

Multi-core computation of transfer matrices for strip lattices in the Potts model

Cristobal A. Navarro
 Department of Computer Science
 Universidad de Chile
 Santiago, Chile
 Email: crinavar@dcc.uchile.cl

Fabrizio Canfora
 Centro de Estudios científicos (CECS)
 Valdivia, Chile
 Email: canfora@cecs.cl

Nancy Hitschfeld
 Department of Computer Science
 Universidad de Chile
 Santiago, Chile
 Email: nancy@dcc.uchile.cl

Abstract—The transfer-matrix technique is a convenient way for studying strip lattices in the Potts model since the computational costs depend just on the periodic part of the lattice and not on the whole. However, even when the cost is reduced, the transfer-matrix technique is still an *NP-hard* problem since the time $T(|V|, |E|)$ needed to compute the matrix grows exponentially as a function of the graph width. In this work, we present a parallel transfer-matrix implementation that scales performance under multi-core architectures. The construction of the matrix is based on several repetitions of the deletion-contraction technique, allowing parallelism suitable to multi-core machines. Our experimental results show that the multi-core implementation achieves speedups of $3.7X$ with $p = 4$ processors and $5.7X$ with $p = 8$. The efficiency of the implementation lies between 60% and 95%, achieving the best balance of speedup and efficiency at $p = 4$ processors for actual multi-core architectures. The algorithm also takes advantage of the lattice symmetry, making the transfer matrix computation to run up to $2X$ faster than its non-symmetric counterpart and use up to a quarter of the original space.

I. INTRODUCTION

The Potts model [1] has been widely used to study physical phenomena of *spin lattices* such as phase transitions in the thermo-dynamical equilibrium. Topologies such as triangular, honeycomb, square, kagome among others are of high interest and are being studied frequently (see [2], [3], [4], [5]). When the number of possible spin states is set to $q = 2$, the Potts model becomes the classic Ising model [6] which has been solved analytically for the whole plane by Onsager [7]. Unfortunately, for higher values of q no full-plane solution has been found yet. Therefore, studying strip lattices becomes a natural approach for achieving an exact but finite representation of the bidimensional plane. The wider the strip, the better the representation. Hopefully, by increasing the width enough, some properties of the full plane would emerge.

One known technique for obtaining the partition function of a strip lattice is to compute a transfer matrix based on the periodic information of the system. One should be aware however that building the transfer matrix is not free of combinatorial computations and exponential cost algorithms. In fact, the problem requires the computation of partition functions which are *NP-hard* problems [8].

With the evolution of multi-core CPUs towards a higher amount of cores, parallel computing is not anymore limited to clusters or super-computing; workstations can also provide

high performance computing. It is in this last category where most of the scientific community lies, therefore parallel implementations for multi-core machines are the ones to have the biggest impact. Latest work in the field of the Potts model has been focused on parallel probabilistic simulations [9], [10] and new sequential methods for computing the exact partition function of a lattice [11], [12], [13]. To the best of our knowledge, there has not been published research regarding parallel multi-core performance of exact transfer-matrix algorithms in the Potts model. The closest related research regarding this subject has been the massive parallel GPU implementations of the Monte Carlo algorithms [9], [14], which is out of the scope of this research since they are not exact. Even when the exact methods have much higher cost than probabilistic ones, they are still important because one can obtain exact behavior of the thermo-dynamical properties of the system such as the free energy, magnetization and specific heat. Once the matrix is computed, it can be evaluated and operated as many times as needed. It is important then to provide a fast way for computing the matrix in its symbolic form and not numerically, since the latter would imply a whole re-computation of the transfer matrix each time a parameter is modified. In this work, we have achieved an implementation that computes the symbolic transfer matrix and scales performance as more processors are available. The implementation can also solve problems larger than the system's available memory since it uses a secondary memory strategy, never storing the full matrix in memory.

The paper is organized as follows: section (II) covers preliminary concepts as well as related work, sections (III) and (IV) explain the details of the algorithm and the additional optimizations to the implementation. In section (VI) we present experimental results such as run time, speedup, efficiency and knee, using different amount of processors. Section (VII) discusses our main results and concludes the impact of the work for practical usage.

II. PRELIMINARIES AND RELATED WORK

Let $G = (V, E)$ be a lattice with $|V|$ vertices, $|E|$ edges and s_i be the state of a *spin* of G with $i \in [1..q]$. The Potts partition function $Z(G, q, \beta)$ is defined as

$$Z(G, q, \beta) = \sum_r e^{-\beta h(G_r)} \quad (1)$$

where $\beta = \frac{1}{K_B T}$, K_B is the Boltzmann constant and $h(G_r)$ is the energy of the lattice at a given state G_r ¹. The Potts model defines the energy of a state G_r with the following Hamiltonian:

$$h(G_r) = -J \sum_{\langle i,j \rangle \in G_r} \delta_{s_i, s_j} \quad (2)$$

Where $\langle i, j \rangle$ corresponds to the edge from vertex v_i to v_j , $r \in [1..q^{|V|}]$, J is the interaction energy ($J < 0$ for *anti-ferromagnetic* and $J > 0$ for *ferromagnetic*) and δ_{s_i, s_j} corresponds to the *Kronecker delta* evaluated at the pair of spins $\langle i, j \rangle$ with states s_i, s_j and expressed as

$$\delta_{s_i, s_j} = \begin{cases} 1 & \text{if } s_i = s_j \\ 0 & \text{if } s_i \neq s_j \end{cases} \quad (3)$$

The free energy of the system is computed as:

$$F = -T \log_e(Z) \quad (4)$$

As the lattice becomes bigger in the number of vertices and edges, the computation of equation (1) becomes rapidly intractable with an exponential cost of $\Theta(q^{|V|})$. In practice, an equivalent recursive method is more convenient than the original definition.

The *deletion-contraction* method, or DC method, was initially used to compute the Tutte polynomial [15] and was then extended to the Potts model after the relation found between the two (see [16], [17]). DC re-defines $Z(\cdot)$ as the following recursive equation

$$Z(G, q, v) = Z(G - e, q, v) + vZ(G/e, q, v) \quad (5)$$

$G - e$ is the *deletion* operation, G/e is the *contraction* operation and the auxiliary variable $v = e^{-\beta J} - 1$ makes $Z(\cdot)$ a polynomial. There are three special cases when DC can perform a recursive step with linear cost:

$$Z(G, q, v) = \begin{cases} (q + v)Z(G/e, q, v); & \text{if } \{e\} \text{ is a spike.} \\ (1 + v)Z(G - e, q, v); & \text{if } \{e\} \text{ is a loop.} \\ q^{|V|}; & \text{if } E = \{\emptyset\}. \end{cases} \quad (6)$$

The computational complexity of DC has a direct upper bound of $O(2^{|E|})$. When $|E| \gg |V|$ a tighter bound is known based on the Fibonacci sequence complexity [18]; $O(\left(\frac{1+\sqrt{5}}{2}\right)^{|V|+|E|})$. In general, the time complexity of DC can be written as

$$T(G) = \min\left(O(2^{|E|}), O\left(\frac{1+\sqrt{5}}{2}\right)^{|V|+|E|}\right) \quad (7)$$

Haggard's et al. [19] work is considered the best implementation of DC for computing the Tutte polynomial for any given graph. Their algorithm, even when it is exponential in time, reduces the computation tree in the presence of loops, multi-edges, cycles and biconnected graphs (as one-step reductions). An important contribution by the authors is that by using a cache, some computations can be reused (i.e sub-graphs that are isomorphic to the ones stored in the cache do not need to be computed again). An alternative algorithm was proposed by Björklund et al.[20] which accomplishes exponential time

only on the number of vertices; $O(2^n n^{O(1)})$ with $n = |V|$. Asymptotically their method is better than DC considering that many interesting lattices have more edges than vertices. However, Haggard *et al.* [19] have stated that the memory usage of Björklund's method is too high for practical usage. For the case of strip lattices, a full application of DC is not practicable since the exponential cost would grow as a function of the total amount of edges, making the computation rapidly intractable. The *transfer matrix* technique, mixed with DC is a better choice since the exponential cost will not depend on the total length of the strip, but instead just on the size of the period.

As soon as the matrix is built, the remaining computations become numerical and less expensive; *i.e.*, matrix multiplications (for finite length) or eigenvalue computations (for infinite length). Bedini *et al.* [12] proposed a method for computing the partition function of arbitrary graphs using a tree-decomposed transfer matrix. In their work, the authors obtain a sub-exponential algorithm based on arbitrary heuristics for finding a good tree decomposition of the graph. This method is the best known so far for arbitrary graphs, but when applied to strip lattices, it costs just as the traditional methods, *i.e.*, the tree-width becomes the width of the strip and the cost is proportional to the Catalan number of the width. Therefore, it is of interest to use parallelism in order to improve the performance of the transfer matrix problem when dealing with strip lattices.

A strip lattice is by default a bidimensional graph $G = (V, E)$ that is periodic at least along one dimension. It can be perceived as a concatenation of n subgraphs K sharing their boundary vertices and edges. Let P be the set of all possible strip lattices, then we formally define G :

$$G\{V, E\} \in P \Leftrightarrow \exists K = \{V', E'\} : G = \bigotimes_1^n K \quad (8)$$

\bigotimes is the special operator for concatenating the periods defined as $K_i = (E_i, V_i)$ of height m . Each period connects to the other periods K_{i-1} and K_{i+1} , except for K_1 and K_n which are the external ones and connect to K_2 and K_{n-1} respectively (see Figure 1).

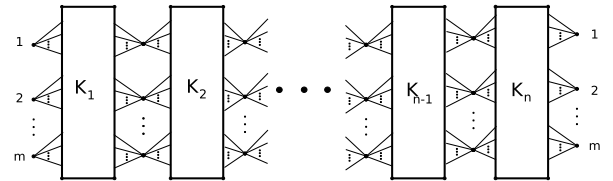


Fig. 1: Strip lattice model with length n and width m .

The main computational challenge when using a matrix transfer based algorithm is the cost of building it because its size increases exponentially as a function of the width of the strip. The problem of the matrix size has been improved by analytic techniques [21]. However, the authors specify that these techniques are only applicable to square and triangular lattices using values of $q = 2$ and $q = 3$ (Ising and three-state Potts respectively). For this reason, we prefer to use a more general transfer matrix based on its combinatorial aspects, with

¹A state G_r is a distribution of spin values on the lattice. It can be seen as a graph with an specific combination of values on the vertices.

the advantage of being useful to any lattice topology, and allowing any value of $q \in \mathbb{R}$.

To the best of our knowledge, there is no mention on the effectiveness on parallelizing transfer matrix algorithms for strip lattices in the Potts model. The core of our work is to focus on the multi-core parallel capabilities of a practical transfer matrix method and confirm or deny the factibility of such computation to run in parallel. In order to achieve parallelism, we use a transfer matrix algorithm based on a modified *deletion-contraction* (DC) scheme. It is in fact a *partial DC* that stops its recursion when the edges to be processed connect a pair of vertices of the next period. As a result, the partial DC generates many partial partition functions associated to combinatorial labels located at the leaves of the recursion tree. These partial partition functions, when grouped by their combinatorial label, make a row of the transfer matrix. A hash table is a good choice for searching and grouping terms in the combinatorial space of the problem.

III. ALGORITHM OVERVIEW

A. Data structure

Our definition of K from section (II) (Figure 1) will be used to model our input data structure. Given any strip lattice G , only the right-most part of the strip lattice is needed, that is, K_n . We will refer to the data structure of K_n as K_{σ_1} to denote the basic case where the structure is equal to the original K_n , not having any additional modification. For simplicity and consistency, we will use a *top-down* enumeration of the vertices, such that the left boundary contains the first m vertices and the right boundary the last m ones. We will introduce the following naming scheme for left and right boundary vertices, this will be *shared* and *external* vertices, respectively. *Shared vertices* correspond to the left-most ones, *i.e.* the vertices that are shared with K_{n-1} (see Figure 2) and are indexed top-down from 0 to $m-1$. *External vertices* are the ones of the right side (*i.e.*, the right end of the whole strip lattice) of K_{σ_1} and are indexed top-down from $|V| - m - 1$ to $|V| - 1$.

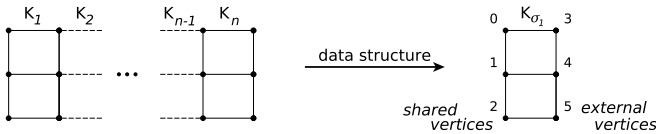


Fig. 2: Example data structure for a square lattice of width $m = 3$.

B. Computing the transfer matrix M .

Computing the transfer matrix M is a repetitive process that involves combinatorial operations over K_{σ_1} . For a better explanation of the algorithm, we introduce two terminologies; *initial configurations* and *terminal configurations*. These configurations define a combinatorial sequence of identifications for external and shared vertices, respectively, and correspond to the set of all non-crossing partitions. Given the lattice width m , the number of initial and terminal configurations is the sequence of the Catalan numbers:

$$C_m = \frac{1}{m+1} \binom{2m}{m} = \frac{(2m)!}{(m+1)!m!} = \prod_{k=2}^m \frac{m+k}{k} \quad (9)$$

Initial configurations, denoted σ_i with $i \in [0..C_m - 1]$, define a combinatorial sequence of identifications just on the external vertices. The *terminal configurations*, denoted φ_j with $j \in [0..C_m - 1]$, define a combinatorial sequence of identifications just on the shared vertices. Initial configurations generate terminal ones (using the DC method) but not vice versa. As stated before, K_{σ_1} is the basic case and matches K_n . That is, σ_1 is the initial configuration where no identifications are applied to the external vertices. It is equivalent as saying that σ_1 is the empty partition of the Catalan set. Similarly, φ_1 corresponds to the base case where no shared vertices are identified. In other words, φ_1 is the empty configuration for the Catalan set on the shared vertices. Additionally, initial and terminal configurations have a maximum of m vertices, and eventually will contain less vertices as more identifications are performed.

The idea of the algorithm is to compute the transfer matrix M in rows, by repeatedly applying the partial DC, each time to a different initial configuration K_{σ_i} , a total of C_m times. Each repetition contributes to a row of M . By default, the algorithm cannot know the C_m different sequences of *terminal* and *initial* configurations except for K_{σ_1} which is given as part of the input of the strip lattice and is the one that triggers the computation. This is indeed a problem for parallelization. To solve it, we use a recursive generator $g(A[[]], s, H, S)$ that, with the help of a hash table H , generates all the C_m configurations and stores them in an array S . $A[[]]$ is an auxiliary array that stores the intermediate auxiliary subsequences and s is the accumulated sequence of identifications. Before the first call of $g(A[[]], s, H, S)$, $A = [[0, 1, 2, \dots, m-1]]$, s is null and H as well as S are empty. $g(A[[]], s, H, S)$ is defined as:

```

g(A[[]], s, H, S) {
  if (! add_sequence(s, H, S))
    return
  for(int k=0; k<A.size(); k++){
    for(int j=1; j<A[k].size(); j++){
      for(int i=0; i<j; i++){
        if (can_identify(A[k], i, j)) {
          cA := copy(A)
          cs := copy(s)
          identify(cA, i, j, k, cs)
          divide(cA, i, j, k)
          g(cA, cs, H, S)
        }
      }
    }
  }
}

```

Basically, $g(\dots)$ performs a three-way recursive division of the domain A . For each identification pair i, j , the domain is partitioned into three sets; (1) the top vertices above i , (2) the middle vertices between i, j and (3) the vertices below j . If $|j - i| < 3$ then no set can be created in the middle. The same applies to the top and bottom sets if the distance from i or j to the boundary of the actual domain is less than 1. Each time a new identification i, j is added, the resulting configuration is checked in the hash table. If it is a new one, then it is added, otherwise it is discarded as well as further recursion computations starting from that configuration. Thanks to the hash table, repetitive recursion branches are never computed.

Once $g(\dots)$ has finished, S becomes the array of all possible configurations and H is the hash that maps configurations to indices. At this point one is ready to start computing the matrix in parallel. We start dividing the total amount of rows by the amount of processors. Each processor will be computing a total of C_m/p rows. The initial configuration sequence needed by each processor p_i is obtained in parallel by reading from $S[p_i]$. Once the configuration is read, it is applied to the external vertices of its own local copy of the base case K_{σ_1} . After each processor builds their corresponding $K_{\sigma'_i}$ graph, each one performs a DC procedure in parallel, without any communication cost. This DC procedure is only partial because edges that connect two shared vertices must never be deleted neither contracted, otherwise one would be processing vertices and edges of the next period of the lattice, breaking the idea of a transfer matrix. An example of a partial DC is illustrated in Figure 3 for the case when computing the first row.

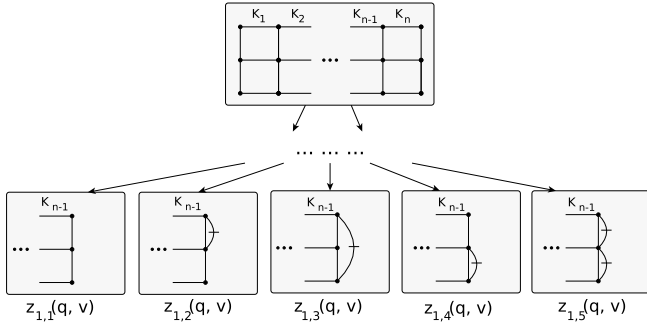


Fig. 3: An example of how terminal configurations are generated from the basic one.

When the DC procedure ends, there will be partial expressions associated to a remanent of the graph at each leaf of the recursion tree. Each remanent corresponds to the part of the graph that was not computed (*i.e.*, edges connecting *shared vertices*) and it is identified by its terminal configuration. Each one of these remanents specifies one of the C_m possible terminal configurations that can exist. For some problems, not all *terminal configurations* are generated from a single DC, but only a subset of them. That is why the generator function is so much needed in order for the algorithm to work in parallel, otherwise there would be a time dependency among the DC repetitions.

For each *terminal configuration* φ_j , its key sequence is computed dynamically along the branch taken on the DC recursion tree; each contraction contributes with a pair of indices from the vertices. Consistent terminal configurations sequences are achieved by using a small algebra that combines the identifications from contractions, made on the *shared vertices*. An identification of two shared vertices $[v_i, v_j]$ will be denoted as $\pi_{i,j}$. Each additional identification adds up to the previous ones to finally form a sequence of a terminal configuration $\pi_{x_1,y_2} + \pi_{x_2,y_2} + \dots + \pi_{x_n,y_n}$. The following properties hold true for sequences:

$$\pi_{a,b} = \pi_{b,a} \quad (10)$$

$$\pi_{a,b} + \pi_{c,d} = \pi_{c,d} + \pi_{a,b} \quad (11)$$

$$\pi_{a,b} + \pi_{b,c} = \pi_{a,b,c} \quad (12)$$

If we apply a lexicographical order to each sequence, we can avoid checking properties (10) and (11).

Each processor must group its partial expressions that share a common *terminal configuration*, so that in the end there is only one final expression $z_{i,j}(q, v)$ per *terminal configuration*. The list of final expressions associated to *terminal configurations* represents one row of the transfer matrix. The final expressions become the elements of the row and the *terminal configurations* are the keys for getting their respective column indices. The terminal configuration sequence is necessary, but not sufficient for knowing its index j in M . This is where H becomes useful for knowing with average $O(1)$ cost what is the actual index j of a given terminal configuration sequence. As a result, each thread t_i can write their final expressions $z_{i,j}(q, v)$ correctly into M . The main idea of the parallelization scheme can be illustrated by using Foster's [22] four-step strategy for building parallel algorithms; *partitioning*, *communication*, *agglomeration*, *mapping*. Figure 4 shows an example using $p = 2$.

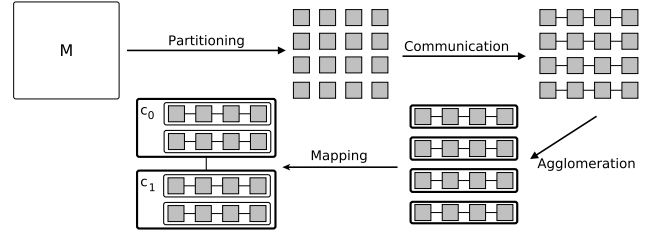


Fig. 4: The parallelism scheme under Foster's four step design strategy using two cores.

Basically, the idea is to give a small amount of B consecutive rows to each processor (for example $B = 2, 4, 8$ or 16) with an offset of $k = pB$ rows per processor. If the work per row is unbalanced, then processing is better to be asynchronous, handling the work by a master process. The asymptotic complexity for computing M under the PRAM model using the CREW variation is upper-bounded by:

$$T(\vec{Z}) = O\left(\frac{C_m}{p}(\min(2^{|E'|}, 1.6182^{|V'|+|E'|}))\right) \quad (13)$$

The complexity equals the cost of applying DC times C_m in parallel with p processors.

When all processors end, the final transfer matrix M is size $C_m \times C_m$:

$$M = \begin{pmatrix} z_{1,1}(q, v) & z_{1,2}(q, v) & \dots & z_{1,C_m}(q, v) \\ z_{2,1}(q, v) & z_{2,2}(q, v) & \dots & z_{2,C_m}(q, v) \\ \dots & \dots & \dots & \dots \\ z_{C_m,1}(q, v) & z_{C_m,2}(q, v) & \dots & z_{C_m,C_m}(q, v) \end{pmatrix} \quad (14)$$

If the strip lattice represents an infinite band, then the next step is to make a numerical evaluation on q, v and study the eigenvalues of M . If the strip lattice is finite, then an initial condition vector \vec{Z}_1 is needed. In that case, M and \vec{Z}_1 together form a partition function vector \vec{Z} based on the following recursion:

$$\vec{Z}(n) = M\vec{Z}(n-1) \quad (15)$$

By solving the recurrence, \vec{Z} becomes:

$$\vec{Z} = M^{n-1}\vec{Z}_1 \quad (16)$$

\vec{Z}_1 is computed by applying DC to each one of the C_m *terminal configurations*:

$$\vec{Z}_1 = (DC(K_{\varphi_1}), DC(K_{\varphi_2}), \dots, DC(K_{\varphi_{C_m}})) \quad (17)$$

The computation of \vec{Z}_1 has very little impact on the overall cost of the algorithm. In fact the cost is practically $O(m)$ because a terminal configuration contains mostly *spikes* and/or *loops*, which are linear in cost. Moreover, *terminal states* can be computed even faster by using the serial and parallel optimizations, but it is often not required. Computing the powers of M^{n-1} should be done in a numerical context, otherwise memory usage will become intractable.

Finally, the first element of \vec{Z} is the partition function of studied the strip lattice. After this point, \vec{Z} is used to study a wide range of physical phenomena. Álvarez and Canfora *et. al.* [23] have reported new exact results for strip lattices such as the kagome of width $m = 5$ using the sequential version of this transfer matrix algorithm.

IV. ALGORITHM IMPROVEMENTS

A. Serial and Parallel paths

The DC contraction procedure can be further optimized for graphs that present serial or parallel paths along their computation (see Figure 5).



Fig. 5: Serial and parallel paths.

Let v_a and v_b be the first and last vertices of a path, respectively. A **serial path** s is a set of edges e_1, e_2, \dots, e_n that connect sequentially $n - 1$ vertices between v_a and v_b . It is possible to process a serial path of n edges in one recursion step by using the following expression;

$$Z(K, q, v) = \left[\frac{(q + v)^n - v^n}{q} \right] Z(K_{-s}, q, v) + v^n Z(K_{/s}, q, v) \quad (18)$$

On the other hand, a **parallel path** p is a set of edges e_1, e_2, \dots, e_n where each one connects redundantly v_a and v_b , forming n possible paths between v_a and v_b . It is also possible to process a parallel path of n edges in one recursion step by using the following expression;

$$Z(K, q, v) = Z(K_{-p}, q, v) + [(1 + v)^n - 1] Z(K_{/p}, q, v) \quad (19)$$

B. Lattice Symmetry

A very important optimization is to detect the lattice's symmetry when building the matrix. By detecting symmetry, the matrix size is significantly lower because all symmetric pair of *terminal states* are grouped as one terminal state. As the width of the strip lattice increases, the number of symmetric states increases too, leading to matrices almost a half the dimension of the original M . We establish symmetry between

two *terminal configurations* φ_a and φ_b with keys π_{a_1, \dots, a_n} and π_{b_1, \dots, b_n} respectively in the following way:

$$\pi_{a_1, \dots, a_n} = \pi_{b_1, \dots, b_n} \Leftrightarrow a_i = (m - 1) - b_{n-i+1} \quad (20)$$

Under symmetry, the resulting matrix has a different numerical sequence of sizes than the original M which obeyed the Catalan numbers. In this case, we denote the size of the matrix as $D(m)$ and it is equal to:

$$D(m) = \frac{C_m}{2} + \frac{m!}{2 \lfloor \frac{m}{2} \rfloor!} \quad (21)$$

As m grows, the $\frac{C_m}{2}$ term increases faster than the second term. For big values of m , $D(m) \approx \frac{C_m}{2}$. Table (I) shows the rate at which a non-symmetric and symmetric matrix grows as m increases.

TABLE I: Growth rate of the size of M under non-symmetric and symmetric cases.

m	non-sym	sym
2	2	2
3	5	4
4	14	10
5	42	26
6	132	76
7	429	232
8	1430	750
9	4862	2494
10	16796	8524

V. IMPLEMENTATION

We made two implementations of the parallel algorithm. One using OpenMP [24] and the other one using MPI [25]. We observed that the MPI implementation achieved better performance than the OpenMP one and scaled better as the number of processors increased. For this, we decided to continue the research with the MPI implementation and have discarded the OpenMP one. We chose a value of $B = 4$ for the block-size (the amount of consecutive rows per process). The value was obtained experimentally by testing different values as powers of 2, in the range 1 – 64. As long as the parallelization is balanced a value of $B > 1$ is beneficial. An important aspect of our implementation is that we make each process generate its own H table and S array. This small sacrifice in memory leads to better performance than if H and M were shared among all processes. There are mainly three reasons why the replication approach is better than the sharing approach: (1) caches will not have to deal with consistency of shared data, (2) the cost of communicating the data structures is saved and (3) the allocation of the replicated data will be correctly placed on memory when working under a NUMA architecture. The last claim is true because on NUMA systems memory allocations on a given process are automatically placed in its fastest location according to the processor of the CPU. It is responsibility of the OS (or make manual mapping) to stick the process to the same processor through the entire computation.

The implementation saves each row to secondary memory as soon as it is computed. Each processor does this with its own file, therefore the matrix is fragmented into p files. This secondary memory strategy is not a problem because practical case shows that numerical evaluation is needed before using

the matrix in its non-fragmented form. In fact, fragmented files allow parallel evaluation of the matrix easier. We will not cover numerical evaluation in our experiments because it is out of the scope of this work.

VI. EXPERIMENTAL RESULTS

We performed experimental tests for the parallel transfer matrix method implemented with MPI. The computer used for all tests is listed in table II.

TABLE II: Hardware and tools used for experiments.

Hardware	Detail
CPU	AMD FX-8350 8-core 4.0GHz
Mem	8GB RAM DDR3 1333Mhz
MPI implementation	open-mpi

Our experimental design consists of measuring the parallel performance of the implementation at computing the transfer matrix of two types of strip lattices (see Figure 6); (1) *square* and (2) *kagome*.

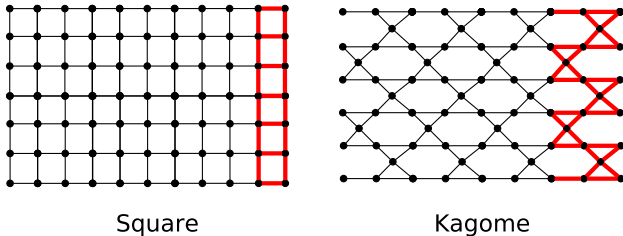


Fig. 6: The two tests to be used for the experiments. The red part is the input for the program.

A. Results on the square test

For the test of the square lattice, we test 9 different strip widths in the range $m \in [2, 10]$. For each width, we measure 8 average execution times, one for each value of $p \in [1, 8]$. As a whole, we perform a total of 72 measurements for the square test. The standard error for the average execution times is below 5%. We also compute other performance measures such as speedup, efficiency and the *knee* [26]. In these tests, we made use of the lattice symmetry for all sizes of m .

Figure 7 shows all four performance measures for the square lattice. From the results, we observe that there is speedup for every value of p as long as $m > 4$. For $m \leq 4$, the problem is not large enough to justify parallel computation, hence the overhead from MPI makes the implementation perform worse than the sequential version. The plot of the execution times confirms this behavior since the curves cross each other for $m < 4$. The maximum speedup obtained was 5.7 when using $p = 8$ processors. From the lower left graphic we can see that efficiency decreases as p increases, which is expected in every parallel implementation. What is important is that for large enough problems (*i.e.*, $m > 6$), efficiency is over 65% for all p . For the case of $p = 4$, we report 94% of efficiency, which is close to perfect linear speedup. For $m \leq 6$, the implementation is not so efficient because the amount of computation involved is not enough to keep all cores working

at full capacity. The *knee* is useful for finding the optimal value of p for a balance between efficiency and computing time. It is called knee because the hint for the optimal value of p is located in the knee of the curve (thought as a leg), that is, its lower right part. In order to know the value of p suggested by the knee, one has to count the position of the closest point to the knee region, in reverse order. Our results of the knee for $m > 6$ show that the best balance of performance and efficiency is achieved with $p = 4$ (for $m \leq 6$, the knee is not effective since there was no speedup in the first place). In other words, while $p = 8$ is faster, it is not as efficient as with $p = 4$.

B. Results on the kagome test

For the test of the kagome lattice, we used 5 different strip widths in the range $m \in [2, 6]$. For each width, we measured 8 average execution times, one for each value of $p \in [1, 8]$. As a whole, we performed a total of 40 measurements for the kagome test. The standard error for the average execution times is below 5%. Additional performance measures such as speedup, efficiency and knee have also been computed. In this test, we found that the block-size of $B = 4$ was a bad choice because the work per row was unbalanced. Instead, we found by experimentation that $B = 1$ makes the work assignment much more balanced. In this test we can only use lattice symmetry for $m = 3, 5$. We decided to run the whole benchmark without symmetry in order to maintain a consistent behavior for all values of m (but we will still report how it performs when using symmetry).

Figure 8 shows the performance results for the kagome strip test. We observe that the performance of the kagome test is similar to that of the square test, but just a little lower because the deletion-contraction repetitions on the graph are not as balanced in computational cost as in the square test. Nevertheless, performance is still significantly beneficial and the maximum speedup is still 5.1X when $p = 8$ on the largest problems. When $m > 5$, the efficiency of the parallel implementation is over 60% for all values of p . In this test, the knee is harder to identify, but by looking into detail on the largest problems one can see a small curve that suggests $p = 4$ which is in fact 90% efficient when solving large problems. For the case of $m = 4$, the knee suggests $p = 2$ processors which is also 90% efficient.

When using symmetry on both tests, we observed an extra improvement in performance of up to 2X for the largest values of m . This improvement applies to both sequential and parallel execution. The size of the matrix transfer matrix is also improved under symmetry, in the best cases we achieved almost half the dimension of the original matrix, which in practice reduces to 1/4 the space of the original non-symmetric matrix. Lattices as the kagome will only have certain values of m where it is symmetric. In the other cases, there is no other option but to do non-symmetric computation. For lattices such as the square lattice, symmetry is always present.

C. Static vs dynamic scheduler

The implementation used for all tests used a static scheduler, hence the block-size B . We also implemented an alternative version used a dynamic scheduler. The dynamic scheduler

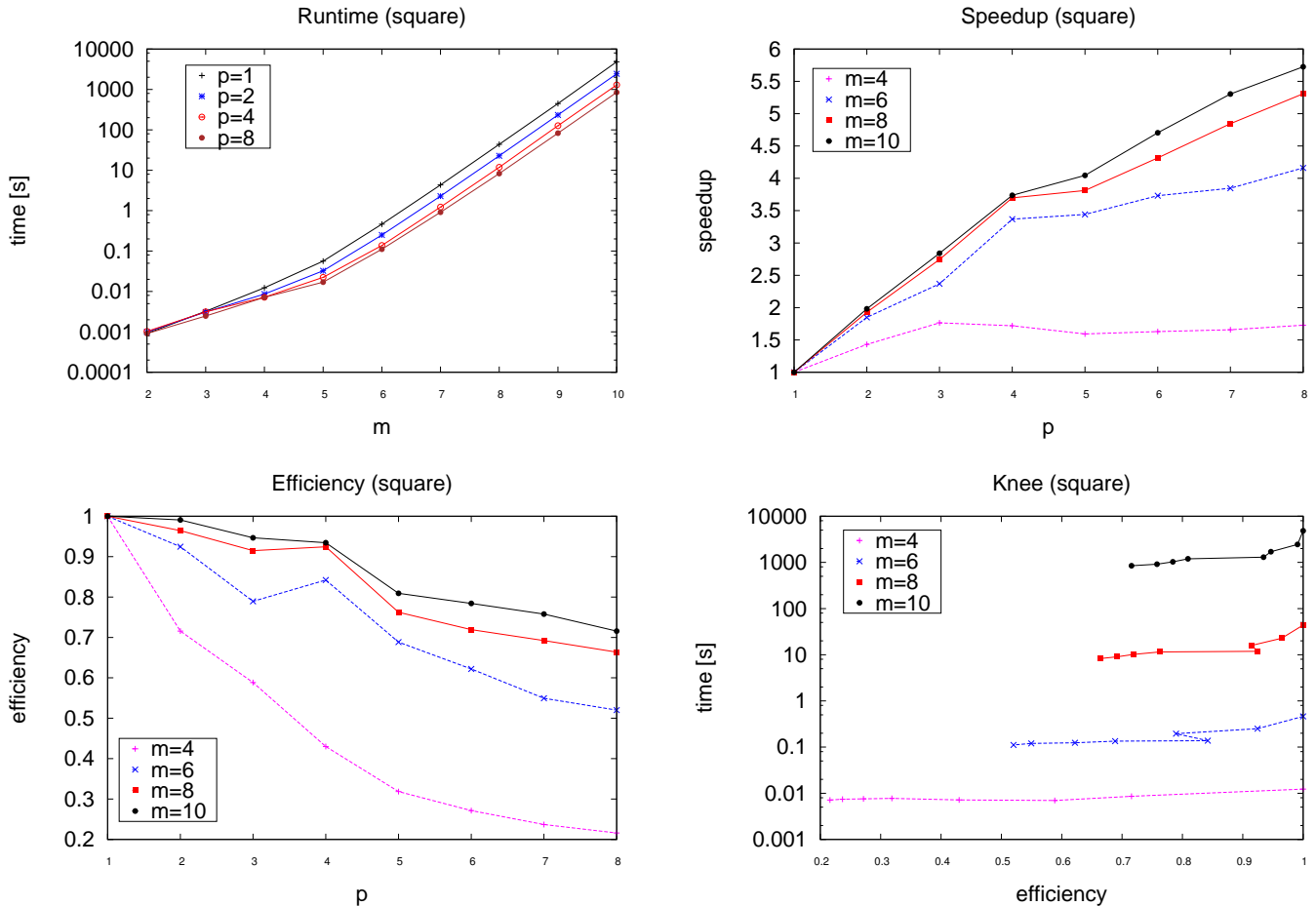


Fig. 7: Runtime, speedup, efficiency and knee for different sizes of the square strip lattice.

was implemented by using a master process that handled the jobs to the worker processes. For all of our tests, the dynamic scheduler performed slower than the static scheduler. For extreme unbalanced problems, we think that the dynamic scheduler will play a more important role. For the moment, static scheduling is the best option as long as problems stay within a moderate range of work balance.

VII. CONCLUSIONS

In this work we presented a parallel implementation for computing the transfer matrix of strip lattices in the Potts model. The implementation benefits from multi-core parallelism achieving up to 5.7X of speedup with $p = 8$ processors. Our most important result is the efficiency obtained for all speedup values, being the most remarkable one the 3.7X speedup with 95% of efficiency when using $p = 4$ processors. In the presence of symmetric strip lattices, the implementation achieved an extra 2X of performance and used almost a quarter of the space used in a non-symmetric computation.

Our experimental results serve as an empirical proof that multi-core implementations indeed help the computation of such a complex problem as the transfer matrix for the Potts model. It was important to confirm such results not only for the classic square lattice, but also for more complex lattices

with a higher amount of edges, such as the kagome lattice. In the kagome tests, the results were very similar to the ones of the square, but slightly lower because the problem becomes more unbalanced. A natural extrapolation of this behavior would suggest that very complex lattices will be even more unbalanced. We propose to use dynamic scheduling for such complex cases and static scheduling for simpler ones.

The main difficulty of this work was not the parallelization itself, but to make the problem become highly parallelizable, which is not the same. For this, we introduced a preprocessing step that generates all possible *terminal configurations*, which are critical for building the matrix. This step takes an insignificant amount of time compared to the whole problem, making it useful in practice. We also introduced smaller algorithmic improvements to the implementation; (1) fast computation of serial and parallel paths, (2) the exploit of lattice symmetry for matrix size reduction, (3) a set of algebra rules for making consistent keys in all leaf nodes and (4) a hash table for accessing column values of the transfer matrix.

In order to achieve a scalable parallel implementation, some small data structures were replicated among processors while some other data structures per processor were created within the corresponding worker process context, not in any master process. This allocation strategy results in faster

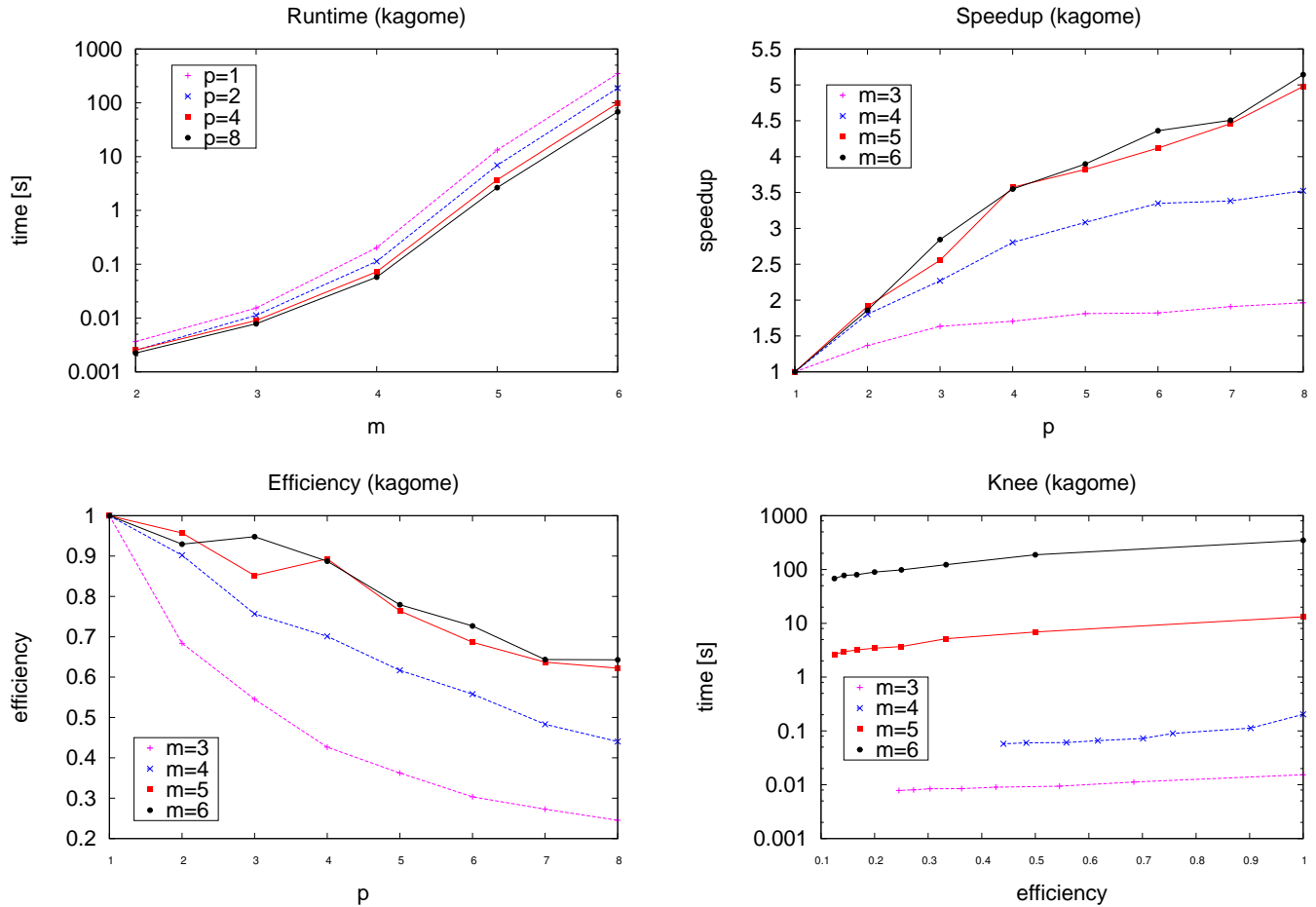


Fig. 8: Runtime, speedup, efficiency and knee for different sizes of the kagome strip lattice.

cache performance and brings up the possibility for exploiting NUMA architectures.

Even when this work was aimed at multi-core architectures, we are aware that a distributed environment can become very useful to achieve even higher parallelism. Since rows are fully independent, sets of rows can be computed on different nodes. In the case of static scheduling (*i.e.*, no master process), there will be no communication overhead because all processes will know their corresponding work based on their rank and the block value B . In the case of dynamic scheduling (*i.e.*, master process), nodes will communicate sending single byte messages, and not matrix data, resulting in a small overhead which should not become a problem if the block value B is well chosen.

It is not a problem to store the matrix fragmented into many files as long as the matrix is in its symbolic form. Practical case shows that it is first necessary to evaluate the matrix on q and v before doing any further computation. The full matrix is needed only after numerical evaluation has been performed on every row. Again, this evaluation can be done in parallel. Under this scenario, it is evident that parallel computation of transfer matrices is highly recommendable and useful in practice. The authors of this work have achieved new exact results on a wider kagome strip lattice with the help of this implementation [23].

Modern multi-core architectures have proven to be useful for improving the performance of hard problems such as the computation of the transfer matrix in the Potts model. In the future, we are interested in further improving the algorithm in order to build more efficient transfer matrices.

ACKNOWLEDGMENT

The authors would like to thank *CONICYT* for funding the PhD program of Cristóbal A. Navarro. This work was partially supported by the FONDECYT projects N° 1120495 and N° 1120352.

REFERENCES

- [1] R. B. Potts, "Some generalized order-disorder transformation," in *Transformations, Proceedings of the Cambridge Philosophical Society*, vol. 48, 1952, pp. 106–109.
- [2] S.-C. Chang and R. Shrock, "Exact potts model partition functions on strips of the honeycomb lattice," *Physica A: Statistical Mechanics and its Applications*, vol. 296, no. 1-2, p. 48, 2000. [Online]. Available: <http://arxiv.org/abs/cond-mat/0008477>
- [3] R. Shrock and S.-H. Tsai, "Exact partition functions for potts antiferromagnets on cyclic lattice strips," *Physica A*, vol. 275, p. 27, 1999. [Online]. Available: <http://arxiv.org/abs/cond-mat/9907403>

- [4] S.-C. Chang, J. Salas, and R. Shrock, "Exact potts model partition functions on wider arbitrary-length strips of the square lattice," *Journal of Statistical Physics*, vol. 107, no. 5/6, pp. 1207–1253, 2002. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S037843710100142X>
- [5] S.-C. Chang, J. L. Jacobsen, J. Salas, and R. Shrock, "Exact potts model partition functions for strips of the triangular lattice," *Physica A*, vol. 286, no. 1-2, p. 59, 2002. [Online]. Available: <http://arxiv.org/abs/cond-mat/0211623>
- [6] E. Ising, "Beitrag zur theorie des ferromagnetismus," *Zeitschrift Für Physik*, vol. 31, no. 1, pp. 253–258, 1925. [Online]. Available: <http://dx.doi.org/10.1007/BF02980577>
- [7] L. Onsager, "The effects of shape on the interaction of colloidal particles," *Annals of the New York Academy of Sciences*, vol. 51, no. 4, pp. 627–659, 1949. [Online]. Available: <http://dx.doi.org/10.1111/j.1749-6632.1949.tb27296.x>
- [8] G. J. Woeginger, "Combinatorial optimization - eureka, you shrink!" M. Jünger, G. Reinelt, and G. Rinaldi, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2003, ch. Exact algorithms for NP-hard problems: a survey, pp. 185–207. [Online]. Available: <http://dl.acm.org/citation.cfm?id=885909.885927>
- [9] E. E. Ferrero, J. P. De Francesco, N. Wolovick, and S. A. Cannas, "q-state potts model metastability study using optimized gpu-based monte carlo algorithms," *Arxiv preprint arXiv:11010876*, p. 26, 2011. [Online]. Available: <http://arxiv.org/abs/1101.0876>
- [10] J. J. Tapia and R. D'Souza, "Parallelizing the cellular potts model on graphics processing units," *Computer Physics Communications*, vol. 182, no. 4, pp. 857–865, 2011.
- [11] A. K. Hartmann, "Partition function of two- and three-dimensional potts ferromagnets for arbitrary values of $q > 0$," *PHYS.REV.LETT.*, vol. 94, p. 050601, 2005. [Online]. Available: [doi:10.1103/PhysRevLett.94.050601](https://doi.org/10.1103/PhysRevLett.94.050601)
- [12] A. Bedini and J. L. Jacobsen, "A tree-decomposed transfer matrix for computing exact potts model partition functions for arbitrary graphs, with applications to planar graph colourings," *Journal of Physics A: Mathematical and Theoretical*, vol. 43, no. 38, p. 385001, 2010. [Online]. Available: <http://stacks.iop.org/1751-8121/43/i=38/a=385001>
- [13] R. Shrock, "Exact potts model partition functions on ladder graphs," *Physica A: Statistical Mechanics and its Applications*, vol. 283, no. 3-4, p. 73, 2000. [Online]. Available: <http://arxiv.org/abs/cond-mat/0001389>
- [14] C. Castaneda-Marroquen, C. Navarrete, A. Ortega, M. Alfonseca, and E. Anguiano, "Parallel metropolis-montecarlo simulation for potts model using an adaptable network topology based on dynamic graph partitioning," in *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on*, July 2008, pp. 89–96.
- [15] W. T. Tutte, "A contribution to the theory of chromatic polynomials," *J. Math.*, vol. 6, pp. 80–91, 1954.
- [16] W. D. y Merino C, "The potts model and the tutte polynomial," *J. Math. Phys.*, vol. 43, pp. 1127–1149, 1 2000.
- [17] A. D. Sokal, "The multivariate tutte polynomial (alias potts model) for graphs and matroids," *Surveys in Combinatorics*, p. 54, 2005.
- [18] H. S. Wilf, *Algorithms and Complexity*, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2002.
- [19] G. Haggard, D. J. Pearce, and G. Royle, "Computing tutte polynomials," *ACM Trans. Math. Softw.*, vol. 37, pp. 24:1–24:17, September 2010. [Online]. Available: <http://doi.acm.org/10.1145/1824801.1824802>
- [20] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto, "Computing the tutte polynomial in vertex-exponential time," *CoRR*, vol. abs/0711.2585, 2007.
- [21] G. A. P. M. Ghaemi, "Size reduction of the transfer matrix of two-dimensional ising and potts models," 2, vol. 4, 2003.
- [22] I. Foster, "Designing and building parallel programs: Concepts and tools for parallel software engineering." Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [23] P. D. Alvarez, F. Canfora, S. A. Reyes, and S. Riquelme, "Potts model on recursive lattices: some new exact results," *European physical journal B. (EPJ B)*, vol. 85, p. 99, 2012.
- [24] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
- [25] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.
- [26] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 408–423, 1989.