A Permutational Boltzmann Machine with Parallel Tempering for Solving Combinatorial Optimization Problems

Mohammad Bagherbeik[1(✉)], Parastoo Ashtari[1], Seyed Farzad Mousavi[1], Kouichi Kanda[2], Hirotaka Tamura[2], and Ali Sheikholeslami[1]

[1] University of Toronto, Toronto, ON M5S2E8, Canada
tabrizimo73@gmail.com
[2] Fujitsu Laboratories Limited, Kawasaki, Kanagawa 211-8588, Japan

Abstract. Boltzmann Machines are recurrent neural networks that have been used extensively in combinatorial optimization due to their simplicity and ease of parallelization. This paper introduces the Permutational Boltzmann Machine, a neural network capable of solving permutation optimization problems. We implement this network in combination with a Parallel Tempering algorithm with varying degrees of parallelism ranging from a single-thread variant to a multi-threaded system using a 64-core CPU with SIMD instructions. We benchmark the performance of this new system on Quadratic Assignment Problems, using some of the most difficult known instances, and show that our parallel system performs in excess of 100× faster than any known dedicated solver, including those implemented on CPU clusters, GPUs, and FPGAs.

Keywords: Parallel Boltzmann Machine · Replica exchange Monte-Carlo · Combinatorial optimization · Quadratic Assignment Problem

1 Introduction

Boltzmann Machines (BM), first proposed by Hinton in 1984 [13], are recurrent, fully connected, neural networks that store information within their symmetric edge weights. When combined with Stochastic Local Search methods such as Simulated Annealing (SA) [1] or Parallel Tempering (PT) [10], BMs can be used to perform combinatorial optimization on complex problems such as TSP [3], MaxSAT [8], and MaxCut [16]. In this paper, we present an algorithm for a Permutational Boltzmann Machine (PBM), structured to solve complex, integer based, permutation optimization problems. We combine this PBM with Parallel Tempering and propose both single-threaded and multi-threaded, software implementations of this PBM + PT system using a 64-core CPU along

The authors would like to thank Fujitsu Laboratories Ltd. and Fujitsu Consulting (Canada) Inc. for providing financial support and technical expertise on this research.

T. Bäck et al. (Eds.): PPSN 2020, LNCS 12269, pp. 317–331, 2020.
https://doi.org/10.1007/978-3-030-58112-1_22

with SIMD instructions. As a proof-of-concept, we show how to solve Quadratic Assignment Problems (QAP) [17] using a PBM and present experimental results on some of the hardest QAP instances from QAPLIB [6], Palubeckis [21], and Drezner [9]. We then show that, over the tested instances, our single-threaded and multi-threaded PBM systems can find the best-known-solutions of QAP problems in excess of 10× and 100× faster than the next best solver respectively.

The rest of this paper is organized as follows: Sect. 2 provides background on BMs and the formulation of QAP problems. Section 3 presents the structure of our Permutational Boltzmann Machine and Sect. 4 presents our single and multi-threaded implementations of a PBM + PT system on a multi-core CPU. Section 5 outlines the experiments conducted to benchmark the performance of our PBM + PT system and presents our results. Section 6 concludes this paper.

2 Background

2.1 Boltzmann Machines

BMs, as shown in Fig. 1, are made up of N neurons, $\{x_1, x_2, \ldots, x_N\}$ with binary states represented by vector $\mathbf{S} = [s_1\, s_2\, \ldots\, s_N]^\intercal \in \{0,1\}^N$. Each neuron, x_i, is connected to other neurons, x_j, via symmetric, real-valued weights, $w_{i,j} \in \mathbb{R}$ where $w_{i,j} = w_{j,i}$ and $w_{i,i} = 0$, forming a 2D matrix, $\mathbf{W} \in \mathbb{R}^{N\times N}$. Each neuron also has a bias value, b_i, which forms $\mathbf{B} \in \mathbb{R}^{N\times 1}$. The cumulative inputs to the neurons, also referred to as their *local fields*, h_i, form $\mathbf{H} \in \mathbb{R}^{N\times 1}$ and are calculated using (1).

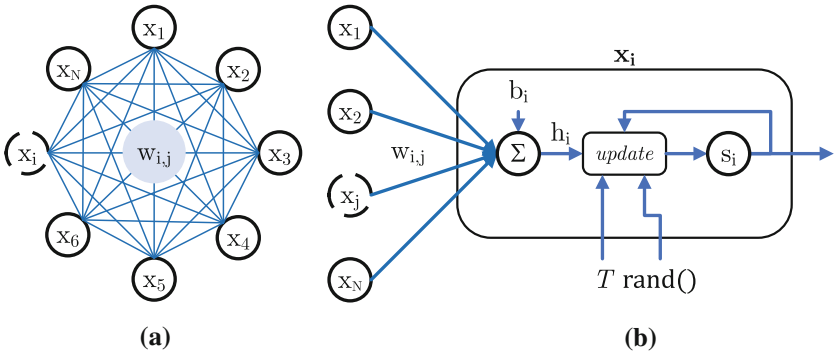


Fig. 1. Structure of a Boltzmann Machine and its neurons. (a) Top level structure of a Boltzmann Machine (b) Detailed structure of a BM neuron, where T is the system temperature and rand() is a uniform random number within [0,1]

$$h_i(\mathbf{S}) = \sum_{j=1}^{N} w_{i,j} s_j + b_i \quad , \quad \mathbf{H}(\mathbf{S}) = \mathbf{WS} + \mathbf{B} \tag{1}$$

$$E(\mathbf{S}) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{i,j}s_i s_j - \sum_{i=1}^{N} b_i s_i = -\frac{\mathbf{S}^\intercal \mathbf{W}\mathbf{S}}{2} - \mathbf{S}^\intercal \mathbf{B} \tag{2}$$

$$P(\mathbf{S}) = \frac{\exp(-E(\mathbf{S})/T)}{\sum_{\forall \mathbf{S}_j \in \{0,1\}^N} \exp(-E(\mathbf{S}_j)/T)} \tag{3}$$

Each possible state of a BM has an associated *energy* term calculated via (2). The probability of the system being in any state depends on the energy of that state as shown in (3). The lower the energy, the higher the probability that the network will be in that state. BMs create an energy landscape for a problem through the weights that connect their neurons where the state(s) with the lowest energy corresponds to a valid solution. The procedures to convert various optimization problems to the BM format are discussed in [12]. The term T in (3), known as the system *temperature*, flattens or sharpens the energy landscape when increased or decreased respectively, providing a method to maneuver the landscape when searching for the global minimum.

Generally, BMs are combined with Simulated Annealing (SA) to solve optimization problems. Using SA, at a time-step t, where the system is in state $\mathbf{S^t}$ with temperature T, the local fields $\mathbf{H}(\mathbf{S^t})$ are calculated using (1). In order to make an update to the system, we must conduct a *trial*. First, a neuron x_i is randomly chosen and the change in energy as a result of flipping its state is calculated via (4). Next, the probability of flipping the neuron's state, P_{move}, is calculated via (5) and is compared against a uniformly distributed random number in [0,1] to determine the change in the neuron's state using (6).

$$\Delta E(\mathbf{S^t}, i) = \Delta E_{s_i \rightarrow !s_i}(\mathbf{S^t}) = -[1 - 2s_i^{\mathrm{t}}]h_i(\mathbf{S^t}) \tag{4}$$

$$P_{move} = min\{1, \exp(-\Delta E/T)\} \tag{5}$$

$$\Delta s_i = \begin{cases} [1 - 2s_i^{\mathrm{t}}] & \text{if } P_{move} \geq rand() \\ 0 & otherwise \end{cases} \tag{6}$$

After the trial, the system state variables s_i, $\mathbf{H}$, and E need to be updated as shown in (7), (8), and (9) respectively, where $\mathbf{W}_{i,*}$ and $\mathbf{W}_{*,i}$ represent row and column i of $\mathbf{W}$ respectively.

$$s_i^{\mathrm{t+1}} = s_i^{\mathrm{t}} + \Delta s_i \tag{7}$$

$$\mathbf{H}(\mathbf{S^{t+1}}) = \mathbf{H}(\mathbf{S^t}) + \Delta s_i \mathbf{W}_{*,i} \tag{8}$$

$$E(\mathbf{S^{t+1}}) = E(\mathbf{S^t}) + \Delta E(\mathbf{S^t}, i) \tag{9}$$

This procedure is repeated a preset number of times, occasionally *cooling* the system by decreasing T until it goes below a certain threshold, T_{thresh}, at which

point the process is terminated and the lowest energy state observed throughout the search is returned. This state may or may not correspond to a valid or optimal solution due to the stochastic nature of the algorithm but, theoretically, if given a long enough cooling schedule, the BM + SA system will eventually converge to an optimal answer [2].

2.2 Quadratic Assignment Problems (QAP)

QAP problems, first formulated in [17], are a class of NP-Hard permutation optimization problems to which many other problems such as the Travelling Salesman Problem can be reduced. While the formulation is relatively simple, QAP remains, to this day, one of the more challenging combinatorial optimization problems. QAP problems entail the task of assigning a set of n facilities to a set of n locations while minimizing the cost of the assignment. QAP problems are comprised of $n \times n$ matrices $\mathbf{F} = (f_{i,j})$ and $\mathbf{D} = (d_{k,l})$ which describe the flows between facilities and distances between locations respectively with the diagonal elements of both matrices being 0. A third $n \times n$ matrix $\mathbf{B_P} = (b_{i,k})$, describes the costs of assigning a facility to a location. All three matrices are comprised of real-valued elements. Given these matrices, each facility must be assigned to a unique location, generating a permutation, $\boldsymbol{\phi} \in \mathbf{S_n}$, where $\mathbf{S_n}$ is the set of all permutations, such that the cost function (10) is minimized.

$$\min_{\boldsymbol{\phi} \in \mathbf{S_n}} cost(\boldsymbol{\phi}) = \min_{\boldsymbol{\phi} \in \mathbf{S_n}} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{i,j} d_{\phi_i,\phi_j} + \sum_{i=1}^{n} b_{i,\phi_i} \tag{10}$$

Generally, there are two variants of the QAP problem: symmetric (*sym*) and asymmetric (*asm*). In the symmetric case, either one or both of $\mathbf{F}$ and $\mathbf{D}$ are symmetric. If one of the matrices is asymmetric, it can be made symmetric by taking the average of an element and its complement. However, if both matrices are asymmetric, we can no longer symmetrize them in this manner. It is important to distinguish between these two cases as they are handled differently by a PBM, as will be shown in Sect. 3.

3 Permutational Boltzmann Machines (PBM)

3.1 Structure and Update Scheme

The PBM's structure is an extension of Clustered Boltzmann Machines (CBM), first proposed by De Gloria [11]. A CBM places neurons that do not have any connections between them into groups called *clusters*. Within a cluster, the states of the neurons have no effect on each other's local fields; simultaneously flipping the states of multiple neurons in the same cluster has the same effect as flipping them in sequence. In a PBM, the neurons are arranged into an $n \times n$ matrix $\mathbf{S_P} = (s_{r,c})$, where each row, r_i, and each column, c_j, forms a cluster, as shown in Fig. 2a. On each cluster, we impose an exactly-1 constraint to ensure that within each row and each column, there is exactly one neuron in the *ON* state.

In the context of a permutation problem, the row-clusters represent a 1-hot encoded integer in $[1, n]$, allowing the neuron states to be represented via the integer permutation vector, $\boldsymbol{\phi}$. The column-clusters, in turn, enforce that every integer is unique. The $n^2 \times n^2$ weight matrix is also reshaped into a 4D $(n \times n) \times (n \times n)$ matrix as shown in (11), allowing the generation of the $\mathbf{w}_{r,c}$ sub-matrices via Kronecker Products (denoted by $\otimes$) of rows and columns of $\mathbf{F}$ and $\mathbf{D}$ via (12). The $n \times n$ local field matrix $\mathbf{H_P}$ is calculated via (13).

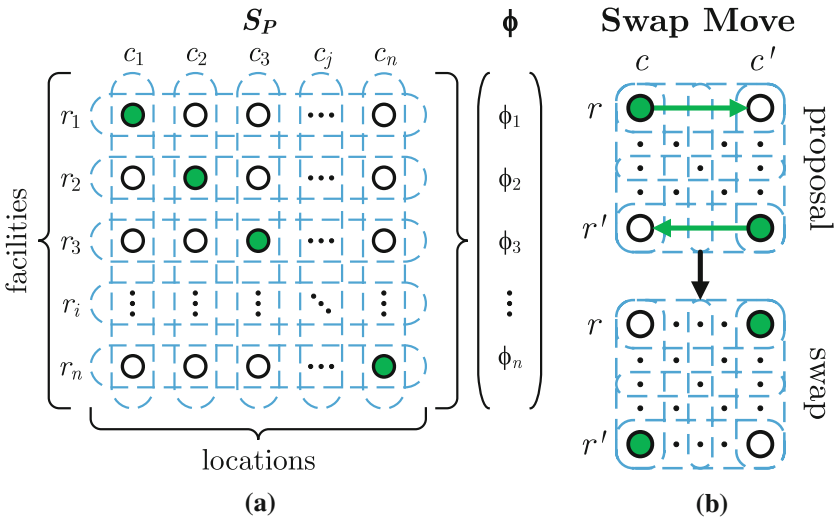


Fig. 2. Structure of a Permutational Boltzmann Machine. (a) The binary neuron state matrix $\mathbf{S_P}$ with row/column cluster structures and the permutation vector $\boldsymbol{\phi}$ (b) Structure of a permutation Swap Move

$$\mathbf{W_P} = \begin{bmatrix} \mathbf{w}_{1,1} & \mathbf{w}_{1,2} & \cdots & \mathbf{w}_{1,n} \\ \mathbf{w}_{2,1} & \mathbf{w}_{2,2} & \cdots & \mathbf{w}_{2,n} \\ \mathbf{w}_{3,1} & \mathbf{w}_{3,2} & \cdots & \mathbf{w}_{3,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{n,1} & \mathbf{w}_{n,2} & \cdots & \mathbf{w}_{n,n} \end{bmatrix} \quad , \quad \mathbf{W_P} \in \mathbb{R}^{(n\times n)\times(n\times n)} \tag{11}$$

$$\mathbf{w}_{r,c} = \begin{cases} -(\mathbf{F}_{r,*})^\intercal \otimes \mathbf{D}_{c,*} & sym \\ -(\mathbf{F}_{r,*})^\intercal \otimes \mathbf{D}_{c,*} - \mathbf{F}_{*,r} \otimes (\mathbf{D}_{*,c})^\intercal & asm \end{cases} \quad , \quad \mathbf{w}_{r,c} \in \mathbb{R}^{n\times n} \tag{12}$$

$$h_{r,c} = \sum_{r'=1}^{n} \sum_{c'=1}^{n} w_{r,c;r',c'} s_{r',c'} + b_{r,c} \quad , \quad \mathbf{H_P} = (h_{r,c}) \in \mathbb{R}^{n\times n} \tag{13}$$

We enforce the $2n$ exactly-1 constraints by not allowing moves that violate the constraints. Assuming that the system is initialized to a valid state that meets all the constraints, we propose trials via moves called *swaps* as shown in Fig. 2b.

A swap proposal involves picking two unique rows, r and r', from the neuron matrix and swapping the states of their ON neurons along columns c and c'. If accepted, this move results in 4 simultaneous bit-flips within the binary neuron matrix. The change in energy as a result of such a move is shown in (14), where the first set of local field terms correspond to the neurons being turned OFF while the second set is due to the neurons being turned ON. The two weights being subtracted are required as we have two pairs of neurons that may be connected across the clusters. The first weight is to compensate for the weight being double added by the local fields of the two neurons turning OFF. The second weight is to account for a coupling that was previously inactive between the two neurons turning ON. As shown in (15), we can directly generate the sum of these weights using $\mathbf{F}$ and $\mathbf{D}$. A trial can then be performed by substituting the ΔE value from (14) into (5) and comparing the generated move probability against a value generated by rand().

$$\Delta E(\boldsymbol{\phi}^{\mathbf{t}}, r, r') = (h^t_{r,c} + h^t_{r',c'}) - (h^t_{r,c'} + h^t_{r',c}) - (w_{r,c;r',c'} + w_{r,c';r',c}) \tag{14}$$

$$w_{r,c;r',c'} + w_{r,c';r',c} = \begin{cases} -2f_{r,r'}d_{c,c'} & sym \\ -(f_{r,r'} + f_{r',r})(d_{c,c'} + d_{c',c}) & asm \end{cases} \tag{15}$$

3.2 Updating the Local Field Matrix

When a swap proposal is accepted, the system state must be updated. Swapping the two values in $\boldsymbol{\phi}$ and adjusting the system energy is simple. However, updating the local field matrix involves a large number of calculations. Attempting to update $\mathbf{H_p}$ via (16) involves fetching four weight sub-matrices from global memory with long access delays. Interestingly, the structure of the weight matrix and the PBM itself allow these calculations to be performed efficiently while storing the majority of required data within L2 or L3 caches. For a symmetric problem, we can generate the required weights with a Kronecker Product operation on the differences between 2 rows of the $\mathbf{F}$ matrix ($\mathbf{\Delta f}$) and 2 rows of the $\mathbf{D}$ matrix ($\mathbf{\Delta d}$) using (17). For an asymmetric problem, an additional update using $\mathbf{F}^\intercal$ and $\mathbf{D}^\intercal$ is required. In this manner, the amount of memory required to store the weight data is reduced from n^4 elements for a monolithic weight matrix to $2n^2$ elements to store $\mathbf{F}$ and $\mathbf{D}$ when the problem is symmetric. For an asymmetric problem, an additional $2n^2$ elements are needed to store $\mathbf{F}^\intercal$ and $\mathbf{D}^\intercal$. Storing a transposed copy of the matrices, while doubling the required memory, provides significant speedups due to a larger number of cache hits when fetching a small number of rows.

$$\mathbf{H}^{t+1}_{\mathbf{P}} = \mathbf{H}^t_{\mathbf{P}} - (\mathbf{w}_{r,c} + \mathbf{w}_{r',c'}) + (\mathbf{w}_{r,c'} + \mathbf{w}_{r',c}) = \mathbf{H}^t_{\mathbf{P}} + \Delta\mathbf{H_P} \tag{16}$$

$$\Delta\mathbf{H_P} = \begin{cases} \mathbf{\Delta f} \otimes \mathbf{\Delta d} = (\mathbf{F}_{r,*} - \mathbf{F}_{r',*})^\intercal \otimes (\mathbf{D}_{c,*} - \mathbf{D}_{c',*}) & sym \\ \mathbf{\Delta f} \otimes \mathbf{\Delta d} + (\mathbf{F}_{*,r} - \mathbf{F}_{*,r'}) \otimes (\mathbf{D}_{*,c} - \mathbf{D}_{*,c'})^\intercal & asm \end{cases} \tag{17}$$

4 System Overview

4.1 Parallel Tempering

A major weakness of Simulated Annealing in traditional BM optimizers is that it can easily get stuck in a local minimum due to the unidirectional nature of the cooling schedule. Parallel Tempering (PT), first proposed in [23] and developed in [14], provides a means of running M cooperative copies (replicas) of the system, each at a different temperature, in order to search a larger portion of the landscape while allowing a mechanism for escaping from local minima. Replicas are generally arranged in order of increasing T from T_{min} to T_{max} in a temperature ladder. A replica, R_k, operating at temperature T_k, can stochastically exchange temperature with the replica immediately above it on the ladder, R_{k+1}, with an *Exchange Acceptance Probability* (EAP) calculated via (18). Figure 3 outlines the structure of an optimization engine using BM replicas with PT.

$$EAP = min\{1, \exp((1/T_k - 1/T_{k+1})(E_k - E_{k+1}))\} \tag{18}$$

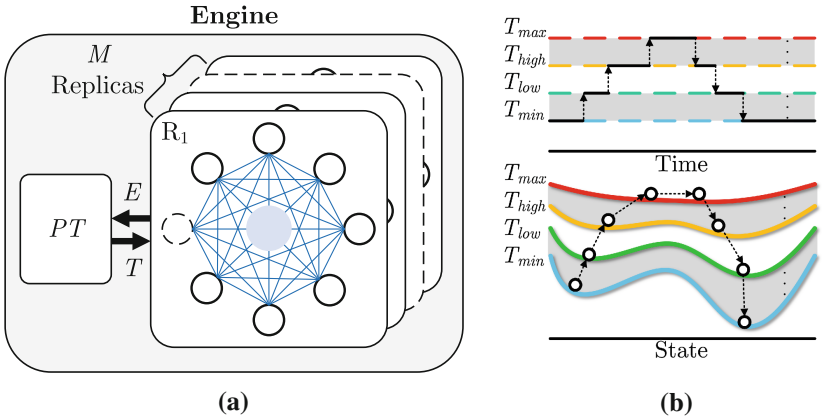


Fig. 3. Overview of a Boltzmann Machine combined with Parallel Tempering (a) Structure of BM + PT Engine (b) Example of a replica escaping from a local minimum and reaching the global optimum via climbing the PT ladder

As implied by (3) and (7), higher T replicas can move around a larger portion of the landscape whereas the moves in lower T replicas are contained to a smaller subspace of the landscape. The ability of replicas to move up or down the ladder, as shown in Fig. 3b, allows for a systematic method of escaping from local minima, making PT a better choice for utilizing parallelism than simply running M disjoint replicas in parallel using SA as proven in [10,14]. In this paper, we implement a PT algorithm based on a modified version of Dabiri's work [7]. One drawback to PT algorithms such as the BM + PT system used in [7] is that their T_{max} and T_{min} must be manually tuned for each problem instance. This

requires considerable time and effort while having dramatic effects on the efficacy of the optimization process. We partially address this issue by selecting, from each family of QAP problems, small instances whose solutions can be verified via exact algorithms, to tune a function that automatically selects these parameters for that family within our system.

4.2 Single-Threaded Program

Algorithm 1 presents our proposed PBM + PT system which can be configured for varying levels of multi-threaded operation. A single PT engine ($U = 1$) is used with $M = 32$ replicas. The algorithm starts by initializing the temperature ladder and assigning random permutations to each replica and populating their $\mathbf{H_P}$ matrices and energy values. The system then enters an optimization loop where it runs Y trials for each replica in sequence using the RUN_R() function, updating their states every time a trial is accepted by calling SWAP(). After all replicas have finished their Y trials, temperature exchanges are performed. Similar to QAP solvers such as [15,18–20,24], this process is repeated until the state corresponding to the best-known-solution (BKS) of a problem, E_{BKS}, is reached by one of the replicas, terminating the loop. The system then returns, for each replica, the minimum energy found and the corresponding state.

4.3 Multi-thread Load Balancing

For our implementation, we targeted a 64-core AMD 3990X CPU. Given the structure of a PBM combined with PT, one of the most intuitive ways to extract parallel speedups is to create a thread for each replica such that they all run on a unique core with their own dedicated L1 and L2 caches.

One issue that arises from this form of parallel execution is that replicas at higher T have higher swap acceptance rates than replicas at lower T resulting in

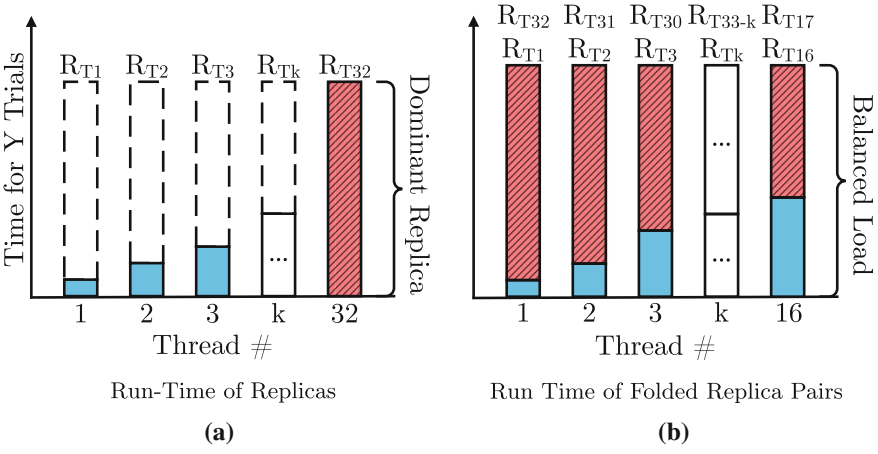


Fig. 4. Load balancing threads via replica folding

Algorithm 1. Permutational Boltzmann Machine with Parallel Tempering

Parameters: M, U, C, Y
Input: $n, E_{BKS}, asm, \mathbf{F}, \mathbf{D}, \mathbf{F}^\intercal, \mathbf{D}^\intercal, \mathbf{B_P}$
Output: $\boldsymbol{E}_{\mathbf{min}}, \boldsymbol{\phi}_{\mathbf{min}}$

Engine Variables
  $\boldsymbol{E}[U][M]$
  $\boldsymbol{E}_{\mathbf{min}}[U][M]$
  $\boldsymbol{\phi}[U][M][n]$
  $\boldsymbol{\phi}_{\mathbf{min}}[U][M][n]$
  $\mathbf{H_P}[U][M][n][n]$
  $\boldsymbol{T}[U][M]$

function RUN_R(u, m)
  for $y \leftarrow 1$ to Y **do**
    $r, r' \leftarrow$ pick 2 unique rows
    $\Delta E \leftarrow (14)$
    $P_{swap} \leftarrow \exp(-\Delta E / \boldsymbol{T}[u][m])$
    if $P_{swap} >$ rand() **then**
      SWAP(u, m, r, r')

▽ Perform Replica Swap Adjustments
function SWAP(u,m,r,r')
  $c, c' \leftarrow \boldsymbol{\phi}[u][m][r], \boldsymbol{\phi}[u][m][r']$
  $\mathbf{fr}[n:1] \leftarrow \mathbf{F}[r,:] - \mathbf{F}[r',:]$
  $\mathbf{dr}[n:1] \leftarrow \mathbf{D}[c,:] - \mathbf{D}[c',:]$
  if asm **then**
    $\mathbf{fc}[n:1] \leftarrow \mathbf{F}^\intercal[r,:] - \mathbf{F}^\intercal[r',:]$
    $\mathbf{dc}[n:1] \leftarrow \mathbf{D}^\intercal[c,:] - \mathbf{D}^\intercal[c',:]$
  for $i \leftarrow 1$ to n **do**
    $\mathbf{H_P}[u][m][i,:] \overset{+}{\leftarrow} \mathbf{fr}[i] * \mathbf{dr}$
    if asm **then**
      $\mathbf{H_P}[u][m][i,:] \overset{+}{\leftarrow} \mathbf{fc}[i] * \mathbf{dc}$
  $\boldsymbol{\phi}[u][m][r], \boldsymbol{\phi}[u][m][r'] \leftarrow c', c$
  $\boldsymbol{E}[u][m] \overset{+}{\leftarrow} \Delta E$
  if $\boldsymbol{E}[u][m] < \boldsymbol{E}_{\mathbf{min}}[u][m]$ **then**
    $\boldsymbol{E}_{\mathbf{min}}[u][m] \leftarrow \boldsymbol{E}[u][m]$
    $\boldsymbol{\phi}_{\mathbf{min}}[u][m] \leftarrow \boldsymbol{\phi}[u][m]$

▽ Exchange Replica Temperatures
function PTEXCHANGE(u)
  for $m \leftarrow 1, M-1$ **do**
    $\delta\beta \leftarrow 1/\boldsymbol{T}[u][m] - 1/\boldsymbol{T}[u][m+1]$
    $\delta E \leftarrow \boldsymbol{E}[u][m] - \boldsymbol{E}[u][m+1]$
    $EAP \leftarrow \exp(\delta\beta * \delta E)$
    if $EAP \geq$ rand() **then**
      $\boldsymbol{T}[u][m], \boldsymbol{T}[u][m+1] \leftarrow \boldsymbol{T}[u][m+1], \boldsymbol{T}[u][m]$

▷ PBM + PT Main Routine

▽ Initialize System
for $u \leftarrow 1, U$ **do**
  $\boldsymbol{T}[u] \leftarrow$ INITFOLDEDLADDER()
  for $m \leftarrow 1, M$ **do**
    $\boldsymbol{T}[m] \leftarrow T_{min} + incr * (m-1)$
    $\boldsymbol{\phi}[u][m] \leftarrow$ RANDOM
    $\boldsymbol{\phi}_{\mathbf{min}}[u][m] \leftarrow \boldsymbol{\phi}[u][m]$
    $\mathbf{H_P}[u][m] \leftarrow (13)$
    $\boldsymbol{E}[u][m] \leftarrow (2)$
    $\boldsymbol{E}_{\mathbf{min}}[u][m] \leftarrow \boldsymbol{E}[u][m]$

▽ Run Optimization
while E_{BKS} not in E_{min} **do**
  #parallel loop threads(U)
  for $u \leftarrow 1, U$ **do**
    #parallel loop threads(C)
    for $thread \leftarrow 1, C$ **do**
      for $i \leftarrow 1, M/C$ **do**
        $m \leftarrow thread * (M/C) + i$
        RUN_R(u, m)

  #parallel loop threads(U)
  for $u \leftarrow 1, U$ **do**
    PTEXCHANGE(u)

return $\boldsymbol{E}_{\mathbf{min}}, \boldsymbol{\phi}_{\mathbf{min}}$

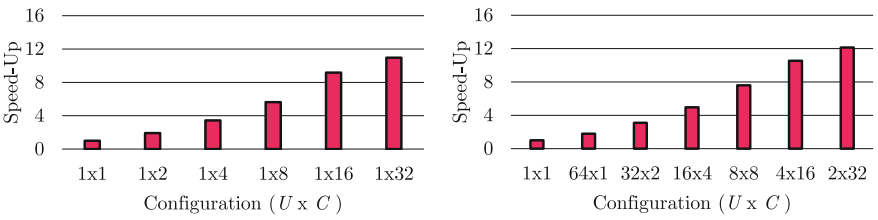
Energies are stored using fp64 while $\mathbf{F}$, $\mathbf{D}$, $\mathbf{B_P}$, and $\mathbf{H_P}$ elements are stored using fp32. Floats allow for the use of fused-multiply-add operations when implementing (17) within SWAP(). To fully utilize our available hardware, SIMD instructions were used wherever possible for significant speed-ups.

more local field updates per trial on average, increasing their run-time. In our experiments, we observed that the number of trials accepted typically increases linearly as T is increased as demonstrated in Fig. 4a. To load-balance the threads, upon initialization of the system, replicas are *folded* and assigned in pairs to threads as shown in Fig. 4b.

The replica-to-thread assignments are static throughout a run to ensure that there is minimal movement of $\mathbf{H_P}$ data between cores. Although the temperature exchanges between replicas can cause load imbalance due to the static folding, their stochastic nature ensures that they are temporary with minimal effects.

4.4 Multi-threaded Configuration Selection

To find the optimal number of engines (U) and *threads-per-engine* (C), we ran all instances within the *sko* and *taiXXb* sets from QAPLIB [6] (excluding *tai*150*b*) 100 times each and recorded the average *time-to-optimum* (TtO) across all 100 runs for each instance. The TtO, reported in seconds, is measured as the average time for a solver to reach the BKS of a problem. We measured TtO values over the selected instances for a system with $U = 1$ across different C values and compared the TtO of each instance against those of a single-threaded system ($U \times C = 1 \times 1$). Figure 5a depicts the average speed-up of a single-engine system as C is varied relative to a 1×1 system, showing that the execution time decreases as the number of threads is increased with diminishing returns. We repeated this experiment, testing different combinations of U and C to find the optimal system configuration. Figure 5b compares the speed-up of different configurations relative to a 1×1 system with the 2×32 configuration having the highest average speed-up despite having no load-balancing. This implies that extra engines, even with load-balancing, cannot make-up for their addtional data movement costs.



Speed-up v Cores-per-Engine **(a)**

Speed-up v Multi-Engine Configs. **(b)**

Fig. 5. Speed-up across multi-core configurations

5 Experiments and Results

We benchmark our PBM optimizers using a 64-core AMD Threadripper 3990X system with 128 GB of DDR4 running on CentOS 8.1. Our system was coded in

C++, using the OpenMP API for multi-threading, and compiled with GCC-9. Two separate variants of our solver were benchmarked: PBM ($U = 1$, $C = 1$) and PBM64 ($U = 2$, $C = 32$). We compare the performance of our systems against eight state-of-the-art solvers, described in Table 1. Solvers [4,5,22] use a preset iteration/time limit as their termination criterion while [15,18–20,24] terminate as soon as the *BKS* is reached. All metrics are taken directly from the respective papers. We benchmarked instances from the QAP Library [6] along with ones created by Palubeckis [21] and Drezner [9]. The sets of instances from Palubeckis and Drezner are generated to be difficult with known optima, with the Drezner set being specifically ill-conditioned for meta-heuristics.

Table 1. State-of-the-art solver descriptions

ID	Year	Algorithm	Platform	Hardware
This	2020	Boltzmann Machine + Parallel Tempering	CPU	AMD 3990X
[4]	2019	Hunting Search	CPU	Intel i5-4300
[5]	2017	Break-Out Local Search	CPU	Intel i7-6700
[15]	2017	Break-Out Local Search	FPGA	Xilinx ZCU102
[18]	2018	Genetic Algorithm + Extremal Optimization + Tabu Search	CPU	8 × AMD 6376
[19]	2016	Extremal Optimization + Tabu Search	CPU	8 × AMD 6376
[20]	2016	Extremal Optimization	CPU	8 × AMD 6376
[22]	2018	Multistart Simulated Annealing	GPU	NVidia Titan X
[24]	2012	Ant Colony Optimization	GPU	4 × NVidia GTX480

5.1 Benchmarks: Previously Solved QAP Instances

Table 2 contains TtO values for our two PBM variants and the solvers in Table 1, across some of the most difficult instances from literature that at least one other solver was able to solve with a 100% success rate within a five minute time-out window. The *bur* set, while not difficult, was included as it is the only *asm* set used in literature. The TtO reported for PBM is the average value across 10 consecutive runs with a 5 min time-out window for each run. For the solvers in Table 1, we report only the TtO from the best solver for each instance. The TtOs where PBM or PBM64 outperform the best solver are highlighted in Table 2. In 44 out of 60 instances, the fastest TtO is reported by one or both of our PBM variants with speed-ups in excess of 10× for PBM and 100× for PBM64 on certain instances. Of the remaining 16 instances, PBM64 has either identical or marginally slower performance compared to the best reported solver.

5.2 Benchmarks: Unsolvable QAP Instances

Table 3 contains performance comparisons between PBM64 and the solver from [19], ParEOTS, across QAP instances that no solver to date could consistently solve within a 5 min time-out window. As neither ParEOTS or PBM64 have a 100% success rate on these instances, we also compare their Average Percentage

Table 2. Time-to-optimum (s) comparisons across difficult QAP instances

	ID	Best Solver	PBM	PBM64		ID	Best Solver	PBM	PBM64		ID	Best Solver	PBM	PBM64
bur26a	[20]	0.027	0.017	0.010	**Inst50**	[19]	17	82.55	5.525	**tai20a**	[20]	2.637	0.720	0.035
bur26b	[20]	0.021	0.043	0.013	**Inst60**	[19]	67	58.27	1.606	**tai25a**	[20]	6.330	0.441	0.029
bur26c	[20]	0.009	0.022	0.011	**Inst70**	[19]	127	101.49	10.67	**tai30a**	[22]	0.76	0.560	0.093
bur26d	[4]	7.951	0.109	0.022	**Inst80**	[19]	116	X	27.66	**tai35a**	[5]	19.20	2.620	0.149
bur26e	[20]	0.010	0.042	0.012	**sko42**	[15]	0.330	0.130	0.023	**tai40a**	[19]	64.00	X	93.68
bur26f	[20]	0.009	0.176	0.012	**sko49**	[15]	2.460	0.972	0.081	**tai12b**	[20]	0.001	0.004	0.009
bur26g	[20]	0.006	0.052	0.012	**sko56**	[19]	0.600	0.891	0.082	**tai15b**	[20]	0.001	0.012	0.010
bur26h	[20]	0.010	0.037	0.011	**sko64**	[19]	1.300	0.568	0.046	**tai20b**	[18]	∼0.0	0.010	0.010
dre15	[19]	∼0.0	0.016	0.010	**sko72**	[19]	8.700	2.733	0.205	**tai25b**	[18]	∼0.0	0.030	0.012
dre18	[19]	∼0.0	0.017	0.026	**sko81**	[18]	22.40	4.125	0.210	**tai30b**	[18]	0.100	0.076	0.017
dre21	[19]	∼0.0	0.054	0.023	**sko90**	[19]	92.00	8.016	0.487	**tai35b**	[19]	0.200	0.168	0.021
dre24	[19]	∼0.0	0.139	0.028	**sko100a**	[19]	69.00	9.597	0.561	**tai40b**	[20]	0.061	0.149	0.016
dre28	[19]	0.1	0.299	0.053	**sko100b**	[19]	45.00	6.772	0.574	**tai50b**	[24]	0.200	1.938	0.111
dre30	[19]	0.1	0.483	0.087	**sko100c**	[18]	56.00	16.02	0.747	**tai60b**	[24]	0.400	4.064	0.222
dre42	[19]	0.7	3.069	0.276	**sko100d**	[19]	37.00	9.579	0.841	**tai80b**	[24]	5.500	9.332	0.594
dre56	[19]	5.6	29.25	1.590	**sko100e**	[19]	47.00	6.138	0.609	**tai100b**	[24]	10.10	6.766	0.481
dre72	[19]	26	152.7	11.01	**sko100f**	[19]	57.00	10.78	0.647	**tho30**	[20]	0.235	0.093	0.019
Inst20	[19]	∼0.0	0.144	0.024	**tai12a**	[20]	0.011	0.004	0.009	**tho40**	[19]	0.400	1.113	0.245
Inst30	[19]	0.1	3.504	0.226	**tai15a**	[20]	0.089	0.040	0.012	**wil50**	[20]	27.54	0.390	0.036
Inst40	[19]	4.0	21.92	0.812	**tai17a**	[20]	0.292	0.053	0.012	**wil100**	[19]	97.00	15.49	0.770

Table 3. Performance across unsolvable instances

	BKS	ParEOTS [19]			PBM64				BKS	ParEOTS [19]			PBM64		
		#bks	APD	Time	#bks	APD	Time			#bks	APD	Time	#bks	APD	Time
dre90	1838	9	0.968	167	9	0.870	69.33	**tai60a**	7205962	3	0.146	255	0	0.997	300.0
dre110	2264	6	6.334	223	6	4.487	229.9	**tai80a**	13499184	0	0.364	300	0	1.831	300.0
dre132	2744	1	22.78	294	3	7.048	244.6	**tai100a**	21052466	0	0.298	300	0	1.709	300.0
Inst100	15008994	1	0.120	300	0	0.185	300.0	**tai150b**	498896643	0	0.061	300	9	∼0.0	111.4
Inst150	58352664	0	0.126	300	0	0.171	300.0	**tai256c**	44759294	0	0.178	300	0	0.139	300.0
Inst200	75405684	0	0.125	300	0	0.153	300.0	**tho150**	8133398	1	0.007	291	0	0.031	300.0
tai50a	4938796	3	0.077	264	3	0.151	262.9								

Deviation, calculated as $APD = 100 \times (Avg - BKS)/BKS$. Avg is calculated as the average of the best cost found in each run. We benchmarked PBM64 using the same procedure reported in [19], running each instance 10 times with a time-out window of 5 min and reporting the average time of the 10 runs along with the number of runs that reached the BKS, $\#bks$

PBM64 displays better performance on the *dre* instances and has a near 100% success rate on *tai150b*. Across other instances, ParEOTS reports equal or better performance despite PBM64 performing better on smaller instances from the same family of problems. Further testing is required to compare the TtO of ParEOTS and PBM64 if ran without a time-out limit.

6 Conclusion

We demonstrated a Permutational Boltzmann Machine with Parallel Tempering, that is capable of solving NP-Hard problems such as QAP in excess of 100× faster than other state-of-the-art solvers. The speed of the PBM is attributed to its simple structure where we can utilize parallelism through the parallel execution of its replicas on dedicated computational units along with using SIMD instructions when performing local field updates. Though our PBM + PT system, which uses a 64-core CPU, was the fastest in solving the majority of the QAP test cases by a wide margin, its flexibility allows it to be scaled to match the user's available hardware while maintaining competitive performance with other state-of-the-art solvers.

References

1. Aarts, E., Korst, J.: Simulated Annealing and Boltzmann Machines (1988)
2. Aarts, E.H.L., Korst, J.H.M.: Boltzmann machines as a model for parallel annealing. Algorithmica **6**(1–6), 437–465 (1991). https://doi.org/10.1007/bf01759053
3. Aarts, E.H., Korst, J.H.: Boltzmann machines for travelling salesman problems. Eur. J. Oper. Res. **39**(1), 79–95 (1989). https://doi.org/10.1016/0377-2217(89)90355-x
4. Agharghor, A., Riffi, M., Chebihi, F.: Improved hunting search algorithm for the quadratic assignment problem. Indonesian J. Electr. Eng. Comput. Sci. **14**, 143 (2019). https://doi.org/10.11591/ijeecs.v14.i1.pp143-154
5. Aksan, Y., Dokeroglu, T., Cosar, A.: A stagnation-aware cooperative parallel breakout local search algorithm for the quadratic assignment problem. Comput. Ind. Eng. **103**, 105–115 (2017). https://doi.org/10.1016/j.cie.2016.11.023
6. Burkard, R.E., Karisch, S.E., Rendl, F.: QAPLIP-a quadratic assignment problem library. J. Global Optim. **10**(4), 391–403 (1997). https://doi.org/10.1023/A:1008293323270
7. Dabiri, K., Malekmohammadi, M., Sheikholeslami, A., Tamura, H.: Replica exchange MCMC hardware with automatic temperature selection and parallel trial. IEEE Trans. Parallel Distrib. Syst. **31**(7), 1681–1692 (2020). https://doi.org/10.1109/TPDS.2020.2972359
8. d'Anjou, A., Grana, M., Torrealdea, F., Hernandez, M.: Solving satisfiability via Boltzmann machines. IEEE Trans. Pattern Anal. Mach. Intell. **15**(5), 514–521 (1993). https://doi.org/10.1109/34.211473
9. Drezner, Z., Hahn, P.M., Taillard, É.D.: Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for metaheuristic methods. Ann. Oper. Res. **139**(1), 65–94 (2005). https://doi.org/10.1007/s10479-005-3444-z
10. Earl, D.J., Deem, M.W.: Parallel tempering: theory, applications, and new perspectives. Phys. Chem. Chem. Phys. **7**(23), 3910 (2005). https://doi.org/10.1039/b509983h
11. Gloria, A.D., Faraboschi, P., Olivieri, M.: Clustered Boltzmann machines: massively parallel architectures for constrained optimization problems. Parallel Comput. **19**(2), 163–175 (1993). https://doi.org/10.1016/0167-8191(93)90046-n

12. Glover, F., Kochenberger, G., Du, Yu.: Quantum bridge analytics I: a tutorial on formulating and using QUBO models. 4OR **17**(4), 335–371 (2019). https://doi.org/10.1007/s10288-019-00424-y
13. Hinton, G.E., Sejnowski, T.J., Ackley, D.H.: Boltzmann machines: constraint satisfaction networks that learn. Carnegie-Mellon University, Department of Computer Science Pittsburgh (1984)
14. Hukushima, K., Nemoto, K.: Exchange Monte Carlo method and application to spin glass simulations. J. Phys. Soc. Jpn. **65**(6), 1604–1608 (1996)
15. Kanazawa, K.: Acceleration of solving quadratic assignment problems on programmable SoC using high level synthesis. In: FSP 2017; Fourth International Workshop on FPGAs for Software Programmers, pp. 1–8 (2017)
16. Korst, J.H., Aarts, E.H.: Combinatorial optimization on a Boltzmann machine. J. Parallel Distrib. Comput. **6**(2), 331–357 (1989). https://doi.org/10.1016/0743-7315(89)90064-6
17. Lawler, E.L.: The quadratic assignment problem. Manage. Sci. **9**(4), 586–599 (1963). https://doi.org/10.1287/mnsc.9.4.586
18. López, J., Múnera, D., Diaz, D., Abreu, S.: Weaving of metaheuristics with cooperative parallelism. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) PPSN 2018. LNCS, vol. 11101, pp. 436–448. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99253-2_35
19. Munera, D., Diaz, D., Abreu, S.: Hybridization as cooperative parallelism for the quadratic assignment problem. In: Blesa, M.J., et al. (eds.) HM 2016. LNCS, vol. 9668, pp. 47–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39636-1_4
20. Munera, D., Diaz, D., Abreu, S.: Solving the quadratic assignment problem with cooperative parallel extremal optimization. In: Chicano, F., Hu, B., García-Sánchez, P. (eds.) EvoCOP 2016. LNCS, vol. 9595, pp. 251–266. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30698-8_17
21. Palubeckis, G.: An algorithm for construction of test cases for the quadratic assignment problem. Informatica Lith. Acad. Sci. **11**, 281–296 (2000)
22. Sonuc, E., Sen, B., Bayir, S.: A cooperative GPU-based parallel multistart simulated annealing algorithm for quadratic assignment problem. Eng. Sci. Technol. Int. J. **21**(5), 843–849 (2018). https://doi.org/10.1016/j.jestch.2018.08.002
23. Swendsen, R.H., Wang, J.S.: Replica Monte Carlo simulation of spin-glasses. Phys. Rev. Lett. **57**(21), 2607–2609 (1986). https://doi.org/10.1103/physrevlett.57.2607
24. Tsutsui, S.: ACO on multiple GPUs with CUDA for faster solution of QAPs. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) PPSN 2012. LNCS, vol. 7492, pp. 174–184. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32964-7_18