Toward Quantum Refactoring:

Self-Organizing Parallel Resource Mapping with Computation Matrix Transform

Liwen Shih, Ph.D. and Hon Lum
Computer Engineering
University of Houston – Clear Lake
Houston, Texas, USA
shih@uhcl.edu

***Abstract*—As Quantum Annealing Computers (QACs) like D-Wave 2000Q Adiabatic Quantum Systems emerge, we aim to investigate the potential synergy between QAC and HPC as we push toward exascale supercomputing, where manual parallel programming for millions of processor cores will become inhibitive. Quantum Refactoring is proposed here only as a possible future concept (not yet implemented on QACs) to automatically tweak the code sequence more efficiently than through repeated manual pair-wise operation swaps to optimize computation speed, memory storage, hit ratio, cost, reliability, power and/or energy saving. To facilitate auto code refactoring suitable for such annealing optimization, a self-organizing matrix transform is proposed in this paper, so that QAC can be applied to auto code sequence permutation via computation matrix transform model toward optimized matching between computation and parallel processor cores. The mathematical model to achieve these goals is through the causal set properties of Matrix Model of Computation (MMC). A sequence of transformations act as the code refactoring to compact code regions as computation decomposition for parallel multi-core/multi-tread execution. Besides the improved software/hardware matching, the self-organizing matrix approach also serve as a novel paradigm for auto parallel programming, as well as a systematic tool for formal design modeling.**

Keywords—Automatic Software-Hardware Resource Mapping; Self-Organized Code Refactoring; Permutation; Causal Matrix Model of Computation; Data-Flow Discovery; Quantum Annealing Optimization

I. PARALLEL MAPPING OPTIMIZATION

The rapid emergence of complex computation, multi-core processing elements (PEs), many-core accelerators and cluster computer architecture, demands better matching between software computation and parallel system architecture. Despite being introduced since late 1950s, current parallel processing technology still falls short to efficiently utilize all the processing power of the available multi-core PEs. Task scheduling is an NP-complete problem, especially in the matching between computation and machine architectures. At program execution, resource allocation is the major issue in task scheduling. Parallel mapping is to partition the computation, and then allocate the PE resource for the divided workload partitions. Parallel software development is another major hurdle, since human mind is accustomed to sequential thinking on logical or analytical matters, but not readily parallel. The quest for better software-hardware matching is constrained on the data-flow dependency within one partition, and among all partitions of computation. The more deterministic the execution order of computation operations is, the easier the computation can be partitioned and matched to available parallel resource [1] [2].

As Quantum Annealing Computers (QACs) like D--Wave Adiabatic Quantum Systems emerge, we aim to investigate the missing link between QAC and HPC as we push toward exascale supercomputing, where manual parallel programming for millions of processor cores will become inhibitive. Quantum Refactoring is proposed here only as a possible future concept (not yet implemented on QACs) to automatically tweak the code sequence more efficiently than through repeated manual pair-wise operation swaps, either by human programmer or optimizing compilers, to improve computation speed, memory storage, hit ratio, cost, reliability, power and/or energy saving. QAC design promises to take advantage of the quantum physics superposition property of handling multiple states simultaneously, as well as avoiding being trapped at a local minimum via quantum tunneling to cut though to global minimum. To facilitate auto code refactoring suitable for such annealing optimization, a self-organizing matrix transform model (MMC) [6] toward optimized matching between computation and parallel processor cores is proposed in this paper, so that QAC can be applied to self-organize code matrix transform. Matrix Modeling of Computation is adopted here to achieve systematic software-hardware match through computation matric transformation steps.

MMC is a Turing-equivalent virtual machine, and a universal container for source code which can represent any finitely realizable physical system. The early MMC model developed is called *imperative MMC* (iMMC). iMMC consists of two matrices – {***C***,***Q***}; ***C*** is the matrix of services which holds the computation operations, and ***Q*** is the matrix of sequences which holds the sequence order of the computation operations; therefore the computation operations in matrix of services ***C may be out of order***. A later model of MMC developed is called *canonical MMC* (cMMC); cMMC only consists of a matrix of services. Due to the *self-organizing* property of cMMC, the matrix of sequence, ***Q,*** is no longer needed in cMMC [6]. The matching algorithm introduced in this paper is leveraging the self-organizing property of cMMC to refactor the sequence order of computation operations according to their data-flow dependency. This auto code refactoring can be time-consuming, thus emerging technologies like quantum annealing can be employed in the

future for better self-optimizing code performance tweaking efficiency.

Both total order and partial order sets can be derived from cMMC easily; cMMC can be refactored according to different optimization goals such as to optimize power consumption, memory usage, execution time, or cost. Since cMMC possesses the intrinsic parallelism property, structurally easy to be interpreted, and can be defined mathematically, cMMC can also be used as a tool for parallel programming and design model representation.

This article first presents the examples of MMC in Section I; Section II to Section VI discuss the properties of cMMC; Section II explains canonicalization, which is the core of cMMC; Section III explains refactoring, which is a key for optimizing the matrix; Section IV and Section V present the *partition of sequence* (PoS) and *block system* respectively, which are the features for partitioning the computation; Sections VII and VIII detail optimized matching regarding how to apply all the properties and features of cMMC in software-hardware matching with the possible future help of Quantum Annealing. It is then followed by conclusion and references.

II. MATRIX MODEL OF COMPUTATION

iMMC

Before proceeding with the demonstration of iMMC, a sequence of operations are first given as an example computation used to demonstrate the representation and implementation of MMC, shown as the nine-operation routine below:

1. a = v1 + v2
2. b = a * v3
3. c = b – v4
4. d = a / c
5. e = b * d
6. f = v1 / v4
7. g = v3 + v4
8. h = f + g
9. i = g – h

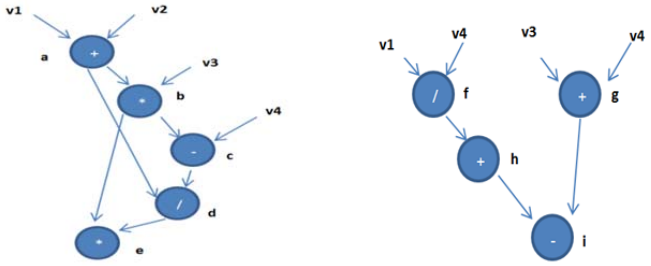

Fig. 1. Data-flow graph discovery.

Fig. 1 depicts the data-flow dependence graph hidden in the example code sequence routine. There are two disjoint data-flow sub-graphs of the routine to be discovered, which can be executed independently. In this case, both do share some same input variables ***v***1, ***v***3, ***v***4 (***v***2 is only used by data-flow graph on the left). But dependency only matters upon temporary variables (***a*** to ***i***). Let's refer to data-flow sub-graph on the left to be DFG_L, and the right to be DFG_R. The corresponding matrix of service, ***C***, of iMMC is shown in Fig. 2.

The first column of iMMC, ***OP***, shows the operators of the computation steps (+, - , *, /), where ***Operations*** in the second column are the computation operations. The labels above each column of the matrix elements indicate the *temporary* variable names. The notation of capital ***C*** (indexed by the row number) represents the computed output, or ***co-domain***, while ***A*** (also indexed by the row number) represents ***argument*** or input. Input variables are always shown on the right pane of the matrix for referencing purpose only, since input variables are regardless in MMC operations. This also means that the MMC theorems consider/apply only upon the middle square matrix (excluding the right input variables portion of the matrix).

OP	Operations	a	h	i	b	f	e	c	g	d	v1	v2	v3	v4
+	a = v1 + v2	C1	-	-	-	-	-	-	-	-	A1	A1	-	-
+	h = f + g	-	-	-	-	-	A8	A8	C8	-	-	A8	-	-
-	i = g - h	-	-	-	-	-	-	A9	A9	C9	-	-	-	-
*	b = a * v3	A2	C2	-	-	-	-	-	-	-	-	-	A2	-
/	f = v1 / v4	-	-	-	-	-	C6	-	-	-	A6	-	-	A6
*	e = b * d	-	A5	-	A5	C5	-	-	-	-	-	-	-	-
-	c = b - v4	-	A3	C3	-	-	-	-	-	-	-	-	-	A3
+	g = v3 + v4	-	-	-	-	-	-	C7	-	-	-	-	A7	A7
/	d = a / c	A4	-	A4	C4	-	-	-	-	-	-	-	-	-

Fig. 2. iMMC, Matrix of Service – C, where the sequence order of computation operations is not a concern.

OP	Operations	a	b	c	d	e	f	g	h	i	v1	v2	v3	v4
+	a = v1 + v2	C1	-	-	-	-	-	-	-	-	A1	A1	-	-
*	b = a * v3	A2	C2	-	-	-	-	-	-	-	-	-	A2	-
-	c = b - v4	-	A3	C3	-	-	-	-	-	-	-	-	-	A3
/	d = a / c	A4	-	A4	C4	-	-	-	-	-	-	-	-	-
*	e = b * d	-	A5	-	A5	C5	-	-	-	-	-	-	-	-
/	f = v1 / v4	-	-	-	-	-	C6	-	-	-	A6	-	-	A6
+	g = v3 + v4	-	-	-	-	-	-	C7	-	-	-	-	A7	A7
+	h = f + g	-	-	-	-	-	A8	A8	C8	-	-	A8	-	-
-	i = g - h	-	-	-	-	-	-	A9	A9	C9	-	-	-	-

Fig. 3. Example 1 of cMMC.

We only define the matrix of sequence, ***Q***, of iMMC, but without real example here, since the creator of MMC, Dr. Sergio Pissanetzky, found that cMMC is a superior matrix model over iMMC, and the development work has since proceeded only on cMMC. However, based on the definitions, matrix of sequence defines the order of services execution, this matrix of sequence has two or more columns where column ***P*** defines the previous sequence, and column ***F*** defines the next sequence, additional columns can be defined and used for control variables. The sequence decisions are based only on the state of the system, following the typical Turing machine model [6].

cMMC

Fig. 3 is an example of cMMC using the same sequence of computation operations.

III. MMC CANONICALIZATION

The self-organizing behavior of cMMC allows cMMC to be constructed with just a single matrix which has the built-in

information of sequence order. This feature gets rid of the need of a separate matrix of sequence, ***Q***, as opposed to iMMC. Therefore, cMMC is a *superset* of iMMC feature-wise. For an MMC to be qualified as cMMC, the matrix has to be in canonical form. As shown earlier, there are two notations in MMC, uppercase character ***C*** represents the computed output and ***A*** represents the argument or the input of a computation operation. A cMMC may consist of one or more computation operations where each operation fills up a row with ***A*** and ***C*** at the appropriate columns. MMC is in *canonical form* if and only if *all C's are lined up at the diagonal, and all **A**'s are located below C diagonal*. The sequence order of such computation operations will then be revealed automatically from top row descending downward bottom row of the cMMC.

The *canonicalization* is the process of forming a cMMC. cMMC possesses intrinsic parallelism, it will guarantee that no violation of data-flow dependency between the computation operations as soon as a cMMC is formed. Total order set, partial order set, partitioning, and mapping can then be determined and perform directly. Furthermore, this is just the least a cMMC is capable of performing; cMMC can be further *refactored* (rearranged) to improve the efficiency of the computation partitioning and resource matching. The refactoring of the row sequence is straight forward and the data-flow dependency can be verified spontaneously due to the self-organizing property of cMMC causal set.

IV. Refactoring of cMMC

The permutation of the row sequence of a cMMC is known as refactoring. Refactoring is done in a way of diagonally swapping the adjacent rows (and their corresponding columns) one pair at a time, in other words, only two adjacent rows can be swapped if and only if the data-flow dependency is not violated, referred to as *legal swaps*. Legal swaps must satisfy the canonical form of cMMC, where ***C***'s are only on the diagonal, and all ***A***'s located below the ***C*** diagonal. Data-flow is defined by the imaginary vertical column-association and horizontal row-association lines between each ***C*** and ***A***, called the *flux line*. Fig. 4 shows the flux lines and their data-flow dependencies. Data-flow propagates from top to bottom and from left to right of the matrix. It first starts with the vertical flux line from ***C*** at row 1 column 1 to ***A*** below it (***C*** as input into next ***A***), if any, then it proceeds to the horizontal flux line from ***A*** to ***C*** on the right side of ***A*** (input ***A*** into ***C***). Let's take Fig. 4 as an example, where the output ***C*** of operation ***x*** is going to be the input ***A*** of operation ***z***. This means operation ***x*** precedes operation ***z*** and so forth.

Quantum annealing can be applied here to simultaneously evaluate all legal permutations to automatically tweak the code sequence more efficiently than through repeated manual operation swaps to optimize computation speed, memory storage, cost, reliability, and/or power saving. Quantum Refactoring produces the minimum cost value of optimal permutation from arriving at the lowest energy state of the quantum annealing computer.

Here a new term describing the energy state of the code-refactoring is introduced, *functional cost*; functional cost is a quantity derived via a cost function corresponding to each permutation state of a particular cMMC. [4] Functional cost can be determined by the *sum of displacement between each **C** and **A***, for example, functional cost is 4 for the cMMC in Fig. 4. Recall that cMMC itself only guarantees the non-violation of data-flow dependency; somehow it doesn't guarantee the effectiveness of the sequence order. By having the notion of functional cost, now a cMMC can be further refactored to minimize the functional cost. In Fig. 4, notice that ***y*** is an independent operation, where it has no data-flow relationship (no inter-row flux line) with either ***x*** or ***z*** operations. Swapping rows ***x*** and ***y*** is therefore a legal permutation. Fig. 5 shows the cMMC after the swapping of operations ***x*** and ***y***. The new functional cost is now minimized from 4 to 2.

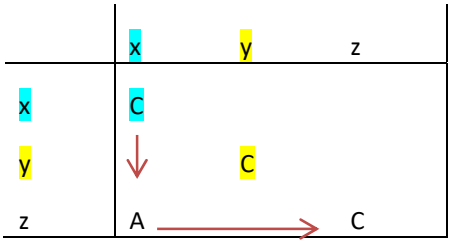

Fig. 4. Example of Flux Lines before Functional Cost Minimization, cost =4.

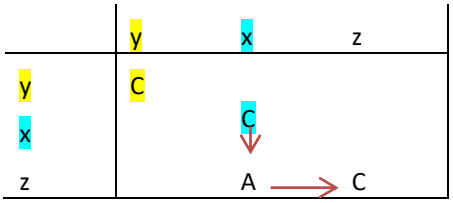

Fig. 5. Example of Flux Lines and after Functional Cost Minimization, cost = 2.

Fig. 6 shows the flux line of example 1. As seen in Fig. 6, the arrows in second block show the data-flow dependency. ***C***6 (f) will be the input ***A***8 below vertically, and then ***A***8 will become the input of ***C***8 (h) on the right hand side horizontally. Same flow applied to ***C***7 (g) and the rest of the cMMC. Therefore, a partial order set can be defined based on the data-flow flux line in cMMC. The total order data-flow graph can also be derived back from cMMC partial order set using the same principle.

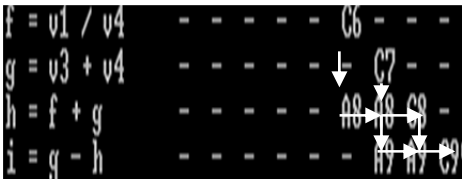

Fig. 6. Flux Lines of Example 1 of cMMC.

Based on the above intrinsic features, cMMC is chosen to serve as an analytical tool as well as a scheduling tool for mapping from computation to parallel processors. Our current scope is mainly focused on the parallel processor allocation/assignment of computation, rather than the scheduling priority.

V. PARTITION OF SEQUENCE (POS)

Based on the notion of flux line, now another new term can be introduced, called *partition of sequence* (PoS). PoS is a subset of cMMC which represents a subset of computation operations that is self-contained in term of data-flow dependency. Each PoS has no data-flow relationship with any other PoS in a particular cMMC; there is no flux line across different PoSs (no inter-PoS flux line). Therefore, each PoS can be matched to a parallel processor/core and can be executed simultaneously with other PoSs. Fig. 7 shows two independent PoSs of example 1.

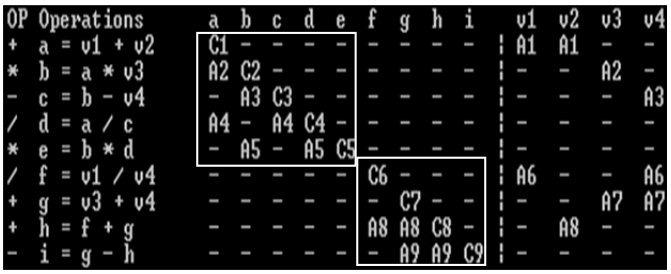

Fig. 7. PoS of Example 1 of cMMC.

Recall that the two PoSs reveal exactly the two data-flow sub-graphs DFG_L and DFG_R previously defined in the data-flow graph automatically.

VI. BLOCK SYSTEM

Block system is based upon group theory in mathematics. Block system is the subsets of PoS. Take the cMMC in Fig. 8 as an example, in order to simplify the problem, there is only one PoS in this example cMMC, in other words, the whole cMMC is one PoS, i.e., both are mathematically equivalent in this case. Recall that each PoS is self-contained and can be matched to a separate processing element (PE) core. Block system is a set of blocks within a PoS that can be further divided and a higher degree of parallelism in fewer operation steps can thus be revealed.

	a	b	c	d	e
a	C				
b		C			
c	A	A	C		
d	A			C	
e	A				C

Fig. 8. cMMC with only one PoS [7].

Fig. 9 shows the PoS with the regions of block system, where a block is a square area bounded by the dotted lines which *consists of at least one **C** element*. Thus there are three blocks in Fig. 9. Instead of having all the operations in the whole PoS allocated to a single PE as in Fig. 8, block system reveals potential two levels of execution. The first level of execution consists of operation ***a*** and ***b***, each operation can be allocated to a different PE, thus two parallel PE can be utilized simultaneously in this step of execution. The second step of execution consists of operations ***d***, ***e***, and ***c***, each operation can be allocated to a different PE, and therefore three parallel PEs can be utilized simultaneously in this level of execution. In this way, 5 operations performed in 2 level steps reveal a possible speed up of 5/2 = 2.5 ignoring inter-PE latency.

	a	d	e	b	c
a	C				
d	A	C			
e	A		C		
b				C	
c	A			A	C

Fig. 9. Regions of Block System in PoS [7].

Determining Regions of Block System

Let's assume Example 2 with just a set of 7 elements {a b c d e f g} with 6 precedence relations. Assume there are 4 minimum functional cost legal permutations:

(a b c d e f g)
(a b c d e g f)
(a e f g b c d)
(a e g f b c d)

Let's not worry about the partial order of precedence relations. This example is used to demonstrate the method of determining the regions of block system by hand.

First pick any one permutation as a starting reference. Let's use the first permutation (a b c d e f g) as starting reference for all calculations. The first element is ***a***. Candidate permutations with 2 or more elements starting with ***a*** must be formed; the goal is to form the candidates with the most number of elements being one less than the entire number of elements in the set. In this case, candidates with 6 elements will be the best for this 7-element set example. There are 6 candidates with 2 or more elements starting with ***a***: ***ab***, ***abc***, ***abcd***, ***abcde***, ***abcdef***, and ***abcdefg***. Unfortunately none of them stays together (*stay together as of combination instead of permutation*) in all 6 minimum functional cost permutations except ***abcdefg***, which is one trivial block region and of no interest. Therefore, ***a*** alone is a candidate to be a region of block system.

However, before declaring ***a*** as a block, one must verify that all other minimum displacement value (MDV) permutations in the same column are also valid regions of block system. They are, indeed, because they are all ***a***. So now declare ***a*** as a block region, and draw a vertical line from top to bottom separating ***a***:

(a|b c d e f g)
(a|b c d e g f)
(a|e f g b c d)
(a|e g f b c d)

At this point, use again the top permutation as reference, the next first element is ***b***. Form candidates **bc**, **bcd**, **bcde**, **bcdef**, and **bcdefg**, one at a time. ***bc*** is found to stay together in all permutations; but in the same column there are ***ef*** and ***eg***, and neither one of them is a block region because they do not stay together across all permutations. Therefore, ***bc*** is not a block region. The next candidate is ***bcd***, ***bcd*** does stay together; before declaring ***bcd*** a block region, check the other

subsets in the same column-- ***efg*** and ***egf***. They do stay together, so ***bcd*** can be declared as a block region, and draw another vertical line:

(a|b c d|e f g)
(a|b c d|e g f)
(a|e f g|b c d)
(a|e g f|b c d)

At this point, the next starting element is ***e***, forming candidates ***ef*** and ***efg***, one at a time. ***ef*** is found not staying together across all permutations, but ***efg*** does; therefore the last three element is a valid region of block system. As a result, the regions of block system for this example are {a}, {b c d}, and {e f g}.

VII. Optimized Matching

Based on all the example properties and features discussed above, more rules and strategies can be formulated to refactor and partition the cMMC to generate a matching between computation and parallel Processor Elements (PEs), which will achieve any particular desired optimization goal in term of power consumption, memory usage, computing speed, and cost. For example, one of such strategies in optimizing execution time and throughput is to determine the size of PoS in Fig. 7, and then map the PoS with the most computation operations (largest size of PoS) to the fastest PE and so forth. Memory usage can be minimized without even needing any additional optimization strategies, since a cMMC with a minimum functional cost will guarantee that the retention of a particular memory location is kept at the minimum duration. That is, memory will be allocated, accessed and released efficiently within the shortest time interval, since the computation operations execute at the strictest sequence order within a compact sequence of operations. Same principle can be applied to lower the power consumption, cost, and increase the performance.

Next section of this paper will discuss the details of parallel partition and mapping optimization. One of the goals in future work is to perform the dynamic matching with consideration of the underlying processing elements topology.

VIII. Parallel Partition and Mapping Optimization

This section discusses how to map the computation to parallel processors utilizing cMMC. Various suitable examples will be introduced under different subsections here to explain and to help readers to understand the process details.

Stages of Mapping Computation to Processors via Matrix Transform

Basically there are four stages of the process from taking in a set of computation to the final optimized mapping of the computation to parallel processors.

A. Stage 1: Matrix Modeling

Computation has to be first gone through a series of process to model the computation (software code) into cMMC format before the computation can be refactored in cMMC. This series of process is done in Stage 1 and it consists of three steps: (i) Transformation MC, (ii) Transformation SV, and (iii) Transformation AS [7]. cMMC produced by these three transformation steps will need to meet certain conditions and requirements in order for a parallel program to work logically correct. Stage 1 is not part of the scope of this paper. For more information, please refer to paper [7].

B. Stage 2: Refactoring of the Matrix

Refactoring has already been explained in detailed in Section IV of this paper. Refactor is done by rearranging the row-column sequence of cMMC symmetrically around matrix diagonal. cMMC will converge to have the minimum functional cost by the end of this refactoring stage.

C. Stage 3: Partitioning of Computation

This stage is to partition or group the computation so that each partition can then be mapped to different PEs. Note that it is the computation being partitioned, but not necessary the matrix itself. Three different partition strategies will be discussed in this paper.

1) Partition by PoS

PoS has been discussed in detailed in section V. Each PoS is data-flow independent from one another, i.e. no data-flow flux going across different PoS. In this mapping strategy, each PoS will be mapped to a different PE, and there will be no inter-PE communication needed. This type of partition method is straight forward and is very easy to implement. PoS can be easily identified by users from the cMMC. As depicted in Fig. 7, each PoS is shaped as a small lower triangular. There will be at least one PoS in cMMC (the entire cMMC is a PoS, thus the entire cMMC shapes as a lower triangular). PoS can be easily determined as well programming wise. The algorithm first picks the first column as the reference column. It checks if there is any vertical ***C*** to ***A*** data-flow at one row to the row below it (of the same reference column). If no vertical ***C*** to ***A*** data-flow found, the program will proceed to the subsequence row (of the same column) till it reaches the end of the row; then it will pick the next column as reference column. If there is a vertical ***C*** to ***A*** data-flow found, the program will use the lower row number (where the ***A*** is found) as the new reference column number and repeat the search process. The row where the last vertical ***C*** to ***A*** data-flow encountered will be marked as the end of that particular PoS (the first row is always the starting of the first PoS). The next row after the end of a PoS, if any, will be once again marked as the beginning row of the next matrix block. The program repeats the same process until the end of column is reached. Each PoS will then be considered as a partition and will be mapped to a PE according the desired parallel mapping optimization in the next mapping stage.

2) Partition by Regions of Block System

The second partition strategy is to determine the regions of block system. Regions of block system have been explained in detailed in Section VI. Regions of block system can further reveal the parallelism of the routine. Define a system $\sum_1 = 10\{ad, be, cf, g, eh, fi, gj, hj, ij\}$. This example is used in Sergio Pissanetzky's paper "Emergence and self-organization in partially ordered set" section 4.2 [7]. There are 10 elements in this system, {a, b, c, d, e, f, g, h, i, j} and 9 precedence relations in its partial order:

$$\begin{aligned} a &< d \\ b &< e \\ c &< f \\ d &< g \\ e &< h \\ f &< i \\ g &< j \\ h &< j \\ i &< j \end{aligned} \quad (1)$$

System $\sum_1$ has 1680 legal permutations. A permutation can be interpreted as a state of a cMMC at a particular time. A permutation is legal when the sequence order (row order) of the system does not violate any precedence relations in its partially order set. For system $\sum_1$, it has 1680 legal permutations with functional cost in the range 18 (6 permutations) to 24 (540 permutations). Fig. 10 depicts system $\sum_1$ before refactoring with maximum functional cost of 24 and Fig. 11 depicts system $\sum_1$ after refactoring with minimum functional cost of 18.

```
                              1  2  3  4  5  6  7  8  9  10
OP        Operations a  b  c  d  e  f  g  h  i  j
          a          C1 -  -  -  -  -  -  -  -  -
          b          -  C2 -  -  -  -  -  -  -  -
          c          -  -  C3 -  -  -  -  -  -  -
=         d = a      A4 -  -  C4 -  -  -  -  -  -
=         e = b      -  A5 -  -  C5 -  -  -  -  -
=         f = c      -  -  A6 -  -  C6 -  -  -  -
=         g = d                -  -  -  A7 -  -  C7 -  -  -
=         h = e                -  -  -  -  A8 -  -  C8 -  -
=         i = f                -  -  -  -  -  A9 -  -  C9 -
+         j = g+h+i  -  -  -  -  -  - A10A10A10 C10
```

Fig. 10. System ∑1 before Refactoring with Maximum Functional Cost of 24.

A group of 6 permutations with minimum functional cost is listed below:

$$\begin{aligned} &(a\ d\ g\ b\ e\ h\ c\ f\ i\ j) \\ &(a\ d\ g\ c\ f\ i\ b\ e\ h\ j) \\ &(b\ e\ h\ a\ d\ g\ c\ f\ i\ j) \\ &(b\ e\ h\ c\ f\ i\ a\ d\ g\ j) \\ &(c\ f\ i\ a\ d\ g\ b\ e\ h\ j) \\ &(c\ f\ i\ b\ e\ h\ a\ d\ g\ j) \end{aligned} \quad (2)$$

```
OP        Operations c  f  i  b  e  h  a  d  g  j
          c          C3 -  -  -  -  -  -  -  -  -
=         f = c      A6 C6 -  -  -  -  -  -  -  -
=         i = f                -  A9 C9 -  -  -  -  -  -
          b          -  -  -  C2 -  -  -  -  -  -
=         e = b      -  -  -  A5 C5 -  -  -  -  -
=         h = e                -  -  -  -  A8 C8 -  -  -  -
          a          -  -  -  -  -  -  C1 -  -  -
=         d = a      -  -  -  -  -  -  -  A4 C4 -  -
=         g = d                -  -  -  -  -  -  -  A7 C7 -
+         j = g+h+i  -  -  A10-  -  A10-  - A10 C10
```

Fig. 11. System ∑1 after Refactoring with Minimum Functional Cost of 18.

The block regions for this group are {a d g}, {b e h}, {c f i}, and {j}, its partial order induced by Eq. 1 is:

$$\begin{aligned} \{a\ d\} &< \{g\} \\ \{g\} &< \{j\} \\ \{b\ e\} &< \{h\} \\ \{h\} &< \{j\} \end{aligned}$$

$$\begin{aligned} \{c\ f\} &< \{i\} \\ \{i\} &< \{j\} \end{aligned} \quad (3)$$

Thus the partitions by regions of block system will be {a d g}, {b e h}, {c f i}, and {j}.

3) Partition by Interleaving PoS

For pipeline system, routine in most compact form with minimum functional cost may not yield the faster execution time since there may be pipeline stalls between operations. In this case, partition by interleaving mutual exclusive PoS will alleviate the pipeline stalling problem. Let's consider Example 1 again as in Fig. 12.

```
OP Operations     a  b  c  d  e  f  g  h  i   v1 v2 v3 v4
+  a = v1 + v2    C1 -  -  -  -  -  -  -  -  | A1 A1 -  -
*  b = a * v3     A2 C2 -  -  -  -  -  -  -  | -  -  A2 -
-  c = b - v4     -  A3 C3 -  -  -  -  -  -  | -  -  -  A3
/  d = a / c     A4 -  A4 C4 -  -  -  -  -  | -  -  -  -
*  e = b * d     -  A5 -  A5 C5 -  -  -  -  | -  -  -  -
/  f = v1 / v4    -  -  -  -  -  C6 -  -  -  | A6 -  -  A6
+  g = v3 + v4    -  -  -  -  -  -  C7 -  -  | -  -  A7 A7
+  h = f + g      -  -  -  -  -  A8 A8 C8 -  | -  A8 -  -
-  i = g - h      -  -  -  -  -  -  A9 A9 C9 | -  -  -  -
```

Fig. 12. Example 1.

Let's assume input variable ***v*** is obtained via memory access. For every operation that requires variable ***v*** as its input variable(s) requires load instruction for memory access, and each load instruction imposes a pipeline stall if the output is needed in its immediate consecutive operation. Thus if operations in Example 1 is subject under pipeline processing, there will be a total of 5 pipeline stalls required for operation ***a***, ***b***, ***c***, ***f***, and ***g***. If the operations are partitioned in a different sequence such as (a, f, b, g, c, h, d, e, i) or (a, f, b, g, c, h, d, i, e), all pipeline stalls will be eliminated. Since ***f*** is not depending on ***a***, apparently ***f*** is from a different matrix block and each matrix block has no data-flow dependency to other matrix blocks. Thus ***f*** can be loaded into the pipeline stage right behind current pipeline stage of ***a***. The next operation in ***a*** matrix block, ***b***, is then being scheduled to run after ***f***. Note here that operations from different matrix blocks take turn to run alternatively, referred to as the partition by interleaving matrix block.

D. Stage 4: Mapping of Partitions to Processing Elements

This is the last stage of the mapping of computation to parallel processors. The mapping strategies are basically the same as partitioning strategies, i.e. mapping by PoS, regions of block system, and/or interleaving matrix blocks. Basically each determined partition can be mapped directly to each PE. Thus the mapping is direct and simple once the partitions have been determined in stage 3.

IX. CONCLUSION

Auto code refactoring via computation matrix transform is proposed and demonstrated, where manually tweaking or tedious sequential operation pairwise swaps to achieve optimized code sequence permutation can be make efficient by the emerging quantum annealing computers to find optimal legal permutations that minimize functional cost, similar to Traveling Salesman Problem (TSP). Computation partitioning

and resource allocation are the major problems in parallel processing. Even with an efficient scheduling algorithm in place, there are still major challenges for programmers to develop the parallel applications. An algorithm that could solve the partitioning and mapping problems efficiently and also capable to serve as a tool to assist engineers to develop their parallel applications is therefore significant. In a related project, the manual instruction sequence tweaking was shown to increase more than 24% computation speed due to improved memory hit rate and access efficiency [8]. We are working toward even more improvement via Quantum Optimized auto code refactoring.

There are four stages in the whole parallel mapping process. Stage 1 is to model the computation source code into cMMC. Stage 1 includes three steps: transformation MC, transformation SV, and transformation AS. These steps are crucial in forming a cMMC that the computation can then be partitioned and mapped to parallel processors. Stage 2 is to refactor the cMMC. This is the major stage of transforming the cMMC to meet the desired parallel optimization goals. Stage 3 is to partition the computation. There are three different partitioning strategies: (1) partition by PoS, (2) partition by regions of block system, and (3) partition by interleaving PoS. Stage 4 is to map the partition to processing elements. There are three mapping strategies similar to partitioning strategies: (1) mapping by PoS, (2) mapping by regions of block system, and (3) mapping by interleaving PoS. A hybrid strategy in both partitioning and mapping may also be used but is not discussed in this paper.

A software tool has been developed, called Matrix Parallel Computation Matcher (MPCM). MPCM is written in C programming language and is based on the causal set property of cMMC. MPCM is capable of taking in a set of computation operations, refactor the operation sequence, and partition the operation sequence according to the particular desired matching goal of the users. At current stage, MPCM is able to parse an input text file with computation operation sequence previously typed in by users, while future work is to develop a program parser for MPCM to parse the programming code directly as the input of computation operations. Matrix Model also provides a formal representation and can be described, defined, and manipulated using formal mathematical notations and logic. This goal can be achieved with the help of workable input interface mechanism as just mentioned; current work has already shown a good result by taking in some pre-optimized computation from other scheduling algorithm, represented it in matrix model, and incorporated more varieties of optimized matching. Furthermore, MPCM can transform the output result back into the original model, because that matrix model is fairly robust in mathematical properties. Matrix computation model is also readily programmable to take advantage of all the underlying advantages of many readily available mathematical matrix techniques.

ACKNOWLEDGMENT

The authors sincerely appreciate the collaboration with our UHCL visiting research scholar Dr. Sergio Pissanetzky, who devotes his research for intelligence emergence understanding with Causal Matrix Modeling of Computation (MMC).

REFERENCES

[1] Shih, Liwen, “Adaptive latency-aware parallel resource mapping: task graph scheduling onto heterogeneous network topology,” XSEDE 2013: 52 ACM 2013 Article

[2] Cui, Yifeng, Efecan Poyraz, Jun Zhou, Scott Callaghan, Phillip Maechling, Thomas Jordan, Liwen Shih, Po Chen, “Accelerating CyberShake Calculations on the. XE6/XK7 Platform of Blue Waters,” XScale 2013 Extreme Scaling Workshop, 2013.

[3] Sergio Pissanetzky: A new type of Structured Artificial Neural Networks based on the Matrix Model of Computation. May 2008.

[4] Sergio Pissanetzky: A New Universal Model of Computation and its Contribution. World Academy of Science, Engineering and Technology 51, 2009.

[5] Sergio Pissanetzky: Applications of the Matrix Model of Computation. World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI’08), 2008.

[6] Sergio Pissanetzky: The Matrix Model of Computation. World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI’08), 2008.

[7] Sergio Pissanetzky: Emergence and Self-organization in Partially Ordered Sets, Complexity, Volume 17, Issue 2, pages 19-38, 2011.

[8] Hakduran Koc and Mehmet Ucar, M.S. Thesis, "Execution Phase Partitioning for Data Intensive Applications”, 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, January 2017