary Computation, pp. 823–828, 1994.

[9] R. Shonkwiler and E. Van Vleck, "Parallel speed-up of monte carlo methods for global optimization," Journal of Complexity, vol. 10, no. 1, pp. 64–95, 1994.

[10] X. Hu, R. Shonkwiler, and M. C. Spruill, Random restarts in global optimization. Georgia Institute of Technology, 2009.

[11] M. Samadi, A. Hormati, J. Lee, and S. Mahlke, "Paragon: Collaborative speculative loop execution on gpu and cpu," in Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, (New York, NY, USA), pp. 64–73, ACM, 2012.

[12] V. Krishnan and J. Torrellas, "Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor," in Proceedings of the 12th International Conference on Supercomputing, ICS '98, (New York, NY, USA), pp. 85–92, ACM, 1998.

[13] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," SIGPLAN Not., vol. 39, pp. 71–81, June 2004.

[14] L. Yan, “Solving combinatorial optimization problems with line-up competition algorithm,” Computers & Chemical Engineering, vol. 27, no. 2, pp. 251 – 258, 2003.

[15] C. P. Gomes and B. Selman, “Algorithm portfolios,” Artif. Intell., vol. 126, pp. 43–62, 2001.

[16] E. Aarts and J. Korst, Simulated annealing and boltzmann machines. New York, NY; John Wiley and Sons Inc., Jan 1988.

[17] W. Weibull et al., “A statistical distribution function of wide applicability,” Journal of applied mechanics, vol. 18, no. 3, pp. 293–297, 1951.