The Cache-Oblivious Gaussian Elimination Paradigm: Theoretical Framework and Experimental Evaluation *

Rezaul Alam Chowdhury Vijaya Ramachandran

UTCS Technical Report TR-06-04

March 29, 2006

Abstract

The cache-oblivious Gaussian Elimination Paradigm (GEP) was introduced by the authors in [6] to obtain efficient cache-oblivious algorithms for several important problems that have algorithms with triply-nested loops similar to those that occur in Gaussian elimination. These include Gaussian elimination and LU-decomposition without pivoting, all-pairs shortest paths and matrix multiplication.

In this paper, we prove several important properties of the cache-oblivious framework for GEP given in [6], which we denote by I-GEP. We build on these results to obtain C-GEP, a completely general cache-oblivious implementation of GEP that applies to any code in GEP form, and which has the same time and I/O bounds as the earlier algorithm in [6], while using a modest amount of additional space. We present an experimental evaluation of the caching performance of I-GEP and C-GEP in relation to the traditional Gaussian elimination algorithm. Our experimental results indicate that I-GEP and C-GEP outperform GEP on inputs of reasonable size, with dramatic improvement in running time over GEP when the data is out of core.

'Tiling', an important loop transformation technique employed by optimizing compilers in order to improve temporal locality in nested loops, is a cache-aware method that does not adapt to all levels in a multi-level memory hierarchy. The cache-oblivious GEP framework (either I-GEP or C-GEP) produces system-independent I/O-efficient code for triply nested loops of the form that appears in Gaussian elimination without pivoting, and is potentially applicable to being used by optimizing compilers for loop transformation.

*Department of Computer Sciences, University of Texas, Austin, TX 78712. Email: {shaikat,vlr}@cs.utexas.edu. This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609.

1 Introduction

Memory in modern computers is typically organized in a hierarchy with registers in the lowest level followed by L1 cache, L2 cache, L3 cache, main memory, and disk, with the access time of each memory level increasing with its level. The two-level I/O model [1] is a simple abstraction of this hierarchy that consists of an internal memory of size M, and an arbitrarily large external memory partitioned into blocks of size B. The *I/O complexity* of an algorithm is the number of blocks transferred between these two levels.

The *cache-oblivious model* [10] is an extension of the two-level I/O model with the additional feature that algorithms do not use knowledge of M and B. A cache-oblivious algorithm is flexible and portable, and simultaneously adapts to all levels of a multi-level memory hierarchy. This model assumes an optimal cache replacement policy is used; standard cache replacement methods such as LRU allow for a reasonable approximation to this assumption. A well-designed cache-oblivious algorithm typically has the feature that whenever a block is brought into internal memory it contains as much useful data as possible ('spatial locality'), and also that as much useful work as possible is performed on this data before it is written back to external memory ('temporal locality').

In [6], we introduced a cache-oblivious framework, which we call *GEP* or the *Gaussian Elimination Paradigm*, for several important problems that can be solved using a construct similar to the computation in Gaussian elimination without pivoting. Traditional algorithms that use this construct fully exploit the spatial locality of data but they fail to exploit the temporal locality, and they run in $\mathcal{O}\left(n^3\right)$ time, use $\mathcal{O}\left(n^2\right)$ space and incur $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/Os. In [6] we presented a framework for in-place cache-oblivious execution of several important special cases of GEP including Gaussian elimination and LU-decomposition without pivoting, all-pairs shortest paths and matrix multiplication; this framework can also be adapted to solve important non-GEP dynamic programming problems such as sequence alignment with gaps, and a class of dynamic programs termed as 'simple-DP' [4] which includes algorithms for RNA secondary structure prediction [15], matrix chain multiplication and optimal binary search tree construction. This framework takes full advantage of both spatial and temporal locality of data to incur only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os while still running in $\mathcal{O}\left(n^3\right)$ time and without using any extra space.

In this paper we re-visit the in-place cache-oblivious framework for GEP, which we call I-GEP, and we prove theorems that establish several important properties of I-GEP. We then build on these results to derive C-GEP, which has the same time and I/O bounds as I-GEP, and is a provably correct and optimal cache-oblivious version of GEP in its full generality (though it needs to use some extra space). We present experimental results that show that both I-GEP and C-GEP significantly outperform GEP especially in out-of-core computations, although improvements in computation time are already realized during in-core computations.

We also consider generalized versions of three major I-GEP applications (Gaussian elimination without pivoting, Floyd-Warshall's APSP, and matrix multiplication) mentioned in [6]; short proofs of correctness for these applications can be obtained using results we prove for I-GEP. We provide a detailed description of the algorithm for transforming simple DP to I-GEP, which was only briefly described in [6]. We also investigate the connection between the *gap* problem (i.e., sequence alignment with gaps [11, 12, 24]) and GEP. In [6] we presented a GEP-like code derived from the classic gap dynamic program, and its I-GEP implementation; but that result is not quite correct. In this paper we present a GEP-like code and its I-GEP implementation that solve several special cases of the gap problem, and show that a small transformation on this I-GEP code actually solves the gap problem in its full generality.

One potential application of I-GEP and C-GEP framework is in compiler optimizations for the memory hierarchy. 'Tiling' is a powerful loop transformation technique employed by optimizing

compilers that improves temporal locality in nested loops. However, this technique is cache-aware, and thus does not produce machine-independent code nor does it adapt simultaneously to multiple levels of the memory hierarchy. In contrast, the cache-oblivious GEP framework produces I/O-efficient portable code for a form of triply nested loops that occurs frequently in practice.

1.1 The Gaussian Elimination Paradigm (GEP)

Let $c[1 \ldots n, 1 \ldots n]$ be an $n \times n$ matrix with entries chosen from an arbitrary set $\mathcal{S}$, and let $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ be an arbitrary function. The algorithm G given in Figure 1 modifies c by applying a given set of updates of the form $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$, where $i, j, k \in [1, n]$. By $\langle i, j, k \rangle$ $(1 \le i, j, k \le n)$ we denote an update of the form $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$, and we let Σ_G denote the set of such updates that the algorithm needs to perform.

In view of the structural similarity between the construct in G and the computation in Gaussian elimination without pivoting, we refer to this computation as the *Gaussian Elimination Paradigm* or *GEP* [6]. Many practical problems fall in this category, for example: all-pairs shortest paths, LU decomposition, and Gaussian elimination without pivoting. Other problems can be solved using GEP through structural transformation, including simple dynamic program [4] and matrix multiplication.

We note the following properties of G, which are easily verified by inspection: Given Σ_G, G applies each $\langle i, j, k \rangle \in \Sigma_G$ on c exactly once, and in a specific order. Given any two distinct updates $\langle i_1, j_1, k_1 \rangle \in \Sigma_G$ and $\langle i_2, j_2, k_2 \rangle \in \Sigma_G$, the update $\langle i_1, j_1, k_1 \rangle$ will be applied before $\langle i_2, j_2, k_2 \rangle$ if $k_1 < k_2$, or if $k_1 = k_2$ and $i_1 < i_2$, or if $k_1 = k_2$ and $i_1 = i_2$ but $j_1 < j_2$.

The running time of G is $\mathcal{O}\left(n^3\right)$ provided both the test $\langle i, j, k \rangle \in \Sigma_G$ and the update $\langle i, j, k \rangle$ in line 4 can be performed in constant time. The I/O complexity is $\mathcal{O}\left(\frac{n^3}{B}\right)$ provided the only cache misses, if any, incurred in line 4 are for accessing $c[i, j]$, $c[i, k]$, $c[k, j]$ and $c[k, k]$; i.e., neither the evaluation of $\langle i, j, k \rangle \in \Sigma_G$ nor the evaluation of f incurs any additional cache misses.

In the rest of the paper we assume, without loss of generality, that $n = 2^p$ for some integer $p \ge 0$.

The function G is a more general version of the GEP computation given in our earlier paper [6] (in Figure 6 of [6]). The GEP code in [6] can be obtained from G by setting $\Sigma_G = \{ \langle i, j, k \rangle \mid k \in [\kappa_1, \kappa_2] \wedge i \in [\iota_1(k), \iota_2(k)] \wedge j \in [\zeta_1(k, i), \zeta_2(k, i)] \}$, where κ_1 and κ_2 are problem-specific constants, and $\iota_1(\cdot)$, $\iota_2(\cdot)$, $\zeta_1(\cdot, \cdot)$ and $\zeta_2(\cdot, \cdot)$ are problem-specific functions defined in [6].

1.2 Organization of the Paper

In Section 2, we present and analyze an $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O in-place cache-oblivious algorithm, called I-GEP, which is the cache-oblivious version of the GEP framework in [6] suitably generalized to handle the more general nature of G. We prove some theorems relating the computation in I-GEP to the computation in GEP. In Section 3, we present cache-oblivious C-GEP, which solves G in its full generality with the same time and I/O bounds as I-GEP, but uses $n^2 + n$ extra space (recall that n^2 is the size of the input/output matrix c). Then in Section 4, we describe generalized versions of three major applications of I-GEP (Gaussian elimination without pivoting, matrix multiplication and Floyd-Warshall's APSP) from [6]. Succinct proofs of correctness of these I-GEP implementations can be obtained using results from Section 2.

We consider the potential application of the GEP framework in compiler optimizations in Section 5. In Section 6, we present experimental results comparing the timing and caching performance of I-GEP and C-GEP with that of the traditional GEP implementation. The study reveals that both I-GEP and C-GEP outperform GEP for reasonably large inputs during in-core computations, and perform significantly better than GEP when the computation is out-of-core.

Section 7 contains detailed treatment of two additional applications (simple DP and the gap problem)

G$(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot, \cdot, \cdot, \cdot)$ is a problem-specific function, and Σ_G is a problem-specific set of updates to be applied on c.)

1. ***for*** $k \leftarrow 1$ ***to*** n ***do***
2. ***for*** $i \leftarrow 1$ ***to*** n ***do***
3. ***for*** $j \leftarrow 1$ ***to*** n ***do***
4. ***if*** $\langle i, j, k \rangle \in \Sigma_G$ ***then*** $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$

Figure 1: GEP: Triply nested ***for*** loops typifying code fragment with structural similarity to the computation in Gaussian elimination without pivoting.

F(X, k_1, k_2)

(X is a $2^q \times 2^q$ square submatrix of c such that $X[1, 1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Function F assumes the following:

(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$
(b) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$.

The initial call to F is F$(c, 1, n)$ for an $n \times n$ input matrix c, where n is assumed to be a power of 2.)

1. ***if*** $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return*** $\{T_{X,[k_1,k_2]} = \{\langle i, j, k \rangle | i \in [i_1, i_2] \wedge j \in [j_1, j_2] \wedge k \in [k_1, k_2]\}$, *and* Σ_G *is the set of updates performed by* G *in Figure 1*$\}$
2. ***if*** $k_1 = k_2$ ***then***
3. $c[i_1, j_1] \leftarrow f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$
4. ***else*** $\{$*The top-left, top-right, bottom-left and bottom-right quadrants of* X *are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*$\}$
5. $k_m \leftarrow \left\lfloor \frac{k_1 + k_2}{2} \right\rfloor$
6. $F(X_{11}, k_1, k_m),\ F(X_{12}, k_1, k_m),\ F(X_{21}, k_1, k_m),\ F(X_{22}, k_1, k_m)$ $\{$*forward pass*$\}$
7. $F(X_{22}, k_m + 1, k_2),\ F(X_{21}, k_m + 1, k_2),\ F(X_{12}, k_m + 1, k_2),\ F(X_{11}, k_m + 1, k_2)$ $\{$*backward pass*$\}$

Figure 2: Cache-oblivious I-GEP. For several special cases of f and Σ_G in Figure 1, we show that F performs the same computation as G (see Section 4), though there are some cases of f and Σ_G where the computations return different results.

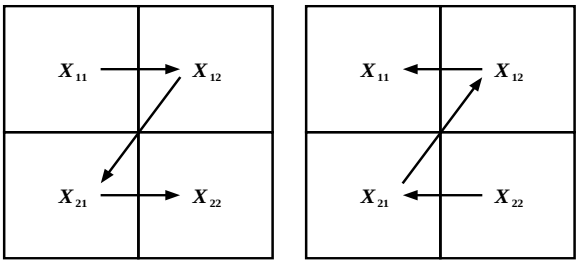

Figure 3: Processing order of quadrants of X by F: **(a)** forward pass, **(b)** backward pass.

of I-GEP which were briefly mentioned in [6]. Finally, we present some concluding remarks in Section 8.

1.3 Related Work

The *Gaussian Elimination Paradigm* was introduced by the authors in [6], where an $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O in-place cache-oblivious algorithm was given for a restricted version (as explained in Section 1.1) of the GEP code in Figure 1. This algorithm when suitably generalized to handle the general nature of G in Figure 1, is referred to as I-GEP in the current paper. The applications of I-GEP we consider in Section 4 are general versions of the three major applications (Gaussian elimination without pivoting, matrix multiplication and Floyd-Warshall's APSP) of the GEP framework originally mentioned in [6]. However, due to space limitations, all proofs of correctness were omitted from [6]. Brief descriptions of simple DP and the gap problem as applications of GEP were also included in [6].

Other known cache-oblivious algorithms for Gaussian elimination for solving systems of linear equations are based on LU decomposition. In [25, 3] cache-oblivious algorithms performing $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O operations are given for LU decomposition without pivoting, while the algorithm in [20] performs LU decomposition with partial pivoting within the same I/O bound. These algorithms use matrix multiplication and solution of triangular linear systems as subroutines. Our algorithm for Gaussian elimination without pivoting (see Section 4.1 and also [6]) is not based on LU decomposition, i.e., it does not call subroutines for multiplying matrices or solving triangular linear systems, and is thus arguably simpler than existing algorithms.

An $\mathcal{O}(mnp)$ time and $\mathcal{O}\left(m+n+p+\frac{mn+np+mp}{B}+\frac{mnp}{B\sqrt{M}}\right)$ I/O algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix is given in [10].

In [4], an $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [21], is given for simple-DP.

A cache-oblivious algorithm for Floyd-Warshall's APSP algorithm is given in [17]. The algorithm runs in $\mathcal{O}\left(n^3\right)$ time and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses. Our I-GEP implementation of Floyd-Warshall's APSP (see Section 4.3 and also [6]) produces exactly the same algorithm.

The main attraction of the Gaussian Elimination Paradigm in [6] and in this paper is that it unifies all problems mentioned above and possibly many others under the same framework, and presents a single I/O-efficient cache-oblivious solution (see C-GEP in Section 3) for all of them.

2 Analysis of I-GEP

In this section we analyze I-GEP, a recursive function F given in Figure 2 that is cache-oblivious, computes in-place, and as shown in [6], is a provably correct implementation of GEP in Figure 1 for several important special cases of f and Σ_G. (This function F does not solve GEP in its full generality, however.) We call this implementation I-GEP to denote an initial attempt at a general cache-oblivious version of GEP as well as an in-place implementation, in contrast to the new implementation (C-GEP) which we give in Section 3 that solves GEP in its full generality but uses a modest amount of additional space.

The inputs to F are a square submatrix X of $c[1 \ldots n, 1 \ldots n]$, and two indices k_1 and k_2. The top-left cell of X corresponds to $c[i_1, j_1]$, and the bottom-right cell corresponds to $c[i_2, j_2]$. These indices satisfy the following constraints:

INPUT CONDITIONS 2.1. *If $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$, k_1 and k_2 are the inputs to* F *in Figure 2, then*

(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ *for some integer* $q \geq 0$

(b) $[i_1, i_2] \neq [k_1, k_2] \quad \Rightarrow \quad [i_1, i_2] \cap [k_1, k_2] = \emptyset$ *and* $[j_1, j_2] \neq [k_1, k_2] \quad \Rightarrow \quad [j_1, j_2] \cap [k_1, k_2] = \emptyset$

Let $Y \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $Z \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Then for every entry $c[i, j] \in X$, $c[i, k]$ can be found in Y and $c[k, j]$ can be found in Z. Input condition **(a)** requires that X, Y and Z must all be square matrices of the same size. Input condition **(b)** requires that $\big(X \equiv Y\big) \ \vee \ \big(X \cap Y = \emptyset\big)$, i.e., either Y overlaps X completely, or does not intersect X at all. Similar constraints are imposed on Z, too.

The base case of F occurs when $k_1 = k_2$, and the function updates $c[i_1, j_1]$ to $f(c[i_1, j_1],\ c[i_1, k_1],\ c[k_1, j_1],\ c[k_1, k_1])$. Otherwise it splits X into four quadrants (X_{11}, X_{12}, X_{21} and X_{22}), and recursively updates the entries in each quadrant in two passes: forward (line 6) and backward (line 7). The processing order of the quadrants are shown in Figure 3. The initial function call is $\mathrm{F}(c, 1, n)$.

Properties of I-GEP. We prove two theorems that reveal several important properties of F. Theorem 2.1 states that F and G are equivalent in terms of the updates applied, i.e., both of them apply exactly the same updates on the input matrix exactly the same number of times. The theorem also states that both F and G apply the updates applicable to any fixed entry in the input matrix in exactly the same order. However, it does not say anything about the total order of the updates. Theorem 2 identifies the exact states of $c[i, k]$, $c[k, j]$ and $c[k, k]$ (in terms of the updates applied on them) immediately before $c[i, j]$ is updated to $f(c[i, j],\ c[i, k],\ c[k, j],\ c[k, k])$. One implication of this theorem is that the total order of the updates as applied by F and G can be different.

Recall that in Section 1.1 we defined Σ_G to be the set of all updates $\langle i, j, k\rangle$ performed by the original GEP algorithm G in Figure 1. Analogously, for the transformed cache-oblivious algorithm F, let Σ_F be the set of all updates $\langle i, j, k\rangle$ performed by $\mathrm{F}(c, 1, n)$.

We assume that each instruction executed by F receives a unique time stamp, which is implemented by initializing a global variable t to 0 before the algorithm starts execution, and incrementing it by 1 each time an instruction is executed (we consider only sequential algorithms in this paper). By the quadruple $\langle i, j, k, t\rangle$ we denote an update $\langle i, j, k\rangle$ that was applied at time t. Let Π_F be the set of all updates $\langle i, j, k, t\rangle$ performed by $\mathrm{F}(c, 1, n)$.

The following theorem states that F applies each update performed by G exactly once, and no other updates; it also identifies a partial order on the updates performed by F.

THEOREM 2.1. *Let Σ_G, Σ_F and Π_F be the sets as defined above. Then*

(a) $\Sigma_F = \Sigma_G$, i.e., both F *and* G *perform the same set of updates;*

(b) $\langle i, j, k, t_1\rangle \in \Pi_F \ \wedge \ \langle i, j, k, t_2\rangle \in \Pi_F \ \Rightarrow \ t_1 = t_2$, i.e., function F *performs each update $\langle i, j, k\rangle$ at most once; and*

(c) $\langle i, j, k'_1, t_1\rangle \in \Pi_F \ \wedge \ \langle i, j, k'_2, t_2\rangle \in \Pi_F \ \wedge \ k'_2 > k'_1 \ \Rightarrow \ t_2 > t_1$, i.e., function F *updates each $c[i, j]$ in increasing order of k values.*

Proof. (Sketch.) $\langle i, j, k\rangle \in \Sigma_F \ \Rightarrow \ \langle i, j, k\rangle \in \Sigma_G$ holds by the check in line 1 of Figure 2.

The reverse direction of (a) can be proved by forward induction on q, while parts (b) and (c) can be proved by backward induction on q, where $n = 2^q$. ■

We now introduce some terminology as well as two functions π and δ which will be used later in this section to identify the exact states of $c[i, k]$, $c[k, j]$ and $c[k, k]$ at the time when F is about to apply $\langle i, j, k\rangle$ on $c[i, j]$.

DEFINITION 2.1. *Let $n = 2^q$ for some integer $q > 0$.*

(a) An aligned subinterval *for n is an interval $[a, b]$ with $1 \le a \le b \le n$ such that $b - a + 1 = 2^r$ for some nonnegative integer $r \le q$ and $a = c \cdot 2^r + 1$ for some integer $c \ge 0$. The* width *of the aligned subinterval is 2^r.*

(b) An aligned subsquare *for n is a pair of aligned subintervals $[a, b], [a', b']$ with $b - a + 1 = b' - a' + 1$.*

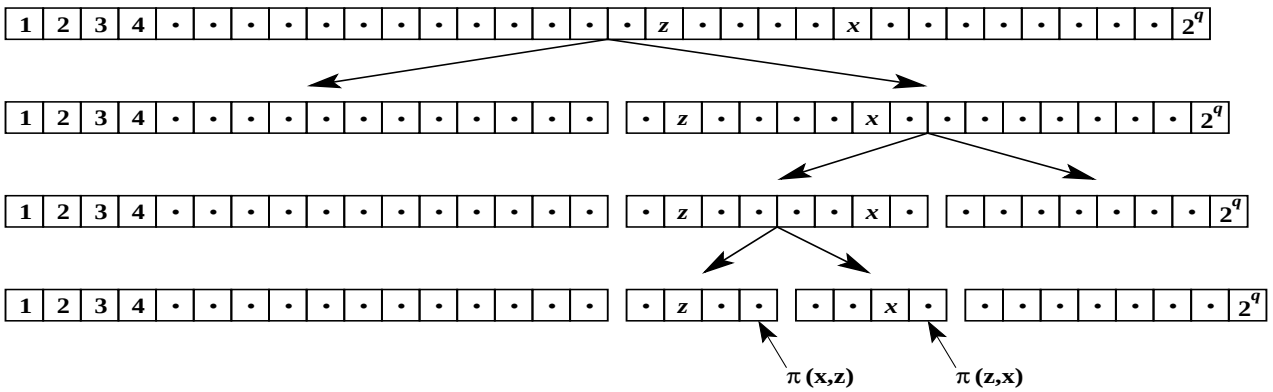

Figure 4: Evaluating $\pi(x, z)$ and $\pi(z, x)$ for $x > z$: Given $x, z \in [1, 2^q]$ such that $x > z$, we start with an initial sequence of 2^q consecutive integers in $[1, 2^q]$, and keep splitting the segment containing both x and z at midpoint until x and z fall into different segments. The largest integer in z's segment gives the value of $\pi(x, z)$, and that in x's segment gives the value of $\pi(z, x)$.

The following observation can be proved by (reverse) induction on r, starting with q, where $n = 2^q$.

OBSERVATION 2.1. *Consider the call $F(c, 1, n)$. Every recursive call is on an aligned subsquare of c, and every aligned subsquare of c of width 2^r for $r \leq q$ is invoked in exactly $n/2^r$ recursive calls on disjoint aligned subintervals $[k_1, k_2]$ of length 2^r each.*

DEFINITION 2.2. *Let x, y, and z be integers, $1 \leq x, y, z \leq n$.*

(a) For $x \neq z$ or $y \neq z$, we define $\delta(x, y, z)$ to be b for the largest aligned subsquare $[a, b], [a, b]$ that contains (z, z), but not (x, y). If $x = y = z$ we define $\delta(x, y, z)$ to be $z - 1$.

We will refer to the $[a, b], [a, b]$ subsquare as the aligned subsquare $S(x, y, z)$ for z with respect to (x, y)*; analogously, $S'(x, y, z)$ is the largest aligned subsquare $[c, d], [c', d']$ that contains (x, y) but not (z, z).*

(b) For $x \neq z$, the aligned subinterval for z with respect to x*, $I(x, z)$, is the largest aligned subinterval $[a, b]$ that contains z but not x; similarly* the aligned subinterval for x with respect to z*, $I(z, x)$, is the largest aligned subinterval $[a', b']$ that contains x but not z;*

We define $\pi(x, z)$ to be the largest index b in the aligned subinterval $I(x, z)$ if $x \neq z$, and $\pi(x, z) = z - 1$ if $x = z$.

Figures 4 and 5 illustrate the definitions of π and δ respectively. For completeness, more formal definitions of δ and π are given in the appendix. The following observation summarizes some simple properties that follow from Definition 2.2.

OBSERVATION 2.2.

(a) If $x \neq z$ or $y \neq z$ then $\delta(x, y, z) \geq z$, and if $x \neq z$ then $\pi(x, z) \geq z$; $I(x, z)$ and $I(z, x)$ have the same length while $S(x, y, z)$ and $S'(x, y, z)$ have the same size; and $S(x, y, z)$ is always centered along the main diagonal while $S'(x, y, z)$ in general will not occur along the main diagonal.

(b) If $x = y = z$ then $\delta(x, y, z) = z - 1$, and if $x = z$ then $\pi(x, z) = z - 1$.

Part (a) in the following lemma will be used to pin down the state of $c[k, k]$ at the time when update $\langle i, j, k \rangle$ is about to be applied, and parts (b) and (c) can be used to pin down the states at that time of $c[i, k]$ and $c[k, j]$, respectively. As with Observation 2.1, this lemma can be proved by backward induction on q. As before the initial call is to $\mathrm{F}(c, 1, n)$.

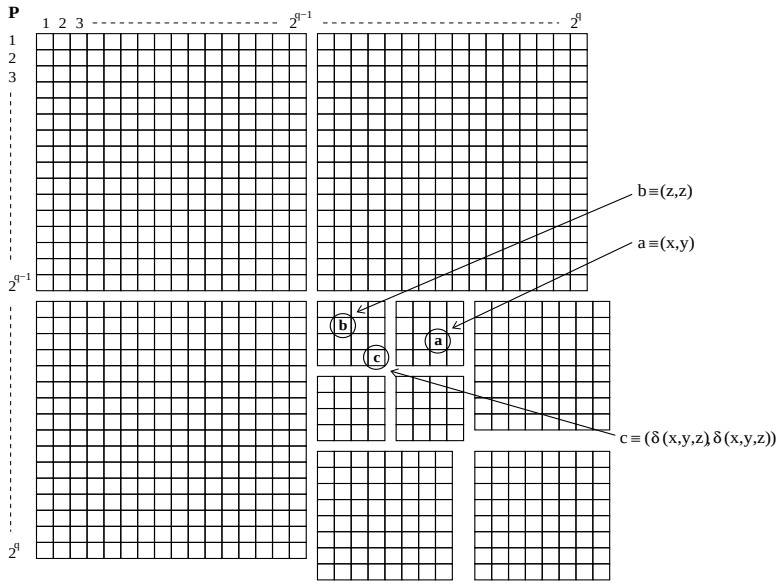

Figure 5: Evaluating $\delta(x, y, z)$: Given $x, y, z \in [1, 2^q]$ (where $q \in \mathbb{Z}^+$), such that $x \neq z \vee y \neq z$, we start with an initial square $P[1 \ldots 2^q, 1 \ldots 2^q]$, and keep splitting the square (initially the entire square P) containing both $P[x, y]$ and $P[z, z]$ into subsquares (quadrants) until $P[x, y]$ and $P[z, z]$ fall into different subsquares. The largest coordinate in $P[z, z]$'s subsquare at that point gives the value of $\delta(x, y, z)$.

LEMMA 2.1. *Let i, j, k be integers, $1 \leq i, j, k \leq n$, with not all i, j, k having the same value.*

(a) There is a recursive call $\mathrm{F}(X, k_1, k_2)$ with $k \in [k_1, k_2]$ in which the aligned subsquares $S(i, j, k)$ and $S'(i, j, k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The aligned subsquare $S(i, j, k)$ will occur only as either X_{11} or X_{22} while $S'(i, j, k)$ can occur as any one of the four subsquares except that it is not the same as $S(i, j, k)$.

If $S(i, j, k)$ occurs as X_{11} then $k \in [k_1, k_m]$ and $\delta(i, j, k) = k_m$; if $S(i, j, k)$ occurs as X_{22} then $k \in [k_m + 1, k_2]$ and $\delta(i, j, k) = k_2$.

(b) If $j \neq k$, let $T(i, j, k)$ be the largest aligned subsquare that contains (i, k) but not (i, j) and let $T'(i, j, k)$ be the largest aligned subsquare that contains (i, j) but not (i, k). There is a recursive call $\mathrm{F}(X, k'_1, k'_2)$ with $k \in [k'_1, k'_2]$ in which the aligned subsquares $T(i, j, k)$ and $T'(i, j, k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The set $\{T(i, j, k), T'(i, j, k)\}$ is either $\{X_{11}, X_{12}\}$ or $\{X_{21}, X_{22}\}$, and $\pi(j, k) = k'$, where k' is the largest integer such that (i, k') belongs to $T(i, j, k)$.

(c) If $i \neq k$, let $R(i, j, k)$ be the largest aligned subsquare that contains (k, j) but not (i, j) and let $R'(i, j, k)$ be the largest aligned subsquare that contains (i, j) but not (k, j). There is a recursive call $\mathrm{F}(X, k''_1, k''_2)$ with $k \in [k''_1, k''_2]$ in which the aligned subsquares $R(i, j, k)$ and $R'(i, j, k)$ will both occur as (different) subsquares of X being called in steps 6 and 7 of the I-GEP pseudocode. The set $\{R(i, j, k), R'(i, j, k)\}$ is either $\{X_{11}, X_{21}\}$ or $\{X_{12}, X_{22}\}$, and $\pi(i, k) = k''$, where k'' is the largest integer such that (k'', j) belongs to $R(i, j, k)$.

Let $c_k(i, j)$ denote the value of $c[i, j]$ after all updates $\langle i, j, k' \rangle \in \Sigma_G$ with $k' \leq k$ have been performed by F, and no other updates have been performed on it. We now present the second main theorem of this section.

THEOREM 2.2. *Let δ and π be as defined in Definition 2.2. Then immediately before function* F *performs the update $\langle i,j,k\rangle$ (i.e., before it executes $c[i,j] \leftarrow f(c[i,j],c[i,k],c[k,j],c[k,k])$), the following hold:*

- $c[i,j] = c_{k-1}(i,j)$,
- $c[i,k] = c_{\pi(j,k)}(i,k)$,
- $c[k,j] = c_{\pi(i,k)}(k,j)$,
- $c[k,k] = c_{\delta(i,j,k)}(k,k)$.

Proof. We prove each of the four claims by turn.

$\underline{c[i,j]}$: By Theorem 2.1, for any given $i,j \in [1,n]$ the value of $c[i,j]$ is updated in increasing value of k, hence at the time when update $\langle i,j,k\rangle$ is about to be applied, the state of $c[i,j]$ must equal $c_{k-1}(i,j)$.

$\underline{c[k,k]}$: Assume that either $k \neq i$ or $k \neq j$, and consider the state of $c[k,k]$ when update $\langle i,j,k\rangle$ is about to be applied. Let $S(i,j,k)$ and $S'(i,j,k)$ be as specified in Definition 2.2, and consider the recursive call $\mathrm{F}(X,k_1,k_2)$ with $k \in [k_1,k_2]$ in which $S(i,j,k)$ and $S'(i,j,k)$ are both called during the execution of lines 6 and 7 of the I-GEP code (this call exists as noted in Lemma 2.1). Also, as noted in Lemma 2.1, the aligned subsquare $S(i,j,k)$ (which contains position (k,k) but not (i,j)) will occur either as X_{11} or X_{22}.

If $S(i,j,k)$ occurs as X_{11} when it is invoked in the pseudocode, then by Lemma 2.1 we also know that $k \in [k_1,k_m]$, and $S'(i,j,k)$ will be invoked as X_{12}, X_{21} or X_{22} in the same recursive call. Thus, $c[k,k]$ will have been updated by all $\langle i,j,k'\rangle \in \Sigma_G$ for which $(k',k') \in S(i,j,k)$, before update $\langle i,j,k\rangle$ is applied to $c[i,j]$ in the forward pass. By Definition 2.2 the largest integer k' for which (k',k') belongs to $S(i,j,k)$ is $\delta(i,j,k)$. Hence the value of $c[k,k]$ that is used in update $\langle i,j,k\rangle$ is $c_{\delta(i,j,k)}(k,k)$.

Similarly, if $S(i,j,k)$ occurs as X_{22} when it is invoked in the pseudocode, then $k \in [k_m+1,k_2]$, and $S'(i,j,k)$ will be invoked as X_{11}, X_{12} or X_{21} in the same recursive call. Since the value of k is in the higher half of $[k_1,k_2]$, the update $\langle i,j,k\rangle$ will be performed in the backward pass in line 7, and hence $c[k,k]$ will have been updated by all $\langle i,j,k'\rangle \in \Sigma_G$ with $k' \leq k_2$. As above, by Definition 2.2, $\delta(i,j,k)$ is the largest value of k' for which (k',k') belongs to $S(i,j,k)$, which is k_2, hence the value of $c[k,k]$ that is used in update $\langle i,j,k\rangle$ is $c_{\delta(i,j,k)}(k,k)$.

Finally, if $i=j=k$, we have $c[k,k] = c_{k-1}(i,j) = c_{\delta(i,j,k)}(k,k)$ by definition of $\delta(i,j,k)$.

$\underline{c[i,k] \text{ and } c[k,j]}$: Similar to the proof for $c[k,k]$ but using parts (b) and (c) of Lemma 2.1. ■

I/O Complexity. Let $I(n)$ be an upper bound on the number of I/O operations performed by F on an input of size $n \times n$. It is not difficult to see [6] that

$$I(n) \leq \begin{cases} \mathcal{O}(n + \frac{n^2}{B}) & \text{if } n^2 \leq \gamma M, \\ 8I\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases} \qquad \text{(equation 2.1)}$$

where γ is the largest constant sufficiently small that four $\sqrt{\gamma M} \times \sqrt{\gamma M}$ submatrices fit in the cache. The solution to the recurrence is $I(n) = \mathcal{O}\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ (assuming a tall cache, i.e., $M = \Omega(B^2)$).

In [6] we show that this bound is, in fact, tight for any algorithm that performs $\Theta\left(n^3\right)$ operations in order to implement the general version of the GEP computation as defined in Section 1.1.

Below we prove a more general upper bound on the I/O operations performed by function F, which will be used in Section 7.1 to determine the I/O complexity of the I-GEP implementation of 'Simple-DP'.

The following theorem assumes that F is called on an $n \times n$ input matrix c (i.e., called as $\mathrm{F}(c, 1, n)$), but considers only the cache misses incurred for applying the updates $\langle i, j, k\rangle \in \Sigma_G$ with $c[i, j] \in Q$, where Q is an $m \times m$ submatrix of c. Thus the implicit assumption is that immediately before any such update $\langle i, j, k\rangle$ is applied on $c[i, j]$, each of $c[i, k]$, $c[k, j]$ and $c[k, k]$ has the correct value (as implied by Theorem 2.2) even if it does not belong to Q.

THEOREM 2.3. *Let Q be an $m \times m$ submatrix of c, where c is the $n \times n$ input matrix to* F. *Then the number of cache misses incurred by* F *while applying all updates $\langle i, j, k\rangle \in \Sigma_G$ with $c[i, j] \in Q$ is $\mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}}\right)$, assuming that Q is too large to fit in the cache, and that the cache is tall (i.e., $M = \Omega\left(B^2\right)$).*

Proof. Let Q_e be an aligned subsquare of c of largest width 2^r such that four such subsquares completely fit in the cache, i.e., $\lambda \cdot 4^r \leq M < \lambda \cdot 4^{r+1}$ for some suitable constant λ. We know from Observation 2.1 that Q_e will be invoked in exactly $\frac{n}{2^r}$ recursive calls of F on disjoint aligned subintervals of $[k_1, k_2]$ of length 2^r each. The number of cache misses incurred in fetching Q_e into the cache along with all (at most three) other aligned subsquares required for updating the entries of Q_e is $\mathcal{O}\left(2^r + \frac{2^r \times 2^r}{B}\right)$, since at most 1 cache miss will occur for accessing each row of the subsquares, and $\mathcal{O}\left(\frac{2^r \times 2^r}{B}\right)$ cache misses for scanning in all entries. Thus the total cache misses incurred by all $\frac{n}{2^r}$ recursive calls on Q_e is $\mathcal{O}\left(\frac{n}{2^r} \times \left(2^r + \frac{2^r \times 2^r}{B}\right)\right) = \mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right)$, since $2^r \times 2^r = \Theta\left(M\right)$.

Now since Q is an $m \times m$ subsquare of c, and all recursive calls on an $2^r \times 2^r$ subsquare incur $\mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right)$ cache misses, the number of cache misses incurred by all recursive calls updating the entries of Q is $\Theta\left(\frac{m^2}{2^r \times 2^r}\right) \times \mathcal{O}\left(n + \frac{n\sqrt{M}}{B}\right) = \mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}} + \frac{m^2 n}{M}\right) = \mathcal{O}\left(\frac{m^2 n}{B\sqrt{M}}\right)$ (since $M = \Omega\left(B^2\right)$). ■

Static Pruning of I-GEP. In line 1 of Figure 2, function $\mathrm{F}(X, k_1, k_2)$ performs dynamic pruning of its recursion tree by computing the intersection of $T_{X,[k_1,k_2]}$ (i.e., the set of all updates $\langle i, j, k\rangle$ with $k \in [k_1, k_2]$ that are applicable on the input submatrix X) with Σ_G. However, sometimes it is possible to perform some static pruning during the transformation of G to F, i.e., recursive calls for processing of some quadrants of X in lines 6 and/or 7 of F can be eliminated completely from the code. In Appendix B we describe how this static pruning of F can be performed. This is the version of I-GEP (with static pruning) that we considered in our earlier paper [6].

3 C-GEP: Extension of I-GEP to Full Generality

In order to express mathematical expressions with conditionals in compact form, in this section we will use *Iverson's convention* [13, 14] for denoting values of Boolean expressions. In this convention we use $|\mathcal{E}|$ to denote the value of a Boolean expression $\mathcal{E}$, where $|\mathcal{E}| = 1$ if $\mathcal{E}$ is true and $|\mathcal{E}| = 0$ if $\mathcal{E}$ is false.

3.1 A Closer Look at I-GEP

Recall that $c_k(i, j)$ denotes the value of $c[i, j]$ after all updates $\langle i, j, k'\rangle \in \Sigma_G$ with $k' \leq k$, and no other updates have been applied on $c[i, j]$ by F, where $i, j \in [1, n]$ and $k \in [0, n]$. Let $\hat{c}_k(i, j)$ be the corresponding value for G, i.e., let $\hat{c}_k(i, j)$ be the value of $c[i, j]$ immediately after the k-th iteration of the outer ***for*** loop in G, where $i, j \in [1, n]$ and $k \in [0, n]$.

By inspecting the original GEP code in Figure 1, we observe that if $\langle i, j, k\rangle \in \Sigma_G$,

$$\hat{c}_k(i, j) = f(\hat{c}_{k-1}(i, j), \hat{c}_{k-|j \leq k|}(i, k), \hat{c}_{k-|i \leq k|}(k, j), \hat{c}_{k-|(i<k) \vee (i=k \wedge j \leq k)|}(k, k)) \qquad \text{(equation 3.2)}$$

H(X, k_1, k_2)

(X is a $2^q \times 2^q$ submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Matrices u_0, u_1, v_0 and v_1 are global, and each initialized to c before the initial call to H is made. Similar to F in Figure 2, H assumes the following: (*a*) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$
(*b*) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$

The initial call to H is H$(c, 1, n)$ for an $n \times n$ input matrix c, where n is assumed to be a power of 2.)

1. ***if*** $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return*** $\{T_{X,[k_1,k_2]} = \{\langle i,j,k\rangle | i \in [i_1,i_2] \wedge j \in [j_1,j_2] \wedge k \in [k_1,k_2]\}$, *and* Σ_G *is the set of updates performed by* G *in Figure 1*$\}$

2. ***if*** $k_1 = k_2$ ***then***

3. $\quad i \leftarrow i_1,\ \ j \leftarrow j_1,\ \ k \leftarrow k_1$

4. $\quad c[i,j] \leftarrow f(c[i,j], u_{|j>k|}[i,k], v_{|i>k|}[k,j], u_{|(i>k)\ \vee\ (i=k\ \wedge\ j>k)|}[k,k])$

5. $\quad$ ***if*** $k = \tau_{ij}(j-1)$ ***then*** $u_0[i,j] \leftarrow c[i,j]$ $\{\tau_{ij}(l) = \max_{\ l'} \{l' \mid l' \leq l \wedge \langle i,j,l'\rangle \in \Sigma_G \cup \{\langle i,j,0\rangle\}\}\}$

6. $\quad$ ***if*** $k = \tau_{ij}(j)$ ***then*** $u_1[i,j] \leftarrow c[i,j]$

7. $\quad$ ***if*** $k = \tau_{ij}(i-1)$ ***then*** $v_0[i,j] \leftarrow c[i,j]$

8. $\quad$ ***if*** $k = \tau_{ij}(i)$ ***then*** $v_1[i,j] \leftarrow c[i,j]$

9. ***else*** $\{$*The top-left, top-right, bottom-left and bottom-right quadrants of* X *are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*$\}$

10. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$

11. $\quad H(X_{11}, k_1, k_m),\ H(X_{12}, k_1, k_m),\ H(X_{21}, k_1, k_m),\ H(X_{22}, k_1, k_m)$ $\{$*forward pass*$\}$

12. $\quad H(X_{22}, k_m+1, k_2),\ H(X_{21}, k_m+1, k_2),\ H(X_{12}, k_m+1, k_2),\ H(X_{11}, k_m+1, k_2)$ $\{$*backward pass*$\}$

Figure 6: C-GEP: A cache-oblivious implementation of GEP (i.e., G in Figure 1) that works for all f and Σ_G.

with $\hat{c}_k(i,j) = \hat{c}_{k-1}(i,j)$ if $\langle i,j,k\rangle \notin \Sigma_G$. Similarly using Theorem 2.2, we can describe the updates performed by F (Figure 2) as follows. If $\langle i,j,k\rangle \in \Sigma_G$,

$$c_k(i,j) = f\left(c_{k-1}(i,j), c_{\pi(j,k)}(i,k), c_{\pi(i,k)}(k,j), c_{\delta(i,j,k)}(k,k)\right) \qquad \text{(equation 3.3)}$$

with $c_k(i,j) = c_{k-1}(i,j)$ if $\langle i,j,k\rangle \notin \Sigma_G$.

Though both G and F start with the same input matrix, at some point of computation F and G would supply different input values to f while applying the same update $\langle i,j,k\rangle \in \Sigma_G$, and consequently f will return different output values, because we know from Observation 2.2(a) that for $i, j < k$, $\pi(j,k) > k - |j \leq k|$, $\pi(i,k) > k - |i \leq k|$ and $\delta(i,j,k) > k - |(i < k)\ \vee\ (i = k\ \wedge\ j \leq k)|$. Whether the final output matrix returned by the two algorithms are the same depends on the update function f, the update set Σ_G, and the input values.

As an example, consider a 2×2 input matrix c, and let $\Sigma_G = \{\langle i,j,k\rangle | 1 \leq i,j,k \leq 2\}$. Then G will compute the entries in the following order: $\hat{c}_1(1,1)$, $\hat{c}_1(1,2)$, $\hat{c}_1(2,1)$, $\hat{c}_1(2,2)$, $\hat{c}_2(1,1)$, $\hat{c}_2(1,2)$, $\hat{c}_2(2,1)$, $\hat{c}_2(2,2)$; on the other hand, F will compute in the following order: $c_1(1,1)$, $c_1(1,2)$, $c_1(2,1)$, $c_1(2,2)$, $c_2(2,2)$, $c_2(2,1)$, $c_2(1,2)$, $c_2(1,1)$. Since both G and F use the same input matrix, the first 5 values computed by F will be correct, i.e., $c_1(1,1) = \hat{c}_1(1,1)$, $c_1(1,2) = \hat{c}_1(1,2)$, $c_1(2,1) = \hat{c}_1(2,1)$, $c_1(2,2) = \hat{c}_1(2,2)$ and $c_2(2,2) = \hat{c}_2(2,2)$. However, the next value, i.e., the final value of $c[2,1]$, computed by F is not necessarily correct, since F sets $c_2(2,1) \leftarrow f(c_1(2,1), c_2(2,2), c_1(2,1), c_2(2,2))$, while G sets $\hat{c}_2(2,1) \leftarrow f(\hat{c}_1(2,1), \hat{c}_1(2,2), \hat{c}_1(2,1), \hat{c}_1(2,2))$. For example, if initially $c[1,1] = c[1,2] = c[2,1] = 0$ and $c[2,2] = 1$, and f just returns the sum of its input values, then F will output $c[2,1] = 8$, while G will output $c[2,1] = 2$.

3.2 C-GEP using $4n^2$ Additional Space

Before explaining how I-GEP can be extended to handle arbitrary f and Σ_G, we will define a function τ_{ij} which will play a crucial role in this extension.

DEFINITION 3.1. *For $1 \leq i, j, l \leq n$, we define $\tau_{ij}(l)$ to be the largest integer $l' \leq l$ such that $\langle i, j, l' \rangle \in \Sigma_G$ provided such an update exists, and 0 otherwise. More formally, for all $i, j, l \in [1, n]$, $\tau_{ij}(l) = \max_{l'} \{l' \mid l' \leq l \wedge \langle i, j, l' \rangle \in \Sigma_G \cup \{\langle i, j, 0 \rangle\}\}$.*

The significance of τ of can be explained as follows. We know from Theorem 2.1 that both F and G apply the updates $\langle i, j, k \rangle$ in increasing order of k values. Hence, at any point of time during the execution of F (or G) if $c[i, j]$ is in state $c_l(i, j)$ ($\hat{c}_l(i, j)$, resp.), where $l \neq 0$, then $\langle i, j, \tau_{ij}(l) \rangle$ is the update that has left $c[i, j]$ in this state. We also note the difference between π (defined in Definition 2.2) and τ: we know from Theorem 2.2 that immediately before applying $\langle i, j, k \rangle$ function F finds $c[i, k]$ in state $c_{\pi(j,k)}(i, k)$, and from the definition of τ we know that $\langle i, k, \tau_{ik}(\pi(j, k)) \rangle$ is the update that has left $c[i, k]$ in this state. Similar observation holds for δ defined in Definition 2.2.

We will extend I-GEP to full generality by modifying F in Figure 2 so that the updates performed by it resemble equation 3.2 instead of equation 3.3. As described below, we achieve this by saving suitable intermediate values of the entries of c in auxiliary matrices as F generates them. Note that for all $i, j, k \in [1, n]$, F computes $c_{k-|j \leq k|}(i, k)$, $c_{k-|i \leq k|}(k, j)$ and $c_{k-|(i<k) \vee (i=k \wedge j \leq k)|}(k, k)$ before it computes $c_k(i, j)$ since we know from Observation 2.2 that $\pi(j, k) \geq k - |j \leq k|$, $\pi(i, k) \geq k - |i \leq k|$ and $\delta(i, j, k) \geq k - |(i < k) \vee (i = k \wedge j \leq k)|$ for all $i, j, k \in [1, n]$. However, these values could be overwritten before F needs to use them. In particular, we may lose certain key values as summarized in the observation below which follows from Theorem 2.1 and the definition of τ.

OBSERVATION 3.1. *Immediately before* F *applies the update $\langle i, j, k \rangle \in \Sigma_G$:*

(a) if $\tau_{ik}(\pi(j, k)) > k - |j \leq k|$ then $c[i, k]$ may not necessarily contain $c_{k-|j \leq k|}(i, k)$;

(b) if $\tau_{kj}(\pi(i, k)) > k - |i \leq k|$ then $c[k, j]$ may not necessarily contain $c_{k-|i \leq k|}(i, k)$; and

(c) if $\tau_{kk}(\delta(i, j, k)) > k - |(i < k) \vee (i = k \wedge j \leq k)|$ then $c[k, k]$ may not necessarily contain $c_{k-|(i<k) \vee (i=k \wedge j \leq k)|}(k, k)$.

If the condition in Observation 3.1(a) holds, we must save $c_{k-|j \leq k|}(i, k)$ as soon as it is generated so that it can be used later by $\langle i, j, k \rangle$. However, $c_{k-|j \leq k|}(i, k)$ is not necessarily generated by $\langle i, k, k - |j \leq k| \rangle$ since this update may not exist in Σ_G in the first place. If $\tau_{ij}(k - |j \leq k|) \neq 0$, then $\langle i, k, \tau_{ij}(k - |j \leq k|) \rangle$ is the update that generates $c_{k-|j \leq k|}(i, k)$, and we must save this value after applying this update and before some other update modifies it. If $\tau_{ij}(k - |j \leq k|) = 0$, then $c_{k-|j \leq k|}(i, k) = c_0(i, k)$, i.e., update $\langle i, j, k \rangle$ can use the initial value of $c[i, k]$. A similar argument applies to $c[k, j]$ and $c[k, k]$ as well.

Now in order to identify the intermediate values of each $c[i, j]$ that must be saved, consider the accesses made to $c[i, j]$ when executing the original GEP code in Figure 1.

OBSERVATION 3.2. *The GEP code in Figure 1 accesses each $c[i, j]$:*

*(a) as $c[i, j]$ at most once in each iteration of the outer **for** loop for applying updates $\langle i, j, k \rangle \in \Sigma_G$;*

*(b) as $c[i, k]$ only in the j-th iteration of the outer **for** loop, for applying updates $\langle i, j', j \rangle \in \Sigma_G$ for all $j' \in [1, n]$;*

*(c) as $c[k, j]$ only in the i-th iteration of the outer **for** loop, for applying updates $\langle i', j, i \rangle \in \Sigma_G$ for all $i' \in [1, n]$; and*

(d) if $i = j$, *as $c[k, k]$ in the i-th iteration of the outer **for** loop for applying updates $\langle i', j', i \rangle \in \Sigma_G$ for all $i', j' \in [1, n]$.*

The updates in Observation 3.2(a) do not need to be stored separately, since we know from Theorem 2.1 that both GEP and I-GEP apply the updates on a fixed $c[i, j]$ in exactly the same order.

Now consider the accesses to $c[i, j]$ in parts (b), (c) and (d) of Observation 3.2. By inspecting the code in Figure 1 (see also equation 3.2), we observe that immediately before G applies the update $\langle i, j', j\rangle$ in Observation 3.2(b), $c[i, j] = \hat{c}_{j-1}(i, j) = \hat{c}_{\tau_{ij}(j-1)}(i, j)$ if $j' \leq j$, and $c[i, j] = \hat{c}_j(i, j) = \hat{c}_{\tau_{ij}(j)}(i, j)$ otherwise. Similarly, immediately before applying the update $\langle i', j, i\rangle$ in Observation 3.2(c), $c[i, j] = \hat{c}_{i-1}(i, j) = \hat{c}_{\tau_{ij}(i-1)}(i, j)$ if $i' \leq i$, and $c[i, j] = \hat{c}_i(i, j) = \hat{c}_{\tau_{ij}(i)}(i, j)$ otherwise. When G is about to apply an update $\langle i', j', i\rangle$ from Observation 3.2(d), $c[i, j] = \hat{c}_{i-1}(i, j) = \hat{c}_{\tau_{ij}(i-1)}(i, j)$ if $i' < i \vee (i' = i \wedge j' \leq j)$, and $c[i, j] = \hat{c}_i(i, j) = \hat{c}_{\tau_{ij}(i)}(i, j)$ otherwise.

Therefore, F must be modified to save the value of $c[i, j]$ immediately after applying the update $\langle i, j, k\rangle \in \Sigma_G$ for $k \in \{\tau_{ij}(i-1), \tau_{ij}(i), \tau_{ij}(j-1), \tau_{ij}(j)\}$. Observe that since there are exactly n^2 possible (i, j) pairs, we need to save at most $4n^2$ intermediate values.

In Figure 6 we present the modified version of F, which we call H. The algorithm has exactly the same structure as F, i.e., it accepts the same inputs as F (one square matrix X, and two integers k_1 and k_2) and assumes the same preconditions on inputs, it decomposes the input matrix in exactly the same way, and processes the submatrices in the same order using similar functions as F does. The only difference between F and H is in the way the updates are performed. In line 3, F updates $c[i, j]$ using entries directly from c, i.e., it updates $c[i, j]$ using whatever values $c[i, j]$, $c[i, k]$, $c[k, j]$ and $c[k, k]$ have at the time of the update. In contrast, H uses four $n \times n$ matrices u_0, u_1, v_0 and v_1 for saving appropriate intermediate values computed for the entries of c as discussed above, which it uses for future updates. We assume that each of the tests in lines 5–8 involving τ_{ij} can be performed in constant time without incurring any additional cache misses.

I/O Complexity & Running Time. The number of cache misses incurred by H can be described using the same recurrence relation (equation 2.1) that was used to describe the cache misses incurred by F in Section 2, and hence the I/O complexity remains the same, i.e., $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$. Function H also has the same $\mathcal{O}\left(n^3\right)$ running time as F, since it only incurs a constant overhead per update applied.

Correctness. Since Theorems 2.1 and 2.2 in Section 2 were proved based on the structural properties of F and not on the actual form of the updates, they continue to hold for H.

The correctness of H, i.e., that it correctly implements equation 3.2 and thus G, follows directly from the following lemma, which can be proved by induction on k using Theorems 2.1 and 2.2, and by observing that H saves all required intermediate values in lines 5–8.

LEMMA 3.1. *Immediately before* H *performs the update* $\langle i, j, k\rangle$, *the following hold:* $c[i, j] = \hat{c}_{k-1}(i, j)$, $v_{|j>k|}[i, k] = \hat{c}_{k-|j\leq k|}(i, k)$, $h_{|i>k|}[k, j] = \hat{c}_{k-|i\leq k|}(k, j)$ *and* $v_{|(i>k) \vee (i=k \wedge j>k)|}[k, k]) = \hat{c}_{k-|(i<k) \vee (i=k \wedge j\leq k)|}(k, k)$.

3.3 Reducing the Additional Space

We can reduce the amount of extra space used by H (Figure 6) by observing that at any point during the execution of H we do not need to store more than $n^2 + n$ intermediate values for future use. In fact, we will show that it is sufficient to use four $\frac{n}{2} \times \frac{n}{2}$ matrices and two vectors of length $\frac{n}{2}$ each for storing intermediate values, instead of using four $n \times n$ matrices.

Let $U \equiv u_0[1 \ldots n, 1 \ldots n]$, $\overline{U} \equiv u_1[1 \ldots n, 1 \ldots n]$, $V \equiv v_0[1 \ldots n, 1 \ldots n]$ and $\overline{V} \equiv v_1[1 \ldots n, 1 \ldots n]$. By U_{11}, U_{12}, U_{21} and U_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrants of U, respectively. We identify the quadrants of $\overline{U}$, V and $\overline{V}$ similarly. For $i \in [1, 2]$, let D_i and $\overline{D}_i$ denote the diagonal entries of U_{ii} and $\overline{U}_{ii}$, respectively.

Now consider the initial call to H, i.e., $\mathrm{H}(X, k_1, k_2)$ where $X = c$, $k_1 = 1$ and $k_2 = n$. We show below that the forward pass in line 11 of this call can be implemented using only $n^2 + n$ extra space. A similar argument applies to the backward pass (line 12) as well.

The first recursive call $\mathrm{H}(X_{11}, k_1, k_2)$ in line 11 will generate U_{11}, $\overline{U}_{11}$, V_{11}, $\overline{V}_{11}$, D_1 and $\overline{D}_1$. The amount of extra space used by this recursive call is thus n^2+n. The entries in U_{11} and V_{11}, however, will not be used by any future updates, and hence can be discarded. The second recursive call $\mathrm{H}(X_{12}, k_1, k_2)$ will use $\overline{U}_{11}$, D_1 and $\overline{D}_1$, and generate V_{12} and $\overline{V}_{12}$ in the space freed by discarding U_{11} and V_{11}. Each update $\langle i, j, k\rangle$ applied by this recursive call retrieves $u_{|j>k|}[i, k]$ from $\overline{U}_{11}$, $v_{|i>k|}[k, j]$ from V_{12} or $\overline{V}_{12}$, and $u_{|(i>k) \vee (i=k \wedge j>k)|}[k, k]$ from D_1 or $\overline{D}_1$. Upon return from $\mathrm{H}(X_{12}, k_1, k_2)$ we can discard the entries in $\overline{U}_{11}$ and V_{12} since they will not be required for any future updates. The next recursive call $\mathrm{H}(X_{12}, k_1, k_2)$ will use $\overline{V}_{11}$, D_1 and $\overline{D}_1$, and generate U_{21} and $\overline{U}_{21}$ in the space previously occupied by $\overline{U}_{11}$ and V_{12}. Each update performed by this recursive call retrieves $u_{|j>k|}[i, k]$ from U_{21} or $\overline{U}_{21}$, $v_{|i>k|}[k, j]$ from $\overline{V}_{11}$, and $u_{|(i>k) \vee (i=k \wedge j>k)|}[k, k]$ from D_1 or $\overline{D}_1$. The last function call $\mathrm{H}(X_{22}, k_1, k_2)$ in line 11 will use $\overline{U}_{21}$, $\overline{V}_{12}$, D_1 and $\overline{D}_1$ for updates, and will not generate any intermediate values. Thus line 11 can be implemented using only four additional $\frac{n}{2} \times \frac{n}{2}$ matrices and two vectors of length $\frac{n}{2}$ each.

Therefore, H can be implemented to work with any arbitrary f and arbitrary Σ_G at the expense of only $n^2 + n$ extra space. The running time and the I/O complexity of this implementation remain $\mathcal{O}\left(n^3\right)$ and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$, respectively.

4 Applications of Cache-Oblivious I-GEP

In this section we consider I-GEP for generalized versions of three major GEP applications considered in [6]. Though the C-GEP implementation given in Section 3 works for all instances of f and Σ_G, it uses extra space, and is slightly more complicated than I-GEP. Our experimental results in Section 6 also show that I-GEP performs slightly better than both variants of C-GEP. Hence an I-GEP implementation is preferable to a C-GEP implementation if it can be proved to work correctly for a given GEP instance.

We consider the following applications of I-GEP in this section.

- In Section 4.1 we consider I-GEP for a class of applications that includes Gaussian elimination without pivoting, where we restrict Σ_G but allow f to be unrestricted.
- In Section 4.2 we consider a class of applications where we do not impose any restrictions on Σ_G, but restrict f to receive all its inputs except the first one (i.e., except $c[i, j]$) from matrices that remain unmodified throughout the computation. An important problem in this class is matrix multiplication.
- In Section 4.3 we consider I-GEP for path computations over closed semirings which includes Floyd-Warshall's APSP algorithm [9] and Warshall's algorithm for finding transitive closures [23]. For this class of problems we restrict both f and Σ_G.

At the end of this paper (Section 7) we consider the remaining two applications of I-GEP mentioned in [6].

4.1 Gaussian Elimination without Pivoting

Gaussian elimination without pivoting is used in the solution of systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [5]. We represent a system of $n-1$ equations in $n-1$ unknowns $(x_1, x_2, \ldots, x_{n-1})$ using an $n \times n$ matrix c, where the i'th $(1 \le i < n)$ row represents the equation $a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,n-1}x_{n-1} = b_i$. The method proceeds in

```
G(c, 1, n)
(The input c[1...n, 1...n] is an n × n matrix. Function f(·,·,·,·) is a problem-specific function, and for
Gaussian elimination without pivoting f(x, u, v, w) = x − (u/w) × v.)
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             if (k ≤ n − 2) ∧ (k < i < n) ∧ (k < j) then c[i, j] ← f(c[i, j], c[i, k], c[k, j], c[k, k])
```

Figure 7: A more general form of the first phase of Gaussian elimination without pivoting.

```
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             c[i, j] ← c[i, j] + a[i, k] × b[k, j]
```

(a)

```
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             if ⟨i, j, k⟩ ∈ Σ_G then
                   c[i, j] ← f(c[i, j], a[i, k], b[k, j], d[k, k]) {a, b, d ≠ c}
```

(b)

Figure 8: **(a)** Modified matrix multiplication algorithm, **(b)** A more general form of the algorithm in Figure 8(a).

two phases. In the first phase, an upper triangular matrix is constructed from c by successive elimination of variables from the equations. This phase requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(\frac{n^3}{B})$ I/Os. In the second phase, the values of the unknowns are determined from this matrix by back substitution. It is straight-forward to implement this second phase in $\mathcal{O}(n^2)$ time and $\mathcal{O}\left(\frac{n^2}{B}\right)$ I/Os, so we will concentrate on the first phase.

The first phase is an instantiation of the GEP code in Figure 1. In Figure 7 we give a computation that is a more general form of the computation in the first phase of Gaussian elimination without pivoting in the sense that the update function f in Figure 7 is arbitrary. The ***if*** condition in line 4 ensures that $i > k$ and $j > k$ hold for every update $\langle i, j, k\rangle$ applied on c, i.e., $\Sigma_G = \{\langle i, j, k\rangle : (1 \leq k \leq n-2) \wedge (k < i < n) \wedge (k < j \leq n)\}$.

The correctness of the I-GEP implementation of the code in Figure 7 can be proved by induction on k using Theorem 2.2 and by observing that each $c[i, j]$ $(1 \leq i, j \leq n)$ settles down (i.e., is never modified again) before it is ever used on the right hand side of an update.

As described in [6] and also in Appendix B, we can apply static pruning on the resulting I-GEP implementation to remove unnecessary recursive calls from the pseudocode.

A similar method solves LU decomposition without pivoting within the same bounds. Both algorithms are in-place. Our algorithm for Gaussian elimination which originally appeared in our earlier paper [6], is arguably simpler than existing algorithms since it does not use LU decomposition as an intermediate step, and thus does not invoke subroutines for multiplying matrices or solving triangular linear systems, as is the case with other cache-oblivious algorithms for this problem [25, 3, 20].

4.2 Matrix Multiplication

We consider the problem of computing $C = A \times B$, where A, B and C are $n \times n$ matrices. Though standard matrix multiplication does not fall into GEP, it does after the small structural modification

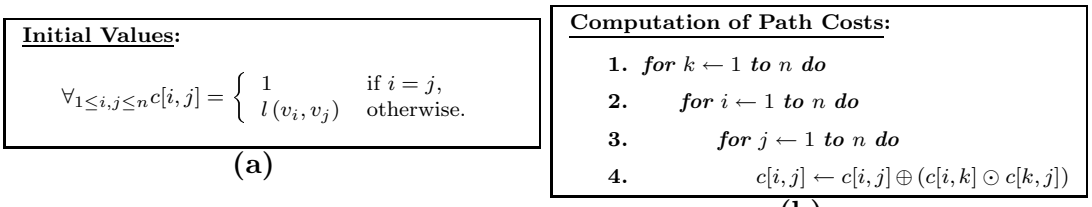

Figure 9: Computation of path costs over a closed semiring $(S, \oplus, \odot, 0, 1)$: **(a)** Initialization of c, **(b)** Computation of path costs.

shown in Figure 8(a) (index k is in the outermost loop in the modified algorithm, while in the standard algorithm it is in the innermost loop); correctness of this transformed code is straight-forward. In [6] we considered this transformed code (in Figure 8(a)) for I-GEP implementation.

The algorithm in Figure 8(b) generalizes the computation in step 4 of Figure 8(a) to update $c[i, j]$ to a new value that is an arbitrary function of $c[i, j], a[i, k], b[k, j]$ and $d[k, k]$, where matrices $a, b, d \neq c$.

The correctness of the I-GEP implementation of the code in Figure 8(b) follows from Theorem 2.1 and from the observation that matrices a, b and d remain unchanged throughout the entire computation.

4.3 Path Computations Over a Closed Semiring

An algebraic structure known as a *closed semiring* [2] serves as a general framework for solving path problems in directed graphs. In [2], an algorithm is given for finding the set of all paths between each pair of vertices in a directed graph. Both Floyd-Warshall's algorithm for finding all-pairs shortest paths [9] and Warshall's algorithm for finding transitive closures [23] are instantiations of this algorithm.

Consider a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. If $(v_i, v_j) \notin E$, $l(v_i, v_j)$ is assumed to have a value 0. The *path-cost* of a path is defined as the product $(\odot)$ of the labels of the edges in the path, taken in order. The path-cost of a zero length path is 1. For each pair $v_i, v_j \in V$, $c[i, j]$ is defined to be the sum of the path-costs of all paths going from v_i to v_j. By convention, the sum over an empty set of paths is 0. Even if there are infinitely many paths between v_i and v_j (due to presense of cycles), $c[i, j]$ will still be well-defined due to the properties of a closed semiring.

The algorithm given in Figure 9(b), which is an instance of GEP, computes $c[i, j]$ for all pairs of vertices $v_i, v_j \in V$. This algorithm performs $\mathcal{O}\left(n^3\right)$ operations and uses $\mathcal{O}\left(n^2\right)$ space. In [6] we considered Floyd-Warshall's APSP which is a specialization of the algorithm in Figure 9(b) in that it performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

Correctness of I-GEP Implementation of Figure 9(b). Recall that $c_0(i, j)$ is the initial value of $c[i, j]$ received by the I-GEP function F in Figure 2, and $c_k(i, j)$ $(1 \le i, j \le n)$ denotes the value of $c[i, j]$ after all updates $\langle i, j, k' \rangle \in \Sigma_G$ with $k' \le k$, and no other updates have been performed on it by F.

For $i, j \in [1, n]$ and $k \in [0, n]$, let $P^k_{i,j}$ denote the set of all paths from v_i to v_j with no intermediate vertex higher than v_k, and let $Q^k_{i,j}$ be the set of all paths from v_i to v_j that have contributed to the computation of $c_k(i, j)$.

The correctness of the I-GEP implementation of the code in Figure 9(b) follows from the following lemma, which can be proved by induction on k using Theorems 2.1 and 2.2.

LEMMA 4.1. *For all* $i, j, k \in [1, n]$, $Q^k_{i,j} \supseteq P^k_{i,j}$.

Since for $i, j \in [1, n]$, $P^n_{i,j}$ contains all paths from v_i to v_j, we have $Q^n_{i,j} \subseteq P^n_{i,j}$, which when combined with $Q^n_{i,j} \supseteq P^n_{i,j}$ obtained from lemma 4.1, results in $Q^n_{i,j} = P^n_{i,j}$.

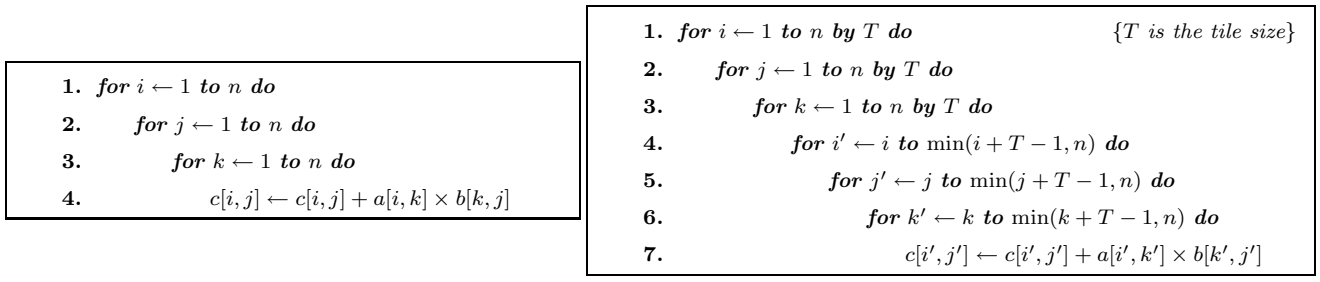

Figure 10: **(a)** Traditional matrix multiplication algorithm, **(b)** Tiled version of the matrix multiplication algorithm of part (a) [16].

5 Cache-Oblivious GEP and Compiler Optimization

'Tiling' is a powerful loop transformation technique employed by optimizing compilers for improving temporal locality in nested loops [16]. This transformation partitions the iteration-space of nested loops into a series of small polyhedral areas of a given *tile size* which are executed one after the other. Tiling a single loop replaces it by a pair of loops, and if the tile size is T then the inner loop iterates T times, and the outer loop has an increment equal to T (assuming that the original loop had unit increments). This transformation can be applied to arbitrarily deep nested loops. Figure 10(b) shows a tiled version of the triply nested loop shown in Figure 10(a) that occurs in matrix multiplication [16].

Cache performance of a tiled loop depends on the chosen tile size T. Choice of T, in turn, crucially depends on (1) the type of the cache (direct mapped or set associative), (2) cache size, (3) block transfer size (i.e., cache line size), and (4) the loop bounds [16, 26]. Thus tiling is a highly system-dependent technique. Moreover, since only a single tile size is chosen, tiling cannot be optimized for all levels of a memory hierarchy simultaneously.

The I-GEP code in Figure 2 and the C-GEP code given in Figure 6 can be viewed as cache-oblivious versions of tiling for the triply nested loops of the form as shown in Figure 1. The nested loop in Figure 1 has an $n \times n \times n$ iteration-space. Both I-GEP and C-GEP are initially invoked on this $n \times n \times n$ cube, and at each stage of recursion they partition the input cube into 8 equal-sized subcubes, and recursively process each subcube. Hence, at some stage of recursion, they are guaranteed to generate subcubes of size $T' \times T' \times T'$ such that $\frac{T}{2} < T' \leq T$, where T is the optimal tile size for any given level of the memory hierarchy. Thus for each level of the memory hierarchy both I-GEP and C-GEP cache-obliviously choose a tile size that is within a constant factor of the optimal tile size for that level. We can, therefore, use I-GEP and C-GEP as cache-oblivious loop transformations for the memory hierarchy.

C-GEP. C-GEP is a legal transformation for any nested loop that conforms to the GEP format given in Figure 1. In order to apply this transformation the compiler must be able to evaluate $\tau_{ij}(i-1)$, $\tau_{ij}(i)$, $\tau_{ij}(j-1)$ and $\tau_{ij}(j)$ for all $i, j \in [1, n]$. For most practical problems this is straight-forward; for example, when $\Sigma_G = \{\langle i, j, k\rangle \mid i, j, k \in [1, n]\}$ which occurs in path computations over closed semirings (see Section 4.3), or even if the computation is not over a closed semiring, we have $\tau_{ij}(l) = l$ for all $i, j, l \in [1, n]$.

I-GEP. Though C-GEP is always a legal transformation for GEP loops, I-GEP is not. Due to the space overhead of C-GEP, I-GEP should be the transformation of choice wherever it is applicable. Moreover, experimental results (see Section 6) suggest that I-GEP outperforms C-GEP in both in-core and out-of-core computations.

We discuss below several conditions based on which the compiler can decide whether I-GEP is a legal transformation for a given GEP code.

First consider the special version of the GEP code presented in our earlier paper [6] (in Figure

6 of [6]) that occurs frequently in practice. In this version the outermost ***for*** loop for k runs from κ_1 to κ_2, the middle ***for*** loop for i runs from $\iota_1(k)$ to $\iota_2(k)$, and the innermost loop for j runs from $\zeta_1(k,i)$ to $\zeta_2(k,i)$, where κ_1 and κ_2 are problem-specific constants, and $\iota_1(\cdot)$, $\iota_2(\cdot)$, $\zeta_1(\cdot,\cdot)$ and $\zeta_2(\cdot,\cdot)$ are problem-specific functions. This version can be obtained from GEP in Figure 1 by setting $\Sigma_G = \{\ \langle i,j,k\rangle\ \mid k \in [\kappa_1,\ \kappa_2]\ \wedge\ i \in [\iota_1(k),\ \iota_2(k)]\ \wedge\ j \in [\zeta_1(k,i),\ \zeta_2(k,i)]\ \}$.

Based on the applications considered in Section 4, it is straight-forward to verify that I-GEP is a legal transformation for the GEP version described above (and given in Figure 6 of [6]) under the following conditions:

(*i*) for all $k \in [\kappa_1,\kappa_2]$, $\iota_1(k) > k$ hold, and for all $k \in [\kappa_1,\kappa_2]$ and $i \in [\iota_1(k),\iota_2(k)]$, $\zeta_1(i,k) > k$ hold (example: Gaussian elimination without pivoting in Section 4.1);

(*ii*) update $\langle i,j,k\rangle$ is of the form $c[i,j] \leftarrow f(c[i,j],a[i,k],b[k,j],d[k,k])$, where $a,b,d \neq c$ (example: matrix multiplication in Section 4.2);

(*iii*) all three loop indices (i,j,k) range from 1 to n, and update $\langle i,j,k\rangle$ is of the form $c[i_1,j_1] \leftarrow c[i_1,j_1] \oplus (c[i_1,k_1] \odot c[k_1,j_1])$, where the computation is over a closed semiring $(S,\oplus,\odot,0,1)$ (example: Floyd-Warshall's APSP in Section 4.3).

Assuming that $\zeta_1(i,k)$ is a nondecreasing function of i, the conditions in part (*i*) of the observation above are equivalent to testing that neither $\iota_1(k) \leq k$ nor $\zeta_1(\iota_1(k),k) \leq k$ has a solution in $[\kappa_1,\kappa_2]$. This verification requires time independent of n. Parts (*ii*) and (*iii*) can also be verified in constant time.

Now consider the general GEP code in Figure 1. Recall the definition of π from Section 2 (and Appendix A), and the definition of τ_{ij} from Section 3.2 (Definition 3.1). The following lemma follows from Observations 3.1 and 3.2 in Section 3.2, and also from the observation that I-GEP will correctly implement GEP if for each $c[i,j]$ and each update in Σ_G that uses $c[i,j]$ on the right hand side, $c[i,j]$ retains the correct value needed for that update until I-GEP applies the update.

LEMMA 5.1. *If $\tau_{ij}(\pi(k,i)) \leq i - |k \leq i|$ for all $\langle i,k,j\rangle \in \Sigma_G$, and $\tau_{ij}(\pi(k,j)) \leq j - |k \leq j|$ for all $\langle k,j,i\rangle \in \Sigma_G$, then I-GEP is a legal transformation for the GEP code in Figure 1.*

6 Experimental Results

We implemented I-GEP (Section 1.1) and both $4n^2$-space and (n^2+n)-space variants of C-GEP (Section 3). We compared these three implementations with the original GEP implementation (Figure 1) for both *in-core* (i.e., when all data completely fit in internal memory) and *out-of-core* (i.e., when only portions of the data fit in internal memory) computations. We used $\Sigma_G = \{\ \langle i,j,k\rangle \mid i,j,k \in [1,n]\ \}$, and updates of the form $c[i,j] \leftarrow \min(c[i,j], c[i,k] + c[k,j])$ which occur in Floyd-Warshall's APSP algorithm [9, 23].

For in-core computations we obtained timing and caching data on two state-of-the-art architectures: Intel Xeon and SUN UltraSPARC-III+. We ran our out-of-core experiments on Intel Xeon machines each equipped with a fast (4.5 ms average seek time) hard disk, and used the STXXL software library [7, 8] for external memory accesses. Here is a summary of the results we obtained:

- **GEP vs. I-GEP/C-GEP.**
  - In-Core Running Time: I-GEP and both variants of C-GEP ran upto 1.7 times faster than GEP even when the entire input fit in internal memory.

- L2 Cache Misses: GEP incurred upto 100 times more L2 cache misses than I-GEP and both variants of C-GEP.
- Out-of-Core I/O Wait Time: When the internal memory available to the algorithms was restricted to half the size of the input matrix, GEP spent upto 500 times more time waiting for I/Os than I-GEP, and upto 180 times more than either variant of C-GEP.

The relative performance of I-GEP/C-GEP with respect to GEP improved as the size of the input increased.

- **I-GEP vs. C-GEP.** In general, in all of our experiments, I-GEP performed the best, followed by the $4n^2$-space variant of C-GEP, which, in turn, performed slightly better than its (n^2+n)-space variant. I-GEP and both variants of C-GEP ran faster than GEP when $n \geq 2^{11}$.

We describe our experiments in more details below.

6.1 In-Core Computation. We ran all implementations on $n \times n$ matrices initialized with random floating point values (4 bytes each), for $n = 2^t$, $t \in [8, 13]$.

6.1.1 Computing Environment. The experiments were run on the following two architectures:

- **Intel Xeon.** A dual processor 3.06 GHz Intel Xeon shared memory machine with 4 GB of RAM and running Ubuntu Linux 5.10 "Breezy Badger". Each processor had an 8 KB L1 data cache (4-way set associative) and an on-chip 512 KB unified L2 cache (8-way). The block size was 64 bytes for both caches.
- **SUN Blade.** A 1 GHz Sun Blade 2000/1000 (UltraSPARC-III+) with 1 GB of RAM and running SunOS 5.9. The processor had an on-chip 64 KB L1 data cache (4-way) and an off-chip 8 MB L2 cache (2-way). The block sizes were 32 bytes for the L1 cache and 512 bytes for the L2 cache.

We used the *Cachegrind* profiler [19] for simulating cache effects on Intel Xeon. The caching data on the Sun Blade was obtained using the *cputrack* utility that keeps track of hardware counters. All algorithms were implemented in C using a uniform programming style and compiled using *gcc* 3.3.4 with optimization parameter -O3. Each machine was exclusively used for experiments (i.e., no other programs were running on them), and on multi-processor machines only a single processor was used.

6.1.2 Implementation Details. We applied the following additional optimizations on our implementations:

- In both I-GEP and C-GEP, in order to reduce the overhead of recursion we solve the problem directly using GEP once the input submatrix X received by the recursive functions becomes very small. We call the size of the input submatrix at which we switch back to GEP the *base-size*. For each implementation, the best value of *base-size*, i.e., for which the implementation ran the fastest, was determined empirically on each machine. On Sun Blade we chose 16×16 as the *base-size*, while on Intel Xeon it was 128×128 for all three implementations of I-GEP and C-GEP.

  The above optimization improves running time at the expense of non-optimal I/O complexity for *base-size*. An alternative to this approach would be to unroll the recursion once the computation reaches matrices of size *base-size* $\times$ *base-size*, and apply the sequence of updates obtained from the unrolled recursion in a loop. This approach maintains optimal I/O complexity for *base-size*. However, depending on how unrolling is implemented upto $\Theta\left(\textit{base-size}^3\right)$ space is required for storing the unrolled update sequence, and thus it is suitable only when the unrolled sequence is short enough to fit in the L1 cache.

n	GEP	I-GEP		C-GEP ($4n^2$ space)		C-GEP (n^2+n space)	
	L2 misses (c_0)	L2 misses (c_1)	ratio (c_0/c_1)	L2 misses (c_2)	ratio (c_0/c_2)	L2 misses (c_3)	ratio (c_0/c_3)
256	11	11	1.00	29	0.38	16	0.69
512	45	56	0.80	155	0.29	76	0.59
1,024	3,603	404	8.92	1,065	3.38	749	4.81
2,048	475,802	3,888	122.38	5,734	82.98	7,321	64.99
4,096	4,324,782	32,075	134.83	42,272	102.31	45,544	94.96
8,192	34,593,080	249,236	138.80	334,306	103.48	352,581	98.11

Table 1: L2 cache misses ($\times 10^3$) on **Sun Blade 2000/1000** with an L2 cache of size 8 MB and 512 B blocks. The figures are averages of 3 runs on random inputs.

n	GEP	I-GEP		C-GEP ($4n^2$ space)		C-GEP (n^2+n space)	
	runtime (t_0)	runtime (t_1)	ratio (t_0/t_1)	runtime (t_2)	ratio (t_0/t_2)	runtime (t_3)	ratio (t_0/t_3)
256	0.34	0.29	1.17	0.31	1.10	0.31	1.10
512	2.71	2.25	1.20	2.32	1.17	2.36	1.15
1,024	22.09	17.69	1.25	18.11	1.22	18.00	1.23
2,048	229.37	140.50	1.63	141.20	1.62	141.22	1.62
4,096	1,893.89	1,118.22	1.69	1,115.76	1.70	1,113.34	1.70
8,192	15,352.28	8,937.21	1.72	9,047.20	1.70	8,851.79	1.73

Table 2: Running time (in seconds) on **Sun Blade 2000/1000**. The figures are averages of 3 runs on random inputs.

n	GEP	I-GEP		C-GEP ($4n^2$ space)		C-GEP (n^2+n space)	
	L2 misses (c_0)	L2 misses (c_1)	ratio (c_0/c_1)	L2 misses (c_2)	ratio (c_0/c_2)	L2 misses (c_3)	ratio (c_0/c_3)
256	11	21	0.52	97	0.11	72	0.15
512	8,450	167	50.60	456	18.53	484	17.46
1,024	67,498	1,286	52.49	2,413	27.97	2,602	25.94
2,048	537,912	10,157	52.96	14,584	36.88	15,202	35.38
4,096	4,299,160	80,857	53.17	98,354	43.71	99,963	43.01

Table 3: L2 cache misses ($\times 10^3$) on **Intel Xeon** with an L2 cache of size 512 KB and 64 B blocks.

n	GEP	I-GEP		C-GEP ($4n^2$ space)		C-GEP (n^2+n space)	
	runtime (t_0)	runtime (t_1)	ratio (t_0/t_1)	runtime (t_2)	ratio (t_0/t_2)	runtime (t_3)	ratio (t_0/t_3)
256	0.08	0.08	1.00	0.14	0.57	0.15	0.53
512	0.72	0.61	1.18	0.88	0.82	0.98	0.73
1,024	5.38	4.74	1.14	5.93	0.91	6.22	0.86
2,048	43.46	37.82	1.15	41.85	1.04	43.26	1.00
4,096	350.06	291.46	1.20	316.18	1.11	324.12	1.08
8,192	2,766.64	2,314.28	1.20	2,436.62	1.14	2,459.65	1.12

Table 4: Running time (in seconds) on **Intel Xeon**. The figures are averages of 3 runs on random inputs.

- Once the computation reached *base-size*, we copied all necessary *base-size* $\times$ *base-size* blocks from the input matrix to static matrices of size *base-size*$\times$ *base-size*, and performed the GEP computation on these smaller matrices. This optimization reduces cache-conflicts in set-associative caches by always using the same non-conflicting blocks of memory for computation. Another approach to reduce these cache-conflicts (which we did not use) would be to use matrices of size $n \times (n + c)$ to store $n \times n$ input matrices, where c is a prime number. This latter approach reduces the chances of mapping two different blocks (starting at different rows) of the input matrix to the same set in the cache by making the distance between them not be a power of 2.
- We used a linear array to store the input matrix, and rearranged the entries of the matrix so that all entries of each *base-size* $\times$ *base-size* block occupy consecutive locations in the array. This optimization reduces the number of I/Os for reading off any *base-size*$\times$ *base-size* block from the input matrix to $\mathcal{O}\left(1 + \frac{\textit{base-size}\times\textit{base-size}}{B}\right)$ down from $\mathcal{O}\left(\textit{base-size} + \frac{\textit{base-size}\times\textit{base-size}}{B}\right)$. It also increases the effectiveness of the prefetcher.
- We optimized the kernel (lines 3–8) of C-GEP for *base-size*, so that very little unnecessary computation is performed while solving it. For example, if $j_2 < k_1$ holds for some specific base case computation, we know that v_0 will never be accessed in line 4 during that computation, and so the 3rd parameter of f can be fixed to $v_1[k, j]$.

6.1.3 Results. We summarize our results below.

- **L2 Misses.** In Tables 1 and 3 we tabulate the L2 cache misses incurred by all implementations on SUN Blade and Intel Xeon, respectively. On SUN Blade we obtained caching data for n upto 2^{13}. However, since the Cachegrind profiler slows down the running program considerably, we did not obtain caching data for n larger than 2^{12} on Intel Xeon. On both machines, I-GEP and both implementations of C-GEP make better use of the L2 cache than GEP, and as the input size grows they utilize the cache even better. On SUN Blade, for $n = 2^{13}$, GEP incurs about 139 times more L2 misses than I-GEP , and about 103 and 98 times more than $4n^2$-space and $n^2 + n$-space implementations of C-GEP, respectively. On Intel Xeon, GEP incurs about 53 times more L2 misses compared to I-GEP, and about 43 times more than any variant of C-GEP. As expected, both versions of C-GEP incur more L2 misses compared to I-GEP, since they use more space. However, on both machines, the (n^2+n)-space variant of C-GEP incurred slightly more cache misses than the $4n^2$-space variant. We think this happens because blocks from a smaller memory segment collide more frequently in the set associative cache compared to blocks from a larger segment.
- **Running Times.** We tabulate the running times of all implementations on SUN Blade and Intel Xeon in Tables 2 and 4, respectively. On SUN Blade, I-GEP and both variants of C-GEP always run faster than GEP; the larger the input the faster they become compared to GEP. For large inputs all these three implementations run almost at the same speed, and for $n = 2^{13}$, all of them run at least 1.70 times faster than GEP. For $n = 2^{13}$, however, the $4n^2$-space variant of C-GEP overflows the internal memory, and the OS starts using the swap space (1 GB) on disk, which explains why this variant does not speed-up as expected for moving from $n = 2^{12}$ to $n = 2^{13}$.

  On Intel Xeon, I-GEP always runs faster than GEP, while both variants of C-GEP catch-up with GEP once n reaches 2^{11}. After that point on, the larger the input becomes the faster they become compared to GEP. However, I-GEP runs the fastest on Xeon, followed by the $4n^2$-space C-GEP, which is, in turn, slightly faster than the $(n^2 + n)$-space C-GEP. For $n = 2^{13}$, I-GEP runs 1.20 times faster than GEP, while $4n^2$-space and $(n^2 + n)$-space C-GEP run 1.14 and 1.12 times faster than GEP, respectively.

I/O Wait Time in Seconds
(Computation Time in Seconds)

(as a function of internal memory size M when $M \leq$ input size $8n^2$)

	M = input size = $8n^2$ bytes			$M = \frac{1}{2} \times$ input size = $4n^2$ bytes			$M = \frac{1}{4} \times$ input size = $2n^2$ bytes		
n	1,024	2,048	4,096	1,024	2,048	4,096	1,024	2,048	4,096
GEP	0 (180)	0 (1,475)	0 (12,140)	945 (193)	7,527 (1,605)	60,305 (12,781)	931 (193)	7,500 (1,563)	60,359 (12,310)
I-GEP	5 (254)	20 (1,991)	77 (16,866)	8 (259)	30 (2,090)	121 (17,228)	14 (258)	45 (2,211)	176 (17,398)
C-GEP ($4n^2$)	17 (282)	67 (2,216)	278 (18,640)	21 (285)	79 (2,230)	321 (18,136)	29 (283)	98 (2,253)	398 (17,952)
C-GEP ($n^2 + n$)	14 (308)	50 (2,312)	195 (18,690)	21 (300)	80 (2,426)	319 (18,568)	30 (290)	110 (2,493)	439 (18,695)

Table 5: I/O wait time (in seconds) on Intel Xeon equipped with a 73.5 GB Fujitsu MAP3735NC hard disk when internal memory (M) available to the algorithm is varied using STXXL. The corresponding computation times (also in seconds) are given within parentheses. *The total running time is the sum of computation time and I/O wait time.* Figures are averages of three independent runs on matrices initialized with random double-precision floats.

The speed-ups obtained from using I-GEP and C-GEP on Intel Xeon are not as dramatic as those obtained on SUN Blade possibly because the Xeon machine has faster caches and RAM than the SUN machine, and also more effective prefetchers.

6.2 Out-of-Core Computation. We ran all implementations on $n \times n$ matrices initialized with random double precision floating point values (8 bytes each), for $n = 2^t$, $t \in [10, 12]$. We varied the amount of internal memory available to each algorithm and measured the I/O wait time (i.e., time spent waiting for an I/O operation to complete) in each case.

6.2.1 Computing Environment. We ran our experiments on dual processor (we used only 1 processor) 3.06 GHz Intel Xeon shared memory machines with 4 GB of RAM, and running Ubuntu Linux 5.10 "Breezy Badger". Each machine is connected to a 73.5 GB 10K RPM Fujitsu MAP3735NC hard disk. Each hard disk has an 8 MB data buffer. The average seek time for reads and writes are 4.5 and 5.0 ms, respectively. The maximum data transfer rate (to/from media) is 106.9 MB/s.

We implemented all algorithms in C++, and compiled using the g++ 3.3.4 compiler with optimization level -O3 and STXXL library version 0.9. The STXXL library [7, 8] is an implementation of the C++ standard template library STL for external memory computations, and is used mainly for experimentation with huge data sets. The STXXL library maintains its own fully associative cache in RAM with pages from the disk. We compiled STXXL with DIRECT-I/O turned on, which ensures that the OS does not cache the data read from or written to the hard disk.

6.2.2 Implementation Details. We applied all optimizations to our out-of-core implementations as we did for the in-core versions.

For each implementation we allocated a single vector of size N, where N is the total space used by the implementation, and allocated all matrices from that vector. This was done in order to restrict the amount of internal memory used by the implementation by configuring the STXXL parameters of the vector. The STXXL vectors are organized as a collection of blocks of size *BlkSize_* which reside in external memory. Accesses to these blocks are organized through a fully associative cache that consists of *Pages_* pages containing *PgSz_* blocks each. The internal memory consumption of a vector is thus $Pages\_ \times PgSz\_ \times BlkSize\_$. In our implementations, we set *BlkSize_* to 64 KB (a practical disk block size), and *PgSz_* to 1 (since we are accessing a single disk). We accept *Pages_* as a parameter which can be used to vary the internal memory available to the single large vector we allocated in the program, and thus also fix the amount of RAM available to the program itself. We have also configured the STXXL vector to use LRU as the paging strategy.

6.2.3 Results. We tabulate our results in Table 5. We ran GEP, I-GEP and both variants of C-GEP for $n = 2^t$, $t \in [10, 12]$. We performed three sets of experiments. In the first set we set *Pages_* so that the internal memory available to the program is exactly the same as the size of the input, that is, $n^2 \times sizeof(double) = 8n^2$ bytes. For the second set of experiments M was set to $\frac{1}{2} \times$ input size $= 4n^2$ bytes, and for the third set to $\frac{1}{4} \times$ input size $= 2n^2$ bytes. For each set we tabulate the amount of time spent by each implementation waiting for the I/O operations to complete, i.e., without doing any actual computation. For each case, we also list (within parentheses) the amount of time spent by the implementation performing actual computation. The total running time of the implementation is the sum of these two.

We make the following observations from Table 5.

- When $M =$ input size, GEP performs the entire computation in-core, and thus there is no I/O wait time. However, both I-GEP and C-GEP use extra space for handling the base case, and C-GEP uses extra space for saving intermediate values. Therefore, they cannot perform the entire computation in-core, which explains their non-zero I/O wait times.
- As M is reduced to $\frac{\text{input size}}{2}$, I/O wait time of GEP increases dramatically. For $n = 2^{10}$, GEP spends about 118 times more time than I-GEP waiting for I/Os, and about 43 times more than either variant of C-GEP. As n increases these ratios also increase, reaching 500 and 188, respectively, for $n = 2^{12}$.
- As M is reduced from $\frac{\text{input size}}{2}$ to $\frac{\text{input size}}{4}$, the I/O wait times of GEP do not change much, which is expected since the I/O-complexity of GEP $\left(\mathcal{O}\left(\frac{n^3}{B}\right)\right)$ is independent of M. However, I/O wait times of both I-GEP and C-GEP increase roughly by a factor of $\sqrt{2} \approx 1.4$, which is also expected since the I/O complexities $\left(\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)\right)$ of both of these implementations have a $\sqrt{M}$ factor in the denominator.
- The computation time of each implementation (GEP, I-GEP or C-GEP) is much larger than the running time of the corresponding in-core C implementation discussed in Section 6.1. This is due to the overhead inherent in the STXXL implementation, which inflates the time needed for each access to the STXXL vector. However, STXXL does not inflate the I/O wait times since these are the times when the program is sitting idle (i.e., not doing any computation) for an I/O operation to complete. We also observe that unlike the in-core computations in Section 6.1, the computation times of I-GEP and C-GEP are larger than that of GEP. We think this happens because the overhead in STXXL implementation makes the time required for computing the kernel (i.e., line 3

of I-GEP in Figure 2, and lines 3–8 of C-GEP in Figure 6) larger than the L2 cache latency, and thus the running time is dominated by the cost of computing the kernel, rather than the cost of accessing the caches. However, in spite of this inflation in computation time, we note that the I/O wait time dominates the total running time of GEP.

7 Additional Applications of Cache-Oblivious I-GEP

In Section 4 we considered generalizations of three major applications of I-GEP from [6]. In this section we consider in more detail the remaining two applications of I-GEP briefly mentioned in [6]:

- In Section 7.1 we consider a class of dynamic programs called 'simple DP' [4] that includes important problems such as RNA secondary structure prediction, matrix chain multiplication and construction of optimal binary search trees. In [6] we briefly outlined how simple DP can be decomposed into a sequence of I-GEP instances using a decomposition technique from [12]. In Section 7.1 we provide a detailed description of this decomposition, its correctness and I/O bounds.
- In [6] we studied the gap problem (i.e., sequence alignment with gaps), and presented the I-GEP implementation of a GEP-like code derived from the classical gap dynamic program. However, that result is not quite correct. In Section 7.2 we study the relationship of the classical gap dynamic program to another GEP-like form and its associated I-GEP; we show that this GEP-like form and its I-GEP correctly solve two special cases of the gap problem, and a small variant of this I-GEP solves gap in its full generality.

7.1 Simple Dynamic Programs

In [4], the term *simple dynamic program* was used to denote a class of dynamic programming problems over a nonassociative semi-ring $(S,\ \min,\ +,\ \infty)$[1] which can be solved in $\mathcal{O}\left(n^3\right)$ time using the dynamic program shown in Figure 11. Its applications include RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. An $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [21] was given in [4] for this class of problems.

In this section, we consider a more general version of simple DP, which is called the *parenthesis problem* in [12], and is described as follows (the generalization comes from the additional term $w(i,k,j)$):

$$c[i,j] = \begin{cases} x_j & \text{if } 0 \le i = j-1 < n, \\ \min_{i<k<j}\left\{c[i,k] + c[k,j] + w(i,k,j)\right\} & \text{if } 0 \le i < j-1 < n; \end{cases} \qquad \text{(equation 7.4)}$$

where x_j's are assumed to be given for $j \in [1,n]$. We also assume that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in-core without incurring any cache misses.

We describe below a method that transforms the dynamic program given in equation 7.4 to a sequence of dynamic programs in GEP. In this method the upper triangular matrix c is decomposed into (forward) diagonal strips of horizontal width $n^{\frac{1}{4}}$, and the entries in c are computed one strip at a time starting from the largest (leftmost) strip. The computation for each strip involves min-plus matrix multiplication and dynamic programs that can be solved with cache-oblivious I-GEP. The resulting algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os. Unlike I-GEP, however, this algorithm uses a modest amount $\left(\mathcal{O}\left(n^{1.75}\right) = o\left(n^2\right)\right)$ of extra space. This method is based on a parallel algorithm for the parenthesis problem given in [12]. There are, however, two major differences between the transformation described here and the algorithm in [12].

[1] In a nonassociative semi-ring min is an associative, commutative and idempotent binary operator; + is a nonassociative and noncommutative binary operator; ∞ is the identity for min and annihilator for +; and the operators distribute over each other.

```
1. for i ← 0 to n − 1 do c[i, i + 1] ← x_{i+1}
2. for d ← 2 to n do
3.     for i ← 0 to n − d do
4.         j ← i + d,  c[i, j] ← ∞
5.         for k ← i + 1 to j − 1 do
6.             c[i, j] ← min{c[i, j], c[i, k] + c[k, j]}
```

Figure 11: The $\mathcal{O}\left(n^3\right)$ time simple DP algorithm.

(i) We reorder the execution of some of the steps in the algorithm (without affecting its correctness) for better space utilization.

(ii) We reduce computations involving 4-dimensional arrays to computations on 2-dimensional arrays (see step 2.1) so that GEP can be applied (since GEP works on a 2-dimensional input matrix).

We now describe the transformation. The recurrence relation in equation 7.4 can be viewed as computing a binary tree of minimum weight [22] in which

(a) each vertex is given a unique label (i, j), $0 \leq i < j \leq n$,

(b) leaves are labeled $(i, i+1)$, $i \in [0, n-1]$ in order from left to right with x_{i+1} being the weight of leaf $(i, i+1)$, and

(c) each internal node (i, j) has weight $w(i, k, j)$, left child (i, k) and right child (k, j) for some $k \in [i+1, j-1]$, and its descendant leaves are labeled $(i', i'+1)$ for $i \leq i' < j$.

For each (i, j) with $0 \leq i < j \leq n$, the dynamic program given by equation 7.4 computes in $c[i, j]$ the cost $g(i, j)$ of the optimal (i.e., minimum-weight) binary tree rooted at (i, j). A *partial tree* T is defined to be a tree rooted at some vertex (i, j) with the subtree rooted at one of its non-leaf nodes (r, s) deleted. Then (r, s) is said to be the *gap* of T (not related to the Gap problem in Section 7.2). Let $L(r, j, i, j)$, $r > i$, be the cost of the partial tree rooted at (i, j) with gap (r, j) such that (r, j) is the right child of (i, j). Let $L(i, s, i, j)$, $s < j$, be defined similarly. Then $L(r, j, i, j) = g(i, r) + w(i, r, j)$, and $L(i, s, i, j) = g(s, j) + w(i, s, j)$. For all other cases $L(r, s, i, j)$ is assumed to be $+\infty$. Let $L^*(r, s, i, j)$ be the cost of the optimal partial tree rooted at (i, j) with gap (r, s).

Initially, $g(i, i+1)$ is given for all $i \in [0, n-1]$, and $g(i, j) = +\infty$ for all others, and $L^*(r, s, i, j)$ is initialized to $+\infty$ for all r, s, i, j.

Given two $n \times n \times n \times n$ 4-dimensional 'matrices' L_1 and L_2, the product $L_3 = L_1 L_2$ is defined as in [12]:

$$L_3(r, s, i, j) = \min_{i \leq k_1 \leq r, s \leq k_2 \leq j} \{L_1(r, s, k_1, k_2) + L_2(k_1, k_2, i, j)\} \text{ for all } 0 \leq i \leq r < s \leq j \leq n.$$

The upper triangular matrix c is decomposed into forward diagonal strips of horizontal width $n^{\frac{1}{4}}$ each, and the entries in c are computed using the following steps:

Step 1. The g values in the first (leftmost) strip of width $n^{\frac{1}{4}}$ is computed using Rytter's algorithm [18]. This takes $\mathcal{O}\left(n^{2.25} \log n\right)$ time and $\mathcal{O}\left(\frac{n^{2.25} \log n}{B}\right)$ I/Os since a straight-forward implementation of Rytter's algorithm involves only linear scans (i.e., no random accesses).

Now starting from the second strip the following two steps are executed for each strip until the last one. This is unlike [12], where the first step (Step 2.1) is executed for all strips, followed by the execution of the second step (Step 2.2) for all strips, thus optimizing parallel computation time.

Interleaving these two steps as we do below allows space reuse in sequential computations, and thus reduces space requirement.

Step 2.1. We compute L^* of the strip (i.e., all entries $L^*(r,s,i,j)$, where both (i,j) and (r,s) belong to the strip, and $i \leq r < s \leq j$) recursively as follows. Let S be a strip of width ν (initially $\nu = n^{\frac{1}{4}}$), and let S_1 and S_2 be strips of width $\frac{\nu}{2}$ each such that they compose S, and the diagonals of S_1 are larger than those of S_2. We recursively compute L^* of S_1 followed by the recursive computation of L^* of S_2, and then we combine these results to compute L^* of S.

We set $L(r,j,i,j) = g(i,r) + w(i,r,j)$ and $L(i,s,i,j) = g(s,j) + w(i,s,j)$ initially. For these initializations $g(i,r)$ and $g(s,j)$ are retrieved from the first strip since $r - i \leq n^{\frac{1}{4}}$ and $j - s \leq n^{\frac{1}{4}}$.

Let G_S, G_1 and G_2 be L^* of S, S_1 and S_2, respectively, and let L_S be L in strip S. Then as shown in [12], $G_S = G_2 L_S G_1$. These multiplications involve computations using four dimensional arrays. We reduce these multiplications to computations using two dimensional arrays as follows.

There are at most $n\nu$ entries in a strip S of width ν. We assign an index to each entry. The first entry in the first row gets index 1, and then we assign indices using consecutive integers such that entries in higher-numbered rows get higher indices, and within the same row entries in higher-numbered columns get higher indices. Thus there are at most $n\nu$ indices. Now let X be an $n\nu \times n\nu$ matrix. We copy each entry from L_S corresponding to the strip S to X. Suppose $(i,j),(r,s) \in S$. Then if $r - i \leq \nu$ and $j - s \leq \nu$, we copy $L_S(r,s,i,j)$ to $X[id(i,j), id(r,s)]$, where $id(i,j)$ and $id(r,s)$ are the indices assigned to (i,j) and (r,s), respectively. Thus the entries from L_S corresponding to the strip S of horizontal width ν form a forward diagonal strip of horizontal width ν^2 in X. However, instead of allocating space for the entire $n\nu \times n\nu$ matrix X, we store this horizontal strip in an $n\nu \times \nu^2$ rectangular matrix. We apply similar transformations to strips S_1 and S_2, too. We can then easily multiply those larger strips in two dimensions.

There are several useful properties of the matrix multiplications performed in this step using the larger strips. First, the multiplication is min-plus, i.e., performing the same update on the same location several times do not affect the final result. Second, updates applicable on the same location can be applied in any order. Third, the updates are somewhat local, i.e., an entry in the output matrix depends only on entries that are horizontally or vertically at most at a distance ν^2 from the corresponding entry in the input matrix. Therefore, we divide the strip of width ν^2 in X into $\mathcal{O}\left(\frac{n}{\nu}\right)$ squares of size $2\nu^2 \times 2\nu^2$ each such that the last ν^2 rows of each square overlaps with the first ν^2 rows of the square below it.

Therefore, $G_S = G_2 L_S G_1$ can be computed using $\mathcal{O}\left(\frac{n}{\nu}\right)$ multiplications involving $2\nu^2 \times 2\nu^2$ matrices, each of which can be implemented cache-obliviously using I-GEP to incur only $\mathcal{O}\left(\frac{\nu^6}{B\sqrt{M}}\right)$ I/Os. For the entire strip the number of cache misses is thus $\mathcal{O}\left(\frac{n}{\nu} \times \frac{\nu^6}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n\nu^5}{B\sqrt{M}}\right)$. For $\nu = n^{\frac{1}{4}}$, the I/O complexity is $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$. Since the number of cache misses decreases by a constant factor as width decreases, the total number of cache misses is $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$. The amount of extra space used is $\mathcal{O}\left(n\nu \times \nu^2\right) = \mathcal{O}\left(n^{1.75}\right)$, which is reused by this step for processing every strip of c.

Step 2.2. Let S be a strip of width ν for which we want to compute g, and let S' be the strip of width $l\nu$ for which g has already been computed (i.e., all previous strips). Then g values for strip S can be computed by the following steps. For $(i,j) \in S$,

$$g'(i,j) = \min_{j - l\nu \leq k \leq i + l\nu} \{g(i,k) + g(k,j) + w(i,k,j)\}.$$

And for $(i,j),(r,s) \in S$,

$$g(i,j) = \min_{i \leq r,s \leq j} \left\{g'(r,s) + L^*(r,s,i,j)\right\} \qquad \text{(equation 7.5)}$$

In the step for computing $g'(i,j)$ we only need to consider those (i,r) and (s,j) such that $(i,r),(s,j) \in S'$. The computation is again min-plus, and the output is updated using entries directly from the input which is unchanged. Therefore, we can implement this step cache-obliviously using I-GEP as in Section 4.2, and update only the entries in S. We observe that S can be completely covered by $\mathcal{O}\left(\frac{n}{\nu}\right)$ non-overlapping squares of size $\nu \times \nu$ each. From Theorem 2.3 we know that I-GEP incurs $\mathcal{O}\left(\frac{\nu^2 n}{B\sqrt{M}}\right)$ cache misses for updating only the entries of any particular $\nu \times \nu$ square. Hence, total number of cache misses incurred for computing g' for S is $\mathcal{O}\left(\frac{n}{\nu} \times \frac{\nu^2 n}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^2 \nu}{B\sqrt{M}}\right)$. For $\nu = n^{\frac{1}{4}}$, the I/O complexity is thus $\mathcal{O}\left(\frac{n^{2.25}}{B\sqrt{M}}\right)$.

Now consider the step that computes g from g'. Before executing this step we copy the entries of g' corresponding to strip S to a linear array g'' such that for any $(r_1,s_1),(r_2,s_2) \in S$, $g'(r_1,s_1)$ appears before $g'(r_2,s_2)$ in g'' provided $r_1 < r_2$, or $r_1 = r_2$ and $s_1 < s_2$. Recall from Step 2.1 that for each $(i,j) \in S$, all $L^*(r,s,i,j)$ values with $i \le r, s \le j$ occupy a single row of width ν^2 in an $n\nu \times n\nu$ array X, and the ordering of the $L^*(r,s,i,j)$ values in that row is exactly similar to the ordering of the $g'(r,s)$ values in g''. Hence for every $(i,j) \in S$, we can compute $g(i,j)$ just by scanning the corresponding row in X and the relevant portion of length $\mathcal{O}\left(\nu^2\right)$ from the linear array g'' and pairing up appropriate entries from these two sources according to equation 7.5. Since there are $\mathcal{O}\left(n\nu\right)$ pairs of (i,j)'s in S, computing all g values for S will incur $\mathcal{O}\left(n\nu\left(1+\frac{\nu^2}{B}\right)\right) = \mathcal{O}\left(n\nu + \frac{n\nu^3}{B}\right)$ cache misses, which is $\mathcal{O}\left(n^{1.25} + \frac{n^{1.75}}{B}\right)$ for $\nu = n^{\frac{1}{4}}$.

Since there are $\mathcal{O}\left(n^{0.75}\right)$ strips of width $n^{\frac{1}{4}}$, step 2 will be executed $\mathcal{O}\left(n^{0.75}\right)$ times, and the total number of cache misses incurred by this step will be thus $\mathcal{O}\left(n^{0.75} \times \left(\frac{n^{2.25}}{B\sqrt{M}} + n^{1.25} + \frac{n^{1.75}}{B}\right)\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + n^2 + \frac{n^{2.5}}{B}\right)$. The I/O complexity of the entire algorithm is, therefore, $\mathcal{O}\left(\frac{n^{2.25}\log n}{B} + \frac{n^3}{B\sqrt{M}} + n^2 + \frac{n^{2.5}}{B}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$, provided $n = \Omega(M)$ and $M = \Omega\left(B^2\right)$.

The correctness of the transformation described above follows from the correctness of the parallel algorithm for the parenthesis problem given in [12], and from the observation that interleaved execution of steps 2.1 and 2.2 for different strips as above (from the largest to the smallest strip) does not affect the correctness of the algorithm. We observe that step 2.1 correctly computes L^* values for every strip since it uses only the g values for the first strip which are computed in step 1 and thus always available. Step 2.2 correctly computes g values for the current strip since it uses only g values of larger strips and L^* values of the current strip, and the order in which we execute steps 2.1 and 2.2 ensures that these values are already computed.

7.2 The Gap Problem

The *gap* problem [11, 12, 24] is a generalization of the *edit distance* problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1 x_2 \ldots x_{m-1}$ into another string $Y = y_1 y_2 \ldots y_{n-1}$ over a finite alphabet Σ, a sequence of consecutive deletes from X corresponds to a gap in X, and a sequence of consecutive inserts into X corresponds to a gap in Y [11]. In many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case two cost functions w and w' are defined, where $w(p,q)$ $(1 \le p < q \le m)$ is the cost of deleting $x_p \ldots x_{q-1}$ from X, and $w'(p,q)$ $(1 \le p < q \le n)$ is the cost of inserting $y_p \ldots y_{q-1}$ into X. Function $s(i,j)$ $(1 \le i \le m, 1 \le j \le n)$ gives the cost of replacing x_{i-1} with y_{j-1} in X.

Let $c[i,j]$ denote the minimum cost of transforming $X_{i-1} = x_1 x_2 \ldots x_{i-1}$ into $Y_{j-1} = y_1 y_2 \ldots y_{j-1}$ (where $1 \le i \le m$ and $1 \le j \le n$) under this general setting. Then

```
1. ∀_{i,j∈[1,n]} c[i,j] ← +∞,  c[1,1] ← 0
2. for i ← 1 to n do
3.     for j ← 1 to n do
4.         if i > 1 ∧ j > 1 then c[i,j] ← min(c[i,j], c[i−1,j−1] + s(i,j))
5.         for k ← 1 to j − 1 do c[i,j] ← min(c[i,j], c[i,k] + w(k,j))
6.         for k ← 1 to i − 1 do c[i,j] ← min(c[i,j], c[k,j] + w'(k,i))
```

Figure 12: DP for the gap problem derived from equation 7.6 (assuming $m = n$).

```
1. ∀_{i,j∈[1,n]} c[i,j] ← +∞,  c[1,1] ← 0
2. for k ← 1 to n do
3.     for i ← 1 to n do
4.         for j ← 1 to n do
5.             if i > 1 ∧ j > 1 then c[i,j] ← min(c[i,j], c[i−1,j−1] + s(i,j))
6.             if k < j then c[i,j] ← min(c[i,j], c[i,k] + w(k,j))
7.             if k < i then c[i,j] ← min(c[i,j], c[k,j] + w'(k,i))
```

Figure 13: Code fragment obtained by moving the innermost ***for*** loops (involving k) in Figure 12 to the outermost position. It turns out that this GEP-like code and its I-GEP implementation solve several special cases of the gap problem.

F(X, k_1, k_2)

(X is a square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$. The initial call is F$(c, 1, n)$. We assume that all $c[i,j]$ are initialized to $+\infty$, except $c[1,1]$ which is initialized to 0.)

```
1. if k1 ≥ max(i2, j2) then return                          {return if T_{X,[k1,k2]} ∩ Σ_G = ∅}
2. if k1 = k2 then
3.     if i1 > 1 ∧ j1 > 1 then c[i1,j1] ← min(c[i1,j1], c[i1−1,j1−1] + s(i1,j1))
4.     if k1 < j1 then c[i1,j1] ← min(c[i1,j1], c[i1,k1] + w(k1,j1))
5.     if k1 < i1 then c[i1,j1] ← min(c[i1,j1], c[k1,j1] + w'(k1,i1))
6. else
7.     k_m ← ⌊(k1+k2)/2⌋
8.     F(X11, k1, k_m), F(X12, k1, k_m), F(X21, k1, k_m), F(X22, k1, k_m)        {forward pass}
9.     F(X22, k_m+1, k2), F(X21, k_m+1, k2), F(X12, k_m+1, k2), F(X11, k_m+1, k2)   {backward pass}
```

Figure 14: I-GEP implementation of the GEP-like code in Figure 13. This implementation (see Figure 2) does not use static pruning as in Appendix B. The update set is $\Sigma_G = \{\langle i,j,k\rangle | i,j,k \in [1,n] \wedge k < max(i,j)\}$, and the update function is in lines 3–5. It turns out that this I-GEP code solves several special cases of the gap problem.

$$c[i,j] = \begin{cases} 0 & \text{if } i = j = 1, \\ g(i,j) & \text{if } i = 1 \ \wedge \ j > 1, \\ h(i,j) & \text{if } j = 1 \ \wedge \ i > 1, \\ \min\{\ c[i-1,j-1] + s(i,j),\ g[i,j],\ h[i,j]\ \} & \text{otherwise;} \end{cases} \qquad \text{(equation 7.6)}$$

where $g(i,j) = \min_{1 \le k < j}\{c[i,k] + w(k,j)\}$ and $h(i,j) = \min_{1 \le k < i}\{c[k,j] + w'(k,i)\}$.

In Figure 12 we give the dynamic program derived from equation 7.6 assuming $m = n$. This dynamic program runs in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n^2)$ space and incurs $\mathcal{O}\left(n^3\right)$ I/Os; an $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/O implementation can be obtained by storing c in two different matrices simultaneously: one in row-major order and the other in column-major order.

In [6] we derived a GEP-like code from equation 7.6 for the gap problem, and presented its I-GEP implementation. That result is not quite correct. We do not know if there is a GEP formulation that solves the gap problem in its full generality. However, in this section we present some results relating the gap dynamic program to I-GEP. Briefly our results are as follows.

- We present a GEP-like computation in Figure 13 which is obtained by moving the innermost loop in Figure 12 to the outermost position. This computation and its I-GEP implementation given in Figure 14 solve the following two special cases of equation 7.6 (we omit the proofs).
  - (*i*) $\forall_{i,j}\{s(i,j) \ge w'(i-1,i) + w(j-1,j)\}$, i.e., cost of replacing x_{i-1} with y_{j-1} is not cheaper than the combined cost of deleting x_{i-1} from X and inserting y_{j-1} into X.
  - (*ii*) $\forall_{i<j}\{w(i,j) = f(j-i),\ w'(i,j) = f'(j-i)\}$ and $\forall_{i \le i', j \le j'}\{s(i,j) \le s(i',j')\}$, i.e., the cost of a gap is a function of its length, and the cost of replacement is nondecreasing with increasing indices of the symbols involved.
- Though the GEP-like code in Figure 13 and its I-GEP implementation in Figure 14 do not solve the gap problem in its full generality, we show in Section 7.2.1 that with a small modification the statically pruned version of the I-GEP in Figure 14 solves the gap problem in its general form (see Appendix B for a description of static pruning).

7.2.1 Relating the Gap Dynamic Program to I-GEP

In this section we consider the I-GEP implementation of the GEP-like code in Figure 13, and study its relation to the code for the general gap problem in Figure 12. Though the GEP-like code in Figure 13 does not exactly match the GEP pattern given in Figure 1, we can still obtain an I-GEP implementation after some minor modifications to the framework (for example, changing f to accept some additional inputs such as i, j, k, $s(i,j)$, $w'(k,i)$, $w(k,j)$ and $c[i-1,j-1]$). These modifications do not change the time or I/O bound of the resulting I-GEP implementation. The I-GEP implementation is given in Figure 14 which uses the update set $\Sigma_G = \{\langle i,j,k\rangle | i,j,k \in [1,n] \wedge k < max(i,j)\}$ and the update function in lines 3–5.

Let us first examine why Figure 14 fails to solve the general version of the gap problem. Observe that when the I-GEP function F updates X_{22} using the entries of X_{12} and X_{21} in the backward pass, the entries in those two quadrants do not necessarily contain correct values. For example, the entries in X_{12} are possibly incorrect since updates involving the intra-quadrant horizontal edges of X_{12} (i.e., updates involving edges (k,j) with weight $w(k,j)$ where both $c[i,k]$ and $c[i,j]$ belong to X_{12}, that is updates in which line 4 of Figure 14 applies) have not yet been applied on X_{12}. Function F eventually applies those updates on X_{12} in the backward pass after completing the computation of X_{22}, which is

Initial Values:

$\forall_{1\le i,j\le n} c[i,j] = \begin{cases} 0 & \text{if } i=j=1, \\ +\infty & \text{otherwise.} \end{cases}$

Initial Function Call: $\mathrm{R}(c)$

$\mathrm{R}(X)$

(X is a square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$, where q (≥ 0) is an integer.)

1. ***if*** $X \neq$ a 1×1 matrix ***then***
2. $\quad k_1 \leftarrow 1,\ \ k_2 \leftarrow 2^{q-1}$
3. $\quad t_s \leftarrow (i_1 - 1, j_1 - 1)$
4. $\quad \mathrm{R}(X_{11}),$
   $\mathrm{B}_1(X_{12}, k_1, k_2, t_s),\ \ \mathrm{R}(X_{12}),$
   $\mathrm{C}_1(X_{21}, k_1, k_2, t_s),\ \ \mathrm{R}(X_{21}),$
   $\mathrm{D}_1(X_{22}, k_1, k_2, t_s),\ \ \mathrm{R}(X_{22})$

$\mathrm{B}_1(X, k_1, k_2, t_s \equiv (i_s, j_s))$

(X is a square submatrix of D such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$.)

1. ***if*** $k_1 \neq k_2$ ***then***
2. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
3. $\quad B_1(X_{11}, k_1, k_m, t_s),\ B_1(X_{12}, k_1, k_m, t_s),$
   $D_1(X_{21}, k_1, k_m, t_s),\ D_1(X_{22}, k_1, k_m, t_s)$
4. $\quad B_1(X_{22}, k_m + 1, k_2, t_s),\ B_1(X_{21}, k_m + 1, k_2, t_s),$
   $D_3(X_{12}, k_m + 1, k_2, t_s),\ D_3(X_{11}, k_m + 1, k_2, t_s)$

$\mathrm{C}_1(X, k_1, k_2, t_s \equiv (i_s, j_s))$

(X is a square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$.)

1. ***if*** $k_1 \neq k_2$ ***then***
2. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
3. $\quad \mathrm{C}_1(X_{11}, k_1, k_m, t_s),\ \mathrm{D}_1(X_{12}, k_1, k_m, t_s),$
   $\mathrm{C}_1(X_{21}, k_1, k_m, t_s),\ \mathrm{D}_1(X_{22}, k_1, k_m, t_s)$
4. $\quad \mathrm{C}_1(X_{22}, k_m + 1, k_2, t_s),\ \mathrm{D}_2(X_{21}, k_m + 1, k_2, t_s),$
   $\mathrm{C}_1(X_{12}, k_m + 1, k_2, t_s),\ \mathrm{D}_2(X_{11}, k_m + 1, k_2, t_s),$

$\mathrm{D}_i(X, k_1, k_2, t_s \equiv (i_s, j_s))$ $\qquad \{i \in [1,3]\}$

(X is a square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$.)

1. ***if*** $k_1 = k_2$ ***then***
2. $\quad$ ***if*** $i_1 > 1 \ \wedge\ j_1 > 1$ ***then*** $c[i_1, j_1] \leftarrow \min(c[i_1, j_1],\ c[i_1 - 1, j_1 - 1] + s(i_1, j_1))$
3. $\quad$ ***if*** $j_1 > j_s + k_1$ ***then*** $c[i_1, j_1] \leftarrow \min(c[i_1, j_1],\ c[i_1, j_s + k_1] + w(j_s + k_1, j_1))$
4. $\quad$ ***if*** $i_1 > i_s + k_1$ ***then*** $c[i_1, j_1] \leftarrow \min(c[i_1, j_1],\ c[i_s + k_1, j_1] + w'(i_s + k_1, i_1))$
5. ***else***
6. $\quad k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
7. $\quad \mathrm{D}_i(X_{11}, k_1, k_m, t_s),\ \ \mathrm{D}_i(X_{12}, k_1, k_m, t_s),\ \ \mathrm{D}_i(X_{21}, k_1, k_m, t_s),\ \ \mathrm{D}_i(X_{22}, k_1, k_m, t_s)$
8. $\quad \mathrm{D}_i(X_{22}, k_m + 1, k_2, t_s),\ \ \mathrm{D}_i(X_{21}, k_m + 1, k_2, t_s),\ \ \mathrm{D}_i(X_{12}, k_m + 1, k_2, t_s),\ \ \mathrm{D}_i(X_{11}, k_m + 1, k_2, t_s)$

Figure 15: Cache-oblivious implementation of the general gap problem (equation 7.6) obtained by modifying the I-GEP implementation of the GEP-like code in Figure 13. The I-GEP has been implemented with static pruning (Appendix B). Function A in this I-GEP (first row in Figure 18) is modified to obtain function R as given above, while all other functions basically remain unchanged.

$G_{ijk}(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot,\cdot,\cdot,\cdot)$ is a problem-specific function, and $\Sigma_{G_{ijk}}$ is a problem-specific set of updates to be applied on c.)

```
1. for i ← 1 to n do
2.     for j ← 1 to n do
3.         for k ← 1 to n do
4.             if ⟨i, j, k⟩ ∈ Σ_{G_ijk} then
                   c[i, j] ← f(c[i, j], c[i, k], c[k, j], c[k, k])
```

$G_{ikj}(c, 1, n)$

(The input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix. Function $f(\cdot,\cdot,\cdot,\cdot)$ is a problem-specific function, and $\Sigma_{G_{ikj}}$ is a problem-specific set of updates to be applied on c.)

```
1. for i ← 1 to n do
2.     for k ← 1 to n do
3.         for j ← 1 to n do
4.             if ⟨i, j, k⟩ ∈ Σ_{G_ikj} then
                   c[i, j] ← f(c[i, j], c[i, k], c[k, j], c[k, k])
```

Figure 16: Two simple variants of GEP (Figure 1) obtained by rearranging the ***for*** loops.

too late, and similarly with X_{21}. However, as explained below, the modifications necessary to make this algorithm work for the gap problem is simple.

For convenience of exposition we will describe the necessary modifications on the statically pruned version of Figure 14 obtained using Appendix B. Static pruning with $\Sigma_G = \{\langle i, j, k\rangle | i, j, k \in [1, n] \wedge k < max(i, j)\}$ will eliminate all recursive calls to B_2, C_2 and D_4 from the I-GEP implementation (see Figures 17-20). We will modify function A in the resulting I-GEP so that after applying the updates involving the inter-quadrant horizontal edges on X_{12} (i.e., updates involving edges (k, j) with weight $w(k, j)$ where $c[i, k] \in X_{11}$ and $c[i, j] \in X_{12}$), it recursively calls itself on X_{12} in order to apply the updates involving the intra-quadrant edges of X_{12}. It does the same with X_{21}. Hence, when inter-quadrant edges are used to update X_{22}, those updates use correct values. Finally, the entries of X_{22} are computed correctly by considering the intra-quadrant edges of X_{22}. The modified algorithm is given in Figure 15, where we denote the modified function A by R. Observe that all other functions (B_1, C_1, D_1, D_2 and D_3) basically remain unchanged. The initial function call is R(c). Before making the initial call, we set all entries of c to $+\infty$, except $c[1, 1]$ which is set to 0.

The correctness of R(X) can be proved easily by induction on q (where $2^q \times 2^q$ is the size of X), assuming that $X[1, 1]$ has already been computed correctly.

The number of cache-misses incurred by R when called on an $n \times n$ input matrix can be described using the same recurrence relation that has been used to compute the I/O complexity of I-GEP ($I(n)$ in Section 2). Therefore, the I/O complexity of R is $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$.

8 Conclusions

We have presented a cache-oblivious framework for problems that can be solved using a construct similar to the computation in Gaussian elimination without pivoting (i.e., using a GE-type construct). We have proved that this framework can be used to obtain efficient in-place cache-oblivious algorithms for several important classes of practical problems. We have also shown that if we are allowed to use only $n^2 + n$ extra space, where n^2 is the size of the input matrix, we can obtain an efficient cache-oblivious algorithm for any problem that can be solved using a GE-type construct. In addition to the practical problems solvable using this framework, it also has the potential of being used by optimizing compilers for loop transformation [16].

However, many important open questions still exist. For example:

1. Can we extend cache-oblivious I-GEP to solve function G in Figure 1 in its full generality without using any extra space, or at least using $o\left(n^2\right)$ space?
2. Can we obtain general cache-oblivious frameworks for other variants of G (for example, for those shown in Figure 16)?
3. Are there simpler transformations of 'simple DP' (or the parenthesis problem) and the gap problem to GEP?

Acknowledgement. We would like to thank Matteo Frigo for his comments. We also thank David Roche for his help in setting up STXXL.

References

[1] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[2] A.V. Aho, J.E. Hopcroft, and J.D Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, pp. 195–206, 1974.
The Princeton University Press, Princeton, New Jersey, 1957.

[3] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 297–308, 1996.

[4] C. Cherng and R.E. Ladner. Cache efficient simple dynamic programming. In *Proc. of the International Conf. on the Analysis of Algorithms*, pp. 49–58, 2005.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd ed., 2001.

[6] R.A. Chowdhury and V. Ramachandran. Cache-Oblivious Dynamic Programming. In *Proc. of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pp. 591–600, 2006.

[7] R. Dementiev. STXXL homepage, documentation and tutorial. http://stxxl.sourceforge.net/.

[8] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. In *Proc. of the 13th Annual European Symposium on Algorithms*, LNCS 3669, pp. 640–651, Springer, 2005.

[9] R.W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[10] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science*, pp. 285–297, 1999.

[11] Z. Galil, and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[12] Z. Galil, and K. Park. Parallel algorithms for dynamic programming recurrences with more than $\mathcal{O}(1)$ dependency. *Journal of Parallel and Distributed Computing*, vol. 21, pp. 213–222, 1994.

[13] K.E. Iverson. *A Programming Language.* Wiley, 1962.

[14] D.E. Knuth. Two notes on notation. *American Mathematical Monthly*, vol. 99, pp. 403–422, 1992.

[15] J. Kleinberg and E. Tardos. *Algorithm Design.* Addison-Wesely Publishing Co., Reading, MA, 2005.

[16] S.S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[17] J.-S. Park, M. Penner and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, vol. 15(9), pp. 769–782, 2004.

[18] W. Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, vol. 59, pp. 297–307, 1988.

[19] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-Linux programs). http://valgrind.kde.org/index.html

[20] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, vol. 18(4), pp. 1065–1081, 1997.

[21] L.G. Valiant. General context-free recognition in less than cubic time. *Journal of Compute and System Sciences*, vol. 10, pp. 308–315, 1975.

[22] V. Viswanathan, S. Huang and H. Liu. Parallel dynamic programming. In *Proc. of IEEE Conference on Parallel Processing*, pp. 497–500, 1990.

[23] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[24] M.S. Waterman. *Introduction to Computational Biology.* Chapman & Hall, London, UK, 1995.

[25] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: making parallel computer I/O practical. In *Proc. of the DAGS/PC Symposium*, pp. 56–63, 1993.

[26] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 30–44, 1991.

APPENDIX

A Formal Definitions of δ and π

In Section 2 we defined functions π and δ (see Definition 2.2) based on the notions of aligned subintervals and aligned subsquares. In this section we define these two functions more formally in closed form.

Recall from Definition 2.2(a) that for $x, y, z \in [1, n]$, $\delta(x, y, z)$ is defined as follows.

- If $x = y = z$, then $\delta(x, y, z) = z - 1$.
- If $x \neq z$ or $y \neq z$, then $\delta(x, y, z) = b$ for the largest aligned subsquare $[a, b], [a, b]$ of $c[1 \ldots n, 1 \ldots n]$ that contains (z, z), but not (x, y), and this subsquare is denoted by $S(x, y, z)$. Now consider the initial function call $\mathrm{F}(X, k_1, k_2)$ on c with $X \equiv c$, $k_1 = 1$ and $k_2 = n$, where $n = 2^q$ for some integer $q \geq 0$. We know from Lemma 2.1(a) that if $S(x, y, z)$ is one of the quadrants of X then it must be either X_{11} or X_{22}, otherwise $S(x, y, z)$ must be entirely contained in one of those two quadrants. Hence, in order to locate $S(x, y, z)$ in X and thus to calculate the value of $\delta(x, y, z)$ we need to consider the following four cases:

  (i) $\underline{(z, z) \in X_{11} \text{ and } (x, y) \notin X_{11}}$: $X_{11} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^{q-1}$ by definition.
  (ii) $\underline{(z, z) \in X_{22} \text{ and } (x, y) \notin X_{22}}$: $X_{22} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^q$ by definition.
  (iii) $\underline{(z, z) \in X_{11} \text{ and } (x, y) \in X_{11}}$: $S(x, y, z) \in X_{11}$, and compute $\delta(x, y, z)$ recursively from X_{11}.
  (iv) $\underline{(z, z) \in X_{22} \text{ and } (x, y) \in X_{22}}$: $S(x, y, z) \in X_{22}$, and compute $\delta(x, y, z)$ recursively from X_{22}.

  Now for each integer $u \in [1, 2^q]$, define $u' = u - 1$ which is a q-bit binary number $u'_q u'_{q-1} \ldots u'_2 u'_1$. Then it is easy to verify that the following recursive function $\rho(x, y, z, q)$ captures the recursive method of computing $\delta(x, y, z)$ described above, i.e., $\delta(x, y, z) = \rho(x, y, z, q)$ if $x \neq z$ or $y \neq z$.

$$\rho\,(x, y, z, q) = \begin{cases} 2^{q-1} & \text{if } \big(x'_q = 1 \vee y'_q = 1\big) \wedge z'_q = 0 \\ 2^q & \text{if } \big(x'_q = 0 \vee y'_q = 0\big) \wedge z'_q = 1 \\ \rho\,(x,\ y,\ z,\ q-1) & \text{if } x'_q = y'_q = z'_q = 0, \\ 2^{q-1} + \rho\,\big(x - 2^{q-1},\ y - 2^{q-1},\ z - 2^{q-1},\ q-1\big) & \text{if } x'_q = y'_q = z'_q = 1. \end{cases}$$

  We can derive a closed form for $\rho\,(x, y, z, q)$ from its recursive definition given above. Let $\boxdot$, $\boxplus$, and $\boxtimes$ denote the bitwise AND, OR and XOR operators, respectively, and define

  (a) $\alpha(x, y, z) = 2^{\lfloor \log_2 \{((x-1)\ \boxtimes\ (z-1))\ \boxplus\ ((y-1)\ \boxtimes\ (z-1))\} \rfloor}$,

  (b) $\overline{u} = 2^r - 1 - u$ (bitwise NOT), and

  (c) $\beta(x, y, z) = \big(\overline{x-1}\ \boxplus\ \overline{y-1}\big)\ \boxdot\ (z-1)$.

  Then

$$\rho\,(x, y, z, q) = \left\lfloor \frac{z-1}{2\alpha(x, y, z)} \right\rfloor \cdot 2\alpha(x, y, z) + \alpha(x, y, z) + \alpha(x, y, z) \boxdot \beta(x, y, z) \qquad \text{(equation A.7)}$$

Now we can formally define function $\delta : [1, 2^q] \times [1, 2^q] \times [1, 2^q] \rightarrow [0, 2^q]$ as follows.

$$\delta(x, y, z) = \begin{cases} z - 1 & \text{if } x = y = z, \\ \rho(x, y, z, q) & \text{otherwise (i.e., } x \neq z \ \vee \ y \neq z). \end{cases}$$

The explicit (nonrecursive) definition of δ is the following, based on equation A.7.

$$\delta(x, y, z) = \begin{cases} z - 1 & \text{if } x = y = z, \\ \left\lfloor \frac{z-1}{2\alpha(x,y,z)} \right\rfloor \cdot 2\alpha(x, y, z) + \alpha(x, y, z) + \alpha(x, y, z) \boxdot \beta(x, y, z) & \text{otherwise.} \end{cases}$$

From Definition 2.2(b), we have that function $\pi : [1, 2^q] \times [1, 2^q] \rightarrow [0, 2^q]$ is the specialization of δ to one dimension, hence we obtain:

$$\pi(x, z) = \delta(x, x, z) = \begin{cases} z - 1 & \text{if } x = z, \\ \rho(x, x, z, q) & \text{otherwise (i.e., } x \neq z). \end{cases}$$

Using the closed form for ρ, we can write π in a closed form as follows:

$$\pi(x, z) = \begin{cases} z - 1 & \text{if } x = z, \\ \left\lfloor \frac{z-1}{2\alpha'(x,z)} \right\rfloor \cdot 2\alpha'(x, z) \ + \ \alpha'(x, z) \ + \ \overline{x - 1} \ \boxdot \ (z - 1) \ \boxdot \ \alpha'(x, z) & \text{otherwise;} \end{cases}$$

where $\alpha'(x, z) = \alpha(x, x, z) = 2^{\lfloor \log_2 \{((x-1) \ \boxtimes \ (z-1)\rfloor}$.

B Static Pruning of I-GEP

In Section 2, the test in line 1 of Figure 2 enables function F to decide during runtime whether the current recursive call is necessary or not, and thus avoid taking unnecessary branches in its recursion tree. However, if the update set Σ_G is available offline (which is usually the case), we can eliminate some of these unnecessary branchings from the code during the transformation of G to F, and thus save on some overhead. As described in our earlier paper [6], we can perform this type of static pruning of F as follows.

Recall that $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ is the input submatrix, and $[k_1, k_2]$ is the range of k-values supplied to F, and they satisfy the input conditions 2.1. Let $Y \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $Z \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Then for every entry $c[i, j] \in X$, $c[i, k]$ can be found in Y and $c[k, j]$ can be found in Z. From input condition 2.1**(a)** we know that X, Y and Z must all be square matrices of the same dimensions. Input condition 2.1**(b)** requires that each of Y and Z either overlaps X completely, or does not intersect X at all. These conditions on the inputs to F implies nine possible arrangements (i.e., relative positions) of X, Y and Z. For different arrangements of these matrices we give a different name to F. Figure 20 identifies each of the nine names (A, B_1, B_2, C_1, C_2, D_1, D_2, D_3 and D_4) with the corresponding arrangement of the matrices. Each of these nine functions will be called an instantiation of F. Given an instantiation F′ of F, Figure 19 expresses the corresponding arrangement of X, Y and Z as a relationship $P(F')$ among the indices i_1, i_2, j_1, j_2, k_1 and k_2. Function A assumes that both Y and Z overlap X, i.e., all required $c[i, k]$ and $c[k, j]$ values can be found in X. Functions B_1 and B_2 both assume that Z and X overlap, but B_1 assumes that Y lies to the left of X, and B_2 assumes that Y lies to the right of X. Functions C_1 and C_2 are called when Y and X overlap, but Z and X do not. Function C_1 is called when Z lies above X, C_2 is called otherwise. Functions D_1, D_2, D_3, and D_4 assume that neither Y nor Z overlap X, and each of them assumes different relative positions of Y and Z with respect to X.

F(X, k_1, k_2) {F can be any of the nine functions $(A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, D_4)$ in column 1 of Figure 18.}
(X is a $2^q \times 2^q$ square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ for some integer $q \geq 0$. Function F assumes the following: (*a*) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$
(*b*) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$
(*c*) $P(F)$ (see Figure 19)

The initial call to F is A$(c, 1, n)$ for an $n \times n$ input matrix c, where n is assumed to be a power of 2.)

1. ***if*** $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ ***then return*** {$T_{X,[k_1,k_2]} = \{\langle i, j, k\rangle | i \in [i_1, i_2] \wedge j \in [j_1, j_2] \wedge k \in [k_1, k_2]\}$, *and* Σ_G *is the set of updates performed by* G *in Figure 1*}

2. ***if*** $k_1 = k_2$ ***then***

3. $c[i_1, j_1] \leftarrow f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$

4. ***else*** {*The following function calls are determined from the table in Figure 18. The top-left, top-right, bottom-left and bottom-right quadrants of* X *are denoted by* X_{11}, X_{12}, X_{21} *and* X_{22}, *respectively.*}

5. $k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$

6. $F_{11}(X_{11}, k_1, k_m)$, $F_{12}(X_{12}, k_1, k_m)$, $F_{21}(X_{21}, k_1, k_m)$, $F_{22}(X_{22}, k_1, k_m)$ {*forward pass*}

7. $F'_{22}(X_{22}, k_m + 1, k_2)$, $F'_{21}(X_{21}, k_m + 1, k_2)$, $F'_{12}(X_{12}, k_m + 1, k_2)$, $F'_{11}(X_{11}, k_m + 1, k_2)$ {*backward pass*}

Figure 17: Cache-oblivious I-GEP reproduced from Figure 2, but here F is assumed to be a template function that can be instantiated to any of the 9 functions given in Figure 19. The recursive calls in lines 6 and 7 are replaced with appropriate instantiations of F which can be determined from Figure 18.

F	F_{11}	F_{12}	F_{21}	F_{22}	F'_{22}	F'_{21}	F'_{12}	F'_{11}
A	A	B_1	C_1	D_1	A	B_2	C_2	D_4
B_i $(i = 1, 2)$	B_i	B_i	D_i	D_i	B_i	B_i	D_{i+2}	D_{i+2}
C_i $(i = 1, 2)$	C_i	D_{2i-1}	C_i	D_{2i-1}	C_i	D_{2i}	C_i	D_{2i}
D_i $(i \in [1, 4])$	D_i	D_i	D_i	D_i	D_i	D_i	D_i	D_i

Figure 18: Functions recursively called by F in Figure 17.

F	$P(F)$
A	$i_1 = k_1 \wedge j_1 = k_1$
B_1	$i_1 = k_1 \wedge j_1 > k_2$
B_2	$i_1 = k_1 \wedge j_2 < k_1$
C_1	$i_1 > k_2 \wedge j_1 = k_1$
C_2	$i_2 < k_1 \wedge j_1 = k_1$
D_1	$i_1 > k_2 \wedge j_1 > k_2$
D_2	$i_1 > k_2 \wedge j_2 < k_1$
D_3	$i_2 < k_1 \wedge j_1 > k_2$
D_4	$i_2 < k_1 \wedge j_2 < k_1$

Figure 19: Function specific pre-condition $P(F)$ for F in Figure 2.

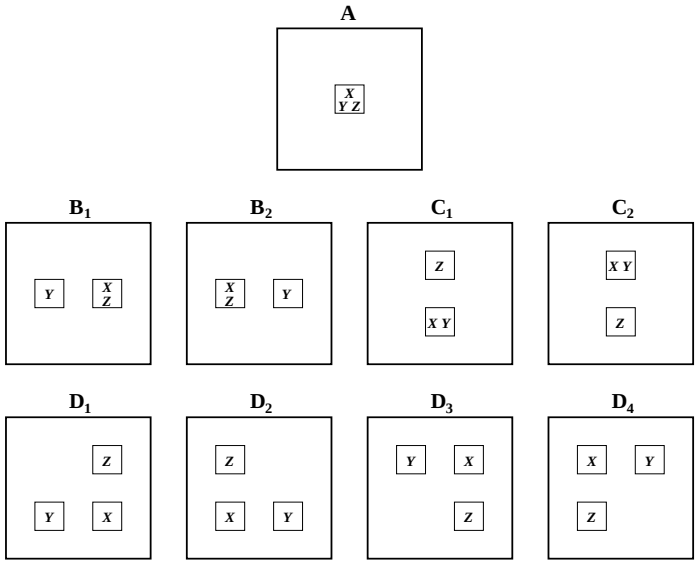

Figure 20: Relative positions of $Y \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $Z \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$ w.r.t. $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ assumed by different instantiations of F.

In Figure 17 we reproduce F from Figure 2, but replace the recursive calls in lines 6 and 7 with instantiations of F. By F_{pq} $(p, q \in [1, 2])$, we denote the instantiation of F that processes quadrant X_{pq} in the forward pass (line 6), and by F'_{pq} $(p, q \in [1, 2])$ we denote the same in the backward pass (line 7). For each of the nine instantiations of the calling function F, Figure 18 associates F_{pq} and F'_{pq} $(p, q \in [1, 2])$ with appropriate instantiations.

A given computation need not necessarily make all recursive calls in lines 6 and 7. Whether a specific recursive call to a function F′ (say) will be made or not depends on $P(F')$ (see Figure 19) and the GEP instance at hand. For example, if $i \geq k$ holds for every update $\langle i, j, k\rangle \in \Sigma_G$, then we do not make any recursive call to function C_2 since the indices in the updates can never satisfy $P(C_2)$. As has already been pointed out in [6], the I-GEP implementation of the code for Gaussian elimination without pivoting can employ static pruning very effectively, in which case, we can eliminate all recursive calls except for those to A, B_1, C_1 and D_1.

The initial function call is $A(c, 1, n)$ (F instantiated to A).