The Journal of Supercomputing (2022) 78:8733–8750
https://doi.org/10.1007/s11227-021-04242-0

Temporal and spatial parallel processing of simulated quantum annealing on a multicore CPU

Hasitha Muthumala Waidyasooriya[1] · Masanori Hariyama[1]

Accepted: 30 November 2021 / Published online: 13 January 2022

Abstract
Simulated quantum annealing (SQA) is a probabilistic approximation method to find a solution for a combinatorial optimization problem using a digital computer. It is possible to simulate large-scale optimization problems on a CPU due to its high external memory capacity. However, the processing time increases exponentially with the number of variables, and parallel implementation is difficult due to the serial nature of the quantum Monte Carlo algorithm used in SQA. In this paper, we propose a method to accelerate SQA on a multicore CPU, based on temporal and spatial parallel processing with high data localization. According to the experimental results using 16-core CPU, we achieved from 8 to 16 times speedup compared to single-core CPU implementations. The proposed method can be used to solve combinatorial optimization problems that have more than 64,000 variables, which was not possible using previous GPU- and FPGA-based accelerators.

Keywords Simulated quantum annealing · Optimization problems · Temporal parallelism · Multicore CPU

1 Introduction

Quantum annealing (QA) [1, 2] is a probabilistic approximation method to find the global optimum of a “combinatorial optimization problem” [3]. Solving large-scale combinatorial optimization problems is important in real-world applications such as traffic-flow simulation [4], financial analysis [5, 6], and graph problems [7, 8]. Quantum annealers such as D-wave [9] use quantum phenomena to find the global

✉ Hasitha Muthumala Waidyasooriya
hasitha@tohoku.ac.jp

Masanori Hariyama
hariyama@tohoku.ac.jp

1 Graduate School of Information Sciences, Tohoku University, 6-3-09, Aramaki-Aza-Aoba, Aoba, Sendai, Miyagi 980-8579, Japan

optimum. However, the number of bits available in quantum annealer is too small to solve many real-world problems. As a result, simulated quantum annealing (SQA) on digital computers is very important.

Quantum annealing in a transverse-field Ising model can be simulated on a digital computer [1]. ASICs (application specific integrated circuits) [10, 11], FPGAs (field programmable gate arrays) [12–16] and GPUs [17–19] are already employed to accelerate SQA. The FPGA and GPU accelerators proposed in [11–13, 17] restrict the interactions among spins to king-graphs or near-neighbor connections. Those accelerators are efficient for simulations that have a small number of interactions among spins (sparsely connected), and not efficient for simulations that have a very large number of interactions among spins (densely connected). When the spins are densely connected, minor embedding [20] is required, and that greatly reduces the number of usable spins [4, 21]. Studies such as [14, 19] are very successful in implementing simulations with densely connected spins. When the number of spins increases with the order of N, the number of interactions among spins increases with the order of N^2. Since interaction coefficients are stored in the external memory (DRAM), the number of spins that can be simulated is restricted by the external memory capacity. The external memory capacity of FPGAs and GPUs is very small compared to that of CPUs. Proposals in [15, 16] manage to solve large-scale "number partitioning problems" without storing all "interaction coefficients' in the memory.' The interaction coefficients are generated in parallel to the computation, without storing those in the external memory. However, "number partitioning" is the only known optimization problem that the interaction coefficients can be generated efficiently, while "MaxCut" problems, "graph coloring" problems, etc., do not work with this method. In order to conduct large-scale simulations of any arbitrary problem, we have to consider using CPUs. However, as reported in other studies [14, 19], the processing time on CPUs is extremely large even using multiple cores.

This paper proposes a novel method to reduce the processing time of SQA on multicore CPUs. The proposed method is inspired by studies [14, 19], that use both temporal parallelism and spatial parallelism. The novelty of this paper is the implementation of Trotter-slice-level parallelism on a multicore CPU while preserving the data dependency and reducing the synchronization. The Trotter-level-parallelism can be implemented in several granularity levels. One is the operation-level granularity which was done in the FPGA implementation of [14]. This method requires very fine-grain (clock-cycle level) scheduling. It is not suitable for multicore CPUs due to large synchronization overhead. Another method is spin-block-level synchronization used in the GPU implementation in [19]. However, this method works for GPUs that have thousands of threads, so that the computation time of a spin becomes too small compared the synchronization time. However, the number of cores in a CPU is very small compared to the parallel computing resources of a GPU and an FPGA. Therefore, how to allocate the cores optimally for temporal and spatial parallelism is important. In this paper, we applied spin-level granularity, discuss temporal parallel computation and a combination of temporal and spatial parallel computation and explain which method can be used according to the specifications of the optimization problem. The proposed method allows a good balance of parallel operations and synchronization overhead. We have achieved over 16 times speedup for large-scale simulations using 16-cores with the same accuracy level of the single-core CPU implementation. For small-scale problems, the processing time of the proposed method is very similar to that of GPU-based implementation in [19].

2 Implementation of simulated quantum annealing on a multicore CPU

2.1 Simulated quantum annealing (SQA)

Ising model is a mathematical model of a system consists of spins and their interactions. The energy or the Hamiltonian of an Ising model is expressed by Eq.(1).

$$H(\sigma) = -\sum_{ij} J_{i,j}\sigma_i\sigma_j - \sum_i h_i\sigma_i \tag{1}$$

A spin is denoted by σ, the interaction coefficient between spin i and j is denoted by $J_{i,j}$ and the local magnetic field is denoted by h_i. A spin has either the value -1 (spin-down) or the value +1 (spin-up). Transverse-field Ising model [1, 22, 23] is used to solve optimization problems. The transverse field controls the rate of transition between states and plays a similar role that the temperature does in simulated annealing [24]. By decreasing transverse field from a very large value to zero, we hopefully drive the system into the optimal state that has the lowest energy. The m-dimensional transverse-field Ising model can be mapped to a (m+1)-dimensional classical Ising model [25]. It uses multiple replicas called "Trotter slices."

```
1  Function one_spin_flip(m,i):
2      $local\_field \leftarrow 0$
       // compute local field
3      for $j \leftarrow 0$ to $N-1$ do
4          if $j < i$ then
5              $sp = spin(t, m, j)$
6          else
7              $sp = spin(t-1, m, j)$
8          end
9          $local\_field + = sp \times J(i,j)/M + sp \times J(j,i)/M$
10     end
       // compute transverse field
11     $local\_field += J_{tran} \times (spin(t-1, m+1, i) - spin(t, m-1, i))$
12     $energy\_diff = 2 \times spin(t-1, m, i) \times local\_field$
       // one spin flip
13     if $exp(-energy\_diff \times M) > rand\_num$ then
14         $spin(t, m, i) = \neg spin(t-1, m, i)$
15     end
16
17 Function main():
       // Monte Carlo step loop
18     for $t \leftarrow 1$ to $T$ do
           // Trotter slice loop
19         for $m \leftarrow 0$ to $M-1$ do
               // Spin loop
20             for $i \leftarrow 0$ to $N-1$ do
21                 one_spin_flip$(t, m, i)$
22             end
23         end
24     end
```

Algorithm 1: An extract of the SQA algorithm.

2.2 Spatial and temporal parallelism of quantum Monte-Carlo simulation

Quantum Monte Carlo simulation [26] is used to simulate the quantum tunneling phenomena of a transverse-field Ising model. It is shown in Algorithm 1. The "*one spin flip*" computation from lines 1 to 10 is repeated for all spins in a Trotter slice, all Trotter slices and all Monte Carlo steps (time steps). The number of spins, the number of Trotter slices and the number of Monte Carlo steps are denoted by N, M and T, respectively. The term *one spin flip*(t, m, i) represents the computation belonging to spin i of Trotter-slice m at the Monte Carlo step t. The transverse field is denoted by J_{trans}. As we can see in line 4, the computation of the *local_field* of a Trotter slice is a summation operation. We cannot process two spins in the same Trotter slice in parallel, since the value of the previous spin $i-1$ required by the computation of spin i. Spins of two neighboring Trotter slices are required for the computation of "*one spin flip*" of a Trotter slice. In this algorithm, *local_field* computation considers all possible connections among all N spins. Therefore, it is possible to process both sparsely and densely connected spins. When there is no interaction between two spins, the corresponding interaction coefficient is set to zero. However, computations are performed even an interaction coefficient is zero.

3 Optimizing parallel computations in a multicore CPU

3.1 Spatial parallelism of local field computation

As shown in Algorithm 1, the computation of the *local field* is a series of multiplications and additions, where the spin values and the interaction coefficients are multiplied and then added together. We can apply "parallel reduction" on multiple cores to decrease the processing time. Algorithm 2 shows how to perform the reduction operation in parallel using multiple threads.

```
// compute local field
#pragma omp parallel for reduction(+: $local\_field$)
for $j \leftarrow 0$ to $N-1$ do
    if $j < i$ then
        $sp = spin(t, m, j)$
    else
        $sp = spin(t-1, m, j)$
    end
    $local\_field += sp \times J(i,j)/M + sp \times J(j,i)/M$
end
```

Algorithm 2: Parallel reduction on *local field* computation.

3.2 Temporal parallelism of multiple Trotter slices

```
// Monte Carlo step loop
for $t \leftarrow 1$ to $T$ do
    for $y \leftarrow 0$ to $N + M - 1$ do
        #pragma omp parallel
        begin
            $threadID$ = get thread number
            for $x \leftarrow M/cores - 1$ to $0$ do
                $m = threadID + (x \times cores)$
                $i = y - m$
                // one spin flip
                if $i >= 0$ $AND$ $k < N$ $AND$ $m < M$ then
                    // compute local field
                    $local\_field \leftarrow 0$
                    for $j \leftarrow 0$ to $N - 1$ do
                        if $j < i$ then
                            $sp = spin(t, m, j)$
                        else
                            $sp = spin(t - 1, m, j)$
                        end
                        $local\_field+ = sp \times J(i, j)/M + sp \times J(j, i)/M$
                    end
                    // compute transverse field
                    $local\_field$ += $J_{tran} \times (spin(t - 1, m + 1, i) - spin(t, m - 1, i))$
                    $energy\_diff = 2 \times spin(t - 1, m, i) \times local\_field$
                    // one spin flip
                    if $exp(-energy\_diff \times M) > rand\_num$ then
                        $spin(t, m, i) = \neg spin(t - 1, m, i)$
                    end
                end
            end
            #pragma omp barrier
        end
    end
end
```

Algorithm 3: Trotter slice-level temporal parallel computation.

```
$BlockSize = cores/M$
for $t \leftarrow 1$ to $T$ do
    for $y \leftarrow 0$ to $N + M - 1$ do
        #pragma omp parallel begin
            $threadID$ = get thread number
            $m = threadID/BlockSize$
            $i = y - m$
            // local energy computation
            if $i >= 0$ AND $k < N$ AND $m < M$ then
                // partially compute local field per thread block
                for $tid \leftarrow 0$ to $BlockSize - 1$ do
                    if $threadID = tid + BlockSize \times m$ then
                        $LF[threadID] \leftarrow 0$
                        for $j = tid \times N/BlockSize$ to
                          $(tid + 1) \times N/BlockSize - 1$ do
                            if $j < i$ then
                                $sp = spin(t, m, j)$
                            else
                                $sp = spin(t - 1, m, j)$
                            end
                            $LF[threadID]+ = sp \times J(i, j)/M + sp \times J(j, i)/M$
                        end
                    end
                end
            end
            #pragma omp barrier
            if $threadID = BlockSize \times m$ AND $k < N$ AND $m < M$ then
                for $j = 0$ to $BlockSize - 1$ do
                    $local\_field+ = LF[threadID + j]$
                end
                // compute transverse field
                $local\_field += J_{tran} \times (spin(t - 1, m + 1, i) - spin(t, m - 1, i))$
                $energy\_diff = 2 \times spin(t - 1, m, i) \times local\_field$
                // one spin flip
                if $exp(-energy\_diff \times M) > rand\_num$ then
                    $spin(t, m, i) = \neg spin(t - 1, m, i)$
                end
            end
        end
    end
end
```

Algorithm 4: Trotter-slice-level temporal and spatial parallel computation.

Let us consider the data dependency among multiple Trotter slices of Algorithm 1. There are *M* Trotter slices computed one after the other. The computation of *spin i* of Trotter slice *m* at the Monte Carlo step *t* requires *spin i* of Trotter slice *m-1* of the same Monte Carlo step and *spin i* of Trotter slice *m+1* of the previous Monte Carlo step $t - 1$. This data dependency is shown in Fig. 1. The computation of spin *i* of Trotter Slice *m* at the Monte Carlo step *t* is given by *spin*(*t*, *m*, *i*). The computation of *spin*(*t*, *m*, *i*) requires the data of $spin(t, m - 1, i)$ and $spin(t - 1, m + 1, i)$. Since $spin(t - 1, m + 1, i)$ is already computed in the previous Monte Carlo step, we

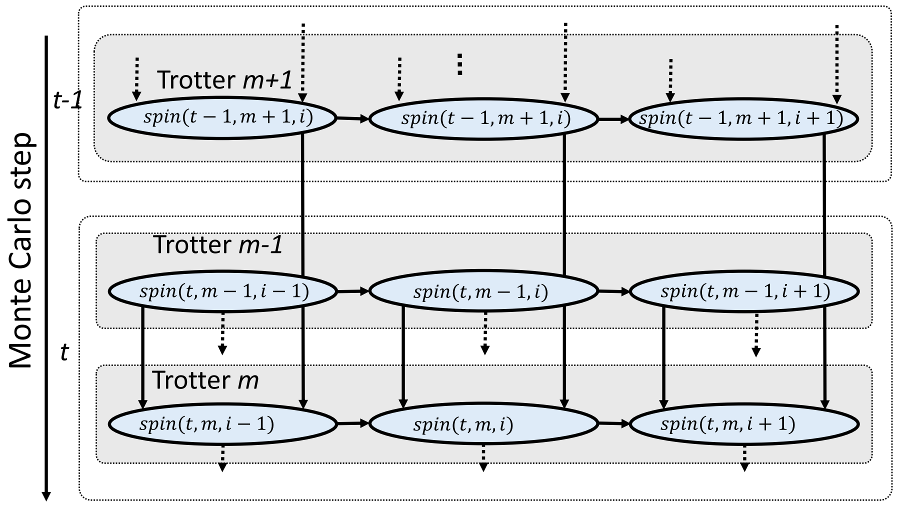


Fig. 1 Data dependency of spin i among multiple Trotter slices. Note that, only a small portion of the data dependencies among other spins are shown

have to schedule the computation of $spin(t-1, m+1, i)$ after the computation of $spin(t, m-1, i)$ is completed. As a result, we can compute $spin(t, m-1, i+1)$ and $spin(t, m, i)$ in parallel without violating the data dependency. We can schedule rest of the computations of spins among Trotter slices in parallel in similar manner. We call this method Trotter-slice-level temporal parallelism.

We can compute mutually independent computations among multiple Trotters slices using multiple threads. The computation of three Trotter slices in parallel using three threads is shown in Fig. 2. Threads 0, 1 and 2 are assigned to spin $i+1$ of Trotter slice 0, spin i of Trotter slice 1 and spin $i-1$ of Trotter slice 2, respectively. We have to synchronize the data after all the computation of all three threads to preserve the data dependency.

Algorithm 2 shows the proposed implementation method to realize Trotter-slice-level temporal parallelism using multiple threads. When the number of Trotter slices is greater than (and a multiple of) the number of cores, a set of Trotters slices are processed in parallel, and such sets are processed in serial. This is represented by the loop at line 5. If the number of Trotter slices equals to the number of cores, $M/cores = 1$, so that the loop in line 5 executes only once. When the number of Trotter slices is smaller than the number of cores, we only use a portion of the cores

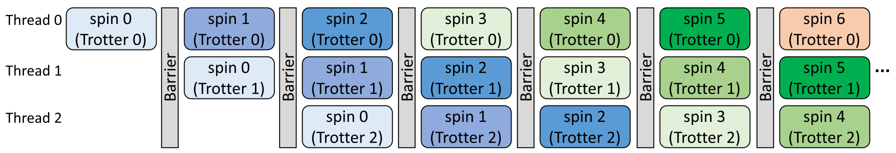


Fig. 2 Parallel processing of spins belonging to three different Trotter slices

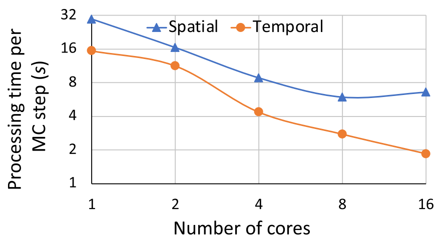
which equals the number of Trotter slices. Therefore, $M/cores = 1$, and the loop in line 5 executes only once. From lines 6 to 18, each thread is assigned to a Trotter slice and multiple threads are processed in parallel for different spins as shown in Fig.2. For example, "*one spin flip*" of spins (0, i), $(1, i - 1)$ ... $(M - 1, i - M + 1)$ is computed in parallel using M threads. On the other hand, the same spin is computed in different Trotter slices in serial to preserve the data dependency. Since all Trotter slices use the same interaction coefficients for the same spin, the same coefficients are accessed more often. This increases the data locality of the coefficients.

As explained in the above method, when the number of Trotter slices equals or greater than (and a multiple of) the number of cores, all the cores are used. When the number of Trotter slices is smaller than the number of cores, all the cores cannot be used for temporal parallel computation. In this case, we can use the remaining core for spatial parallel computation of the *local field* energy similar to Algorithm 2. The combined temporal and spatial parallel computation is shown by Algorithm 4. In this method, we assume that the number of cores is a multiple of the number of Trotter slices. Therefore, for each Trotter slice, we consider a *thread block* of *cores*/*M* as shown in line 1 of Algorithm 4 , where *M* is the number of Trotter slices. The *local field* energy computation is done per each *thread block* as shown in lines from 11 to 13. Since there are multiple threads in a *thread block*, we divide the computation into the number of threads per a *thread block* and compute in parallel. A barrier is placed until the partial computation of *local field* energy for all threads blocks is completed. Then, per each *thread block*, we add all the partial *local field* energies together as shown in lines 19 to 21. The temporal parallel computation is done in *thread block* basis while preserving the data dependency, similar to Algorithm 3 . The spatial parallel computation is done within a *thread block*. Since the *thread blocks* are properly synchronized, the data dependency is preserved.

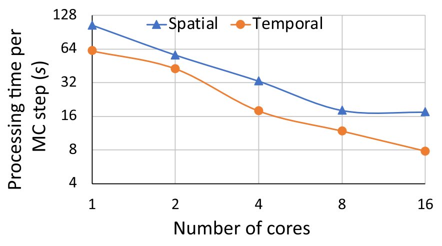
4 Evaluation

The evaluation is done using 16-core Intel Xeon Gold 6242 CPU with 192 GB of system memory. The L3 cache memory size is 22MB. The operating system is CentOS 7.9. CPU codes are compiled using Intel compiler version 19.1.3.304. Optimization options are "O3, xHost" and "qopenmp."

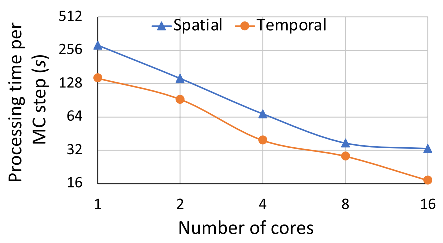
Figure 3 shows the processing time comparison of different multicore implementations. One implementation uses only the spatial parallelism explained in Algorithm 2 . We consider this as a conventional method, since this is a straightforward implementation of parallel computation on simulated quantum annealing. The other implementation is the proposed spatial and temporal parallelism. Figure 3a, b and c shows the processing times for 65,536, 131,072 and 196,608 spins, respectively. The number of Trotter slices is 16. The processing time of the conventional method is saturated at 8 cores. However, we can see a continuous reduction in the processing time from 1-core to 16-cores in the proposed implementation. We can expect further processing time reduction by using a CPU with more cores. The "16-core implementation of the proposed method" is nearly 8



(a) Number of spins = 65,536.



(b) Number of spins = 131,072.



(c) Number of spins = 196,608.

Fig. 3 Processing time comparison of different parallel implementations for optimization problems with a large number of spins. Number of Trotter slices=16

times and 16 times faster compared to "1-core implementation of the proposed method" and "1-core implementation of the conventional method," respectively.

Interestingly, when we use only a single core, the processing time of proposed method is nearly half of that of conventional method. In this case, there is no multicore parallelism in either of the implementations since we used only one CPU core. In the conventional method, the computations are scheduled in Trotter slice-serial manner, where all the spins belonging to one Trotter slice are computed and then moved to the computation of the next Trotter slice. However, the proposed method uses Trotter-slice-parallel approach where the same spin of different Trotter slices is computed as early as possible. When the spin is the same, the same coefficient data are accessed over and over, which results in a better data localization. As a result, already cached coefficient data of that particular spin can be used.

Figure 4 shows the speedup against the number of cores for 196,608 spins and 16 Trotter-slice implementation. The speedup of using p CPU cores is denoted by $S(p)$ and calculated by Eq. (2). The processing times of 1-core and p-core implementations are denoted by $T(1)$ and $T(p)$, respectively. According to the results, the speedup decreases with the number of cores and stays under the linear speedup line of $S(p) = p$. This speedup curve is categorized as a sublinear speedup of $S(p) = \alpha p^{\beta}$, where $0 < \alpha < 1$ and $0 < \beta < 1$ as defined in work [27]. Figure 5 shows the efficiency of the proposed implementation. Efficiency of using p cores is denoted by $E(p)$ and calculated by Eq. (3). Efficiency is 0.52 while using 16 cores, which we think a good value. As explained in [27], it is impossible to achieve 100% efficiency, and it decreases with the number of cores.

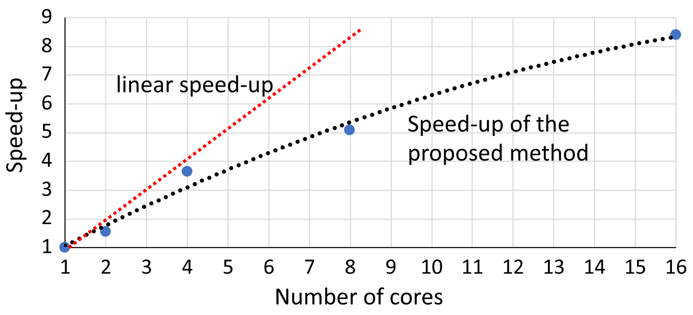


Fig. 4 Speedup against the number of cores for 196,608 spins and 16 Trotter-slice implementation

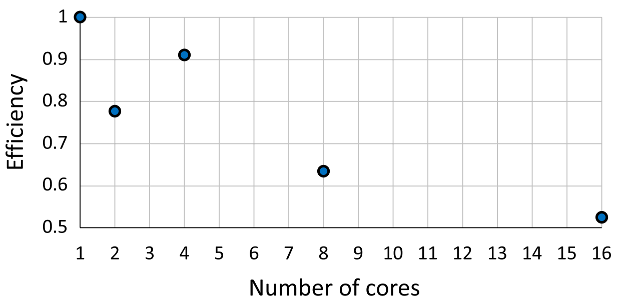


Fig. 5 Efficiency against the number of cores for 196,608 spins and 16 Trotter-slice implementation

$$S(p) = T(1)/T(p) \tag{2}$$

$$E(p) = T(1)/(p \times T(p)) \tag{3}$$

Figure 6 shows the processing time against the number of Trotter slices. We evaluate both "temporal parallel implementation" and "temporal and spatial combined implementation." When the temporal parallel computation is used, the processing time is nearly a constant up to 16 Trotter slices for all spin sizes. Since the CPU has 16 cores, we can implement up to 16 Trotter slices in parallel. When the number of Trotter slices is less than the number of cores, some of the cores remain unused. We use those cores for spatial parallel computation, as proposed in Algorithm 4 and the evaluation shows a considerable reduction in processing time from 4 to 8 Trotter slices, compared to that of the "temporal parallel implementation." As shown in Fig. 6a, b and c, the processing time increase is near-linear for "temporal and partial combined implementation." This shows that all 16 cores are fully utilized for all different Trotter slices sizes. More processing time reduction can be achieved by using more cores.

Although the proposed method is developed for large-scale problems, we can use it for medium-scale problems also. Previous accelerators proposed using FPGAs [14] and GPUs [19] can process only up to 32,768 spins due to memory capacity

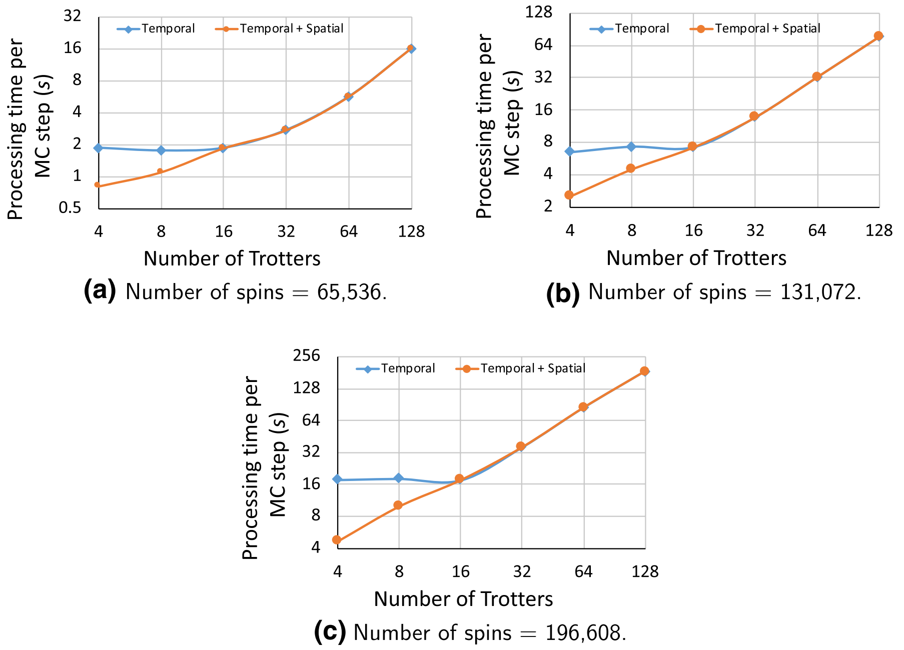
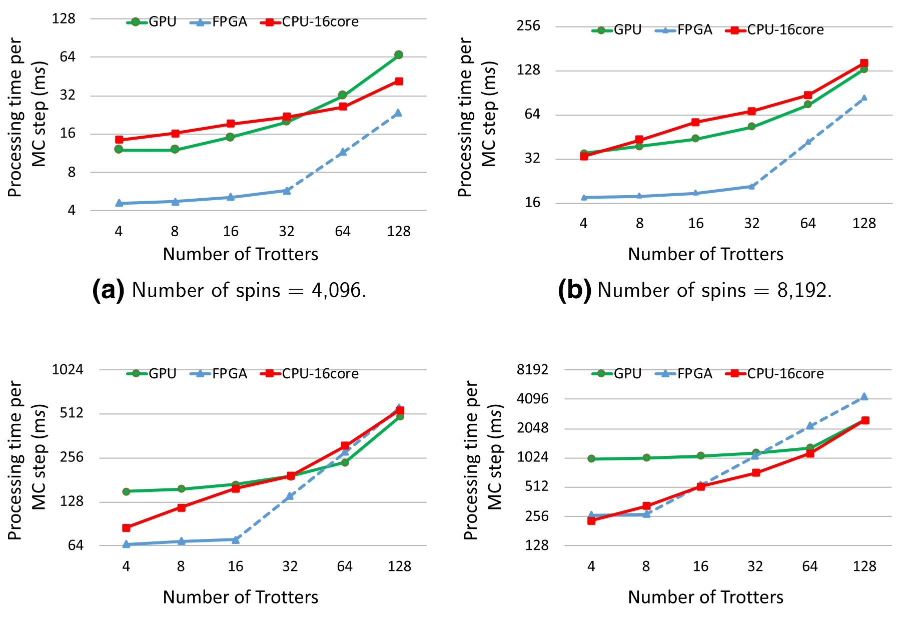


Fig. 6 Number of Trotter slices vs. processing time for optimization problems with a large number of spins

constraint. We can compare our method with those works using such medium-scale simulations. Fig.7 shows the comparison with GPU and FPGA implementations for $4,096 \sim 32,382$ spin sizes. The FPGA is Intel Arria 10 10AX115N3F40E2SG and the GPU is NVIDIA GV100. The FPGA implementation is the best for small spin sizes such as 4,096, 8,192 and 16,384 spins as shown in Fig. 7a, b and c, respectively. This is because of two reasons. One is the zero synchronization overhead in FPGAs. When the number of spins is small, the processing time is also small. As a result, the synchronization overhead of both CPUs and GPUs is not negligible compared to the computation time. In FPGAs, all operations are scheduled precisely for each clock cycle to preserve the data dependency, so that the explicit synchronization is not required. Another reason is the high temporal parallelism in FPGA for small spin sizes. When the number of spins is small, less resources are required for FPGAs to implement parallel computation among Trotter slices. Therefore, all the Trotter slices are processed in parallel, and the processing time is nearly the same even the number of Trotter slices is increased. However, when the number of spins increases, more FPGA resources are required to implement parallel computation. When there are no more resources for parallel computation of Trotter slices, some of the Trotter slices have to be processed in serial in the FPGA. The dashed lines in Fig. 7 show the approximated processing time, where some of the Trotter-slice-blocks are processed in serial. A Trotter-slice-block contains one or more Trotter slices. In this case, it is fair to assume that we can add the processing times of the Trotter-slice-blocks when those are processed one after the other in serial.



(a) Number of spins = 4,096. **(b)** Number of spins = 8,192. **(c)** Number of spins = 16,384. **(d)** Number of spins = 32,768.

Fig. 7 Comparison against GPU and FPGA implementations using medium-size optimization problems. Dashed-lines show the approximated processing time of FPGA implementation, that is calculated under the assumption of processing multiple Trotter slice-blocks serially and the Trotter slices in a block in parallel

The processing time of the proposed CPU implementation is very similar to that of the GPU implementation. When the number of spins increased to 32,768, the processing time of CPU is better than that of GPU and even closer to that of FPGA. Note that, both CPU and GPU are roughly from the same production era. However, [14] uses old generation Arria 10 FPGA, so that the comparison with FPGA is lightly bias toward CPU. If we used a newer generation FPGA such as Startix 10 or Agilex 10, the FPGA processing speed can be better than that of the CPU. However, there is no significant improvement of the memory capacity even for newer generation FPGAs. Therefore, the proposed method is extremely important to compute large-scale simulations. In addition, recent 64-core CPUs show a great potential for the proposed method to provide extremely large speedup.

Table 1 shows the accuracy comparison of the proposed method using "Gset" [28] benchmarks that have sparsely connected spins. We used the "best known cuts" values from [19]. Those were obtained by investigating several previous studies such as [29–32]. The results of single-core CPU are also taken from [19]. In this evaluation, each benchmark is simulated for 100 samples, where each sample is executed for 1000 Monte Carlo steps (MC steps). The initial data are randomly generated. The number of Trotters is 16 for all implementations. According to the results, the accuracy of the average cut is more than 97% compared to the "best know cut." We can also see that the best and average cuts are similar to those in single-core

Table 1 Comparison of accuracy using MaxCut problems with sparsely connected nodes

Benchmark	Number of		Best	CPU single-core [19]		Proposed	
	Nodes	Edges	Known	1000 MC steps		1000 MC steps	
			Cut	Best	Average	Best	Average
G9	800	19,176	2054	2037	2006 (97.7%)	2050	2010 (97.9%)
G13	800	1600	**582**	**582**	580 (99.7%)	**582**	580 (99.7%)
G18	800	4694	992	988	974 (98.2%)	987	974 (98.2%)
G19	800	4661	906	903	889 (98.1%)	902	890 (98.2%)
G20	800	4672	**941**	**941**	926 (98.4%)	**941**	926 (98.4%)
G21	800	4667	931	929	913 (98.1%)	930	911 (97.9%)
G31	2000	19,990	3309	3278	3241 (97.9%)	3274	3233 (97.7%)
G34	2000	4000	**1384**	**1384**	1378 (99.6%)	**1384**	1378 (99.6%)
G39	2000	11,778	2408	2379	2345 (97.4%)	2377	2343 (97.3%)
G40	2000	11,766	2400	2379	2338 (97.4%)	2388	2338 (97.4%)
G41	2000	11,785	2405	2394	2337 (97.2%)	2394	2336 (97.1%)
G42	2000	11,779	2481	2452	2410 (97.1%)	2451	2413 (97.3%)
G47	1000	9990	6657	6656	6628 (99.6%)	6653	6628 (99.6%)
G50	3000	6000	**5880**	**5880**	5876 (99.9%)	**5880**	5878 (99.9%)
G51	1000	5909	3848	3845	3828 (99.5%)	3844	3827 (99.5%)
G53	1000	5914	3850	3842	3830 (99.5%)	3844	3831 (99.5%)
G54	1000	5916	3852	3848	3827 (99.4%)	3845	3828 (99.4%)

The values indicated in bold equal to the best known cut

implementations. The slight difference is caused by the initial random solution and random values used for “one spin flip” computation. This shows that there is no accuracy degradation in the proposed parallel implementation, and this is due to the preserving of the data dependency.

Table 2 shows the accuracy comparison of “MaxCut” problems, with densely connected spins. These benchmarks are generated randomly with 80%, 90%, 99% and 100% edge densities. Each benchmark is simulated for 100 samples, where each sample is executed for 4096 Monte Carlo steps (MC steps). According to the results, both single-core implementation and multicore implementation provide similar results.

Figure 8 shows the power efficiency and energy comparison against GPU and FPGA implementations. We use 32,768 spins since that is the largest number of spins that both GPU and FPGA can compute. The power efficiency is calculated by dividing the speedup by power consumption. The speedup is calculated relative to the single-core CPU implementation given in [19]. The power efficiency values of GPU and FPGA are taken from [19], so that all values are calculated relative to same reference. Note that, we used the thermal design power (TDP) value of the CPU as the power consumption, since we do not have an accurate way of calculating it. According to Intel reference in [33], TDP represents the average power in watts, the processor dissipates when operating at base frequency with all cores

Table 2 Comparison of accuracy using MaxCut problems with densely connected nodes

Benchmark	Number of		CPU single-core [19]		Proposed	
	Nodes	Edges	4096 MC steps		4096 MC steps	
			Best	Average	Best	Average
RG4096-80	4096	6,709,248	85,906	84,895	85,955	84,922
RG4096-90	4096	7,547,904	92,390	91,139	92,619	91,092
RG4096-99	4096	8,302,694	94,197	92,528	93,627	92,601
RG4096-100	4096	8,386,560	95,065	94,105	95,506	94,043
RG8192-80	8192	26,840,268	244,834	246,934	244,900	246,773
RG8192-90	8192	30,195,302	254,093	256,634	253,933	256,658
RG8192-99	8192	33,214,832	268,458	270,601	268,630	270,568
RG8192-100	8192	33,550,336	271,557	274,128	271,490	274,020

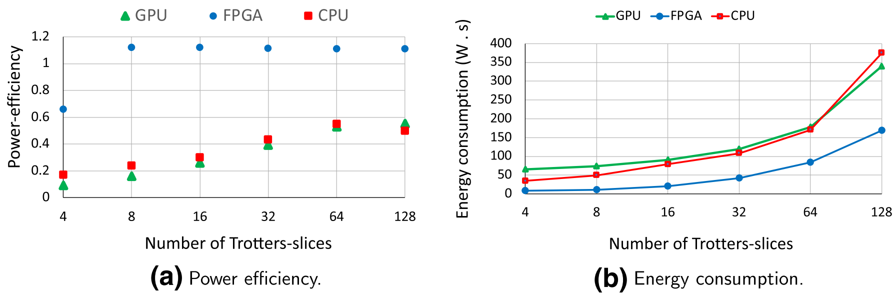


Fig. 8 Power efficiency and energy Comparisons against GPU and FPGA implementations using 32,768 spin implementation. Max TDP is used for the CPU power consumption, while measured power consumption data of GPU and FPGA implementations are taken from [19]

active under an Intel-defined, high-complexity workload. The energy consumption is calculated by multiplying the power and processing time. According to the results in Fig. 8a, the power efficiency of both CPU and GPU is very similar and much lower compared to that of FPGA. We can see similar results for energy consumption in Fig. 8b, where FPGA has the lowest energy consumption. There is a small trend that the energy consumption of CPU implementation increases more rapidly with the number of Trotter slices compared to that of GPUs. This can be due to the better resources utilization of more than 5,000 cores in GPU when processing more Trotters in parallel.

The accelerator proposed in [14] uses multiple FPGAs to increase the processing speed by the factor of "number of FPGAs." However, this method only increases the processing speed but not the problem size. The problem size is still limited by the external memory of one FPGA, and no method has been proposed to increase problem size using multiple FPGAs. Although there is possibility to increase the processing speed using multiple GPUs, no such work has been proposed. Efficient data transfer method among multiple GPUs is required to increase the processing speed and problem size using multiple GPUs. Technologies such as “NVLink” [34] should be exploited in future to realize multi-GPU accelerators. Another possibility for FPGAs and GPUs to compute optimization problems with a large number of spins is to use “out-of-core” data storage and transfer data on-demand. However, relatively small PCIe bandwidth between CPU and accelerators prevents us from doing this efficiently even using current generation high-end accelerators. Another possible scenario is to use very large “unified memory,” where both accelerator and CPU can access the same memory. If the accelerator has the access to the large CPU memory without using the PCIe bus, it is possible to compute extremely large optimization problems.

5 Conclusion

We have proposed a highly parallel CPU accelerator for SQA by exploiting the Trotter slice-level temporal parallelism and spatial parallelism, while preserving the data dependency. Proposed temporal and spatial parallel implementation can be used for both dense and sparse spin models without compromising on accuracy. However, the processing time only depends of the number of nodes and the number of Mote Carlo steps, and independent of the number of edges. Therefore, if the number of nodes and the number of Monte Carlo steps are the same, both sparse and dense models require the same processing time. The proposed method prioritizes temporal parallelism over spatial parallelism. However, it uses spatial parallelism when there are idle cores that are not used for temporal parallel computations. Moreover, the proposed method localizes the memory access, which leads to potential bandwidth savings. We can expect further processing time reduction if we can use CPUs with more cores.

The “16-core implementation of the proposed method” is nearly 8 times and 16 times faster compared to “1-core implementation of the proposed method” and “1-core implementation of the conventional method,” respectively. The processing speed of 16-core implementation is similar to that of a high-end GPU. If we compare against a current generation FPGA, the FPGA may provide better performance for small problems. Since we cannot use FPGAs and GPUs for large-scale problems due to relatively small memory capacity, the proposed CPU-based implementation is very important.

Acknowledgements This research is partly supported by MEXT KAKENHI, grant numbers 19K11998 and 20H04197.

References

1. Kadowaki T, Nishimori H (1998) Quantum annealing in the transverse Ising model. Phys Rev E 58(5):5355

2. Kadowaki T (1998) Study of optimization problems by quantum annealing, Ph. D. Dissertation, Department of Physics, Tokyo Institute of Technology
3. Schrijver A (2003) Combinatorial optimization: polyhedra and efficiency. Springer, Berlin
4. Neukart F, Compostella G, Seidel C, von Dollen D, Yarkoni S, Parney B (2017) Traffic flow optimization using a quantum annealer. Front ICT 4:29
5. Orús R, Mugel S, Lizaso E (2019) Quantum computing for finance: overview and prospects, Reviews in Physics, p. 100028
6. Elsokkary N, Khan FS, La Torre D, Humble TS, Gottlieb J (2017) Financial Portfolio Management using D-Wave Quantum Optimizer: The Case of Abu Dhabi Securities Exchange, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep
7. Titiloye O, Crispin A (2011) Quantum annealing of the graph coloring problem. Discrete Optim 8(2):376–384
8. Ushijima-Mwesigwa H, Negre CF, Mniszewski SM (2017) Graph partitioning using quantum annealing on the d-wave system, in Proceedings of the Second International Workshop on Post Moores Era Supercomputing. ACM, pp. 22–29
9. D-wave (2019) https://www.dwavesys.com
10. Lanting T, Przybysz AJ, Smirnov AY, Spedalieri FM, Amin MH, Berkley AJ, Harris R, Altomare F, Boixo S, Bunyk P et al (2014) Entanglement in a quantum annealing processor. Phys Rev X 4(2):021041
11. Yamaoka M, Yoshimura C, Hayashi M, Okuyama T, Aoki H, Mizuno H (2016) A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing. IEEE J Solid-State Circuits 51(1):303–309
12. Okuyama T, Hayashi M, Yamaoka M (2017) An Ising Computer Based on Simulated Quantum Annealing by Path Integral Monte Carlo Method, in IEEE International Conference on Rebooting Computing (ICRC). IEEE, pp. 1–6
13. Waidyasooriya HM, Araki Y, Hariyama M (2018) Accelerator Architecture for Simulated Quantum Annealing Based on Resource-Utilization-Aware Scheduling and its Implementation Using OpenCL, in International Symposium on Intelligent Signal Processing and Communication Systems (ISPAC), pp. 336–340
14. Aidyasooriya H, Hariyama M (2019) Highly-parallel fpga accelerator for simulated quantum annealing, IEEE Transactions on Emerging Topics in Computing
15. Liu C-Y, Waidyasooriya HM, Hariyama M (2019), Data-transfer-bottleneck-less architecture for FPGA-based quantum annealing simulation, in, (2019) Seventh International Symposium on Computing and Networking (CANDAR). IEEE 2019: 164–170
16. Liu Y, Waidyasooriya HM, Hariyama M (2021) Design space exploration for an FPGA-based quantum annealing simulator with interaction-coefficient-generators, The Journal of Supercomputing, pp. 1–17
17. Weigel M (2012) Performance potential for simulating spin models on GPU. J Comput Phys 231(8):3064–3082
18. Cook C, Zhao H, Sato T, Hiromoto M, Tan SXD (2018) GPU based parallel Ising computing for combinatorial optimization problems in VLSI physical design, arXiv preprint arXiv:1807.10750
19. Waidyasooriya HM, Hariyama M (2020) A GPU-Based Quantum Annealing Simulator for Fully-Connected Ising Models Utilizing Spatial and Temporal Parallelism, IEEE Access, vol. 8, pp. 67 929–67 939
20. Zaribafiyan A, Marchand DJ, Rezaei SSC (2017) Systematic and deterministic graph minor embedding for cartesian products of graphs. Quantum Inf Process 16(5):136
21. Booth M, Reinhardt SP (2017) A. Roy, Partitioning optimization problems for hybrid classical/quantum execution, D-wave technical report series, pp. 01–09
22. Stinchcombe R (1973) Ising model in a transverse field I. Basic theory. J Phys C Solid State Phys 6(15):2459
23. Pfeuty P, Elliott R (1971) The Ising model with a transverse field, II. Ground state properties. J Phys C Solid State Phys 4(15):2370
24. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680
25. Suzuki M (1976) Relationship between d-dimensional quantal spin systems and (d+1)-dimensional Ising systems: equivalence, critical exponents and systematic approximants of the partition function and spin correlations. Progress Theor Phys 56(5):1454–1469

26. Suzuki M, Miyashita S, Kuroda A (1977) Monte Carlo simulation of quantum spin systems. I. Progress Theor Phys 58(5):1377–1387
27. Heath MT (2015) A tale of two laws. Int J High Perform Comput Appl 29(3):320–330
28. Gset (2019) https://web.stanford.edu/ yyye/yyye/Gset/
29. Wu Q, Hao J.-K (2012) A Memetic Approach for the Max-cut Problem, In International Conference on Parallel Problem Solving from Nature. Springer, pp. 297–306
30. Wang Y, Lü Z, Glover F, Hao J-K (2013) Probabilistic grasp-tabu search algorithms for the ubqp problem. Comput Oper Res 40(12):3100–3107
31. Kochenberger GA, Hao J-K, Lü Z, Wang H, Glover F (2013) Solving large scale max cut problems via tabu search. J Heuristics 19(4):565–571
32. Benlic U, Hao J-K (2013) Breakout local search for the max-cutproblem. Eng Appl Artif Intell 26(3):1162–1173
33. Intel Xeon Gold 6242 Processor, https://ark.intel.com/content/www/us/en/ark/products/192440/intel-xeon-gold-6242-processor-22m-cache-2-80-ghz.html
34. Nvlink, Nvswitch (2021) The Building Blocks of Advanced Multi-GPU Communication, https://www.nvidia.com/en-us/data-center/nvlink/