Theory Comput Syst (2010) 47: 878–919
DOI 10.1007/s00224-010-9273-8

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

Rezaul Alam Chowdhury · Vijaya Ramachandran

Published online: 8 June 2010

Abstract We consider triply-nested loops of the type that occur in the standard Gaussian elimination algorithm, which we denote by GEP (or the Gaussian Elimination Paradigm). We present two related cache-oblivious methods I-GEP and C-GEP, both of which reduce the number of cache misses incurred (or I/Os performed) by the computation over that performed by standard GEP by a factor of $\sqrt{M}$, where M is the size of the cache. Cache-oblivious I-GEP computes in-place and solves most of the known applications of GEP including Gaussian elimination and LU-decomposition without pivoting and Floyd-Warshall all-pairs shortest paths. Cache-oblivious C-GEP uses a modest amount of additional space, but is completely general and applies to any code in GEP form. Both I-GEP and C-GEP produce system-independent cache-efficient code, and are potentially applicable to being used by optimizing compilers for loop transformation.

We present parallel I-GEP and C-GEP that achieve good speed-up and match the sequential caching performance cache-obliviously for both shared and distributed caches for sufficiently large inputs.

We present extensive experimental results for both in-core and out-of-core performance of our algorithms. We consider both sequential and parallel implementations, and compare them with finely-tuned cache-aware BLAS code for matrix multiplication and Gaussian elimination without pivoting. Our results indicate that cache-oblivious GEP offers an attractive trade-off between efficiency and portability.

This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609. This journal submission incorporates results on the cache-oblivious paradigm that were presented in preliminary form in [8] and [9].

R.A. Chowdhury (✉) · V. Ramachandran
Department of Computer Sciences, University of Texas, Austin, TX 78712, USA
e-mail: shaikat@cs.utexas.edu

V. Ramachandran
e-mail: vlr@cs.utexas.edu

Keywords Cache-oblivious · Gaussian elimination · All-pairs shortest path · Matrix multiplication · Parallel · Tiling

1 Introduction

Memory in modern computers is typically organized in a hierarchy with registers in the lowest level followed by several levels of caches (L1, L2 and possibly L3), RAM, and disk. The access time and size of each level increases with its depth, and block transfers are used between adjacent levels to amortize the access time cost.

The *two-level I/O model* [2] is a simple abstraction of the memory hierarchy that consists of a *cache* (or *internal memory*) of size M, and an arbitrarily large *main memory* (or *external memory*) partitioned into blocks of size B. An algorithm is said to have caused a *cache-miss* (or *page fault*) if it references a block that does not reside in the cache and must be fetched from the main memory. The *cache complexity* (or *I/O complexity*) of an algorithm is the number of block transfers or I/O operations it causes, which is equivalent to the number of cache misses it incurs. Algorithms designed for this model often crucially depend on the knowledge of M and B, and thus do not adapt well when these parameters change.

The *ideal-cache model* [16] is an extension of the two-level I/O model which assumes that an optimal cache replacement policy is used, and requires that the algorithm remains oblivious of cache parameters M and B. A *cache-oblivious* algorithm is flexible and portable, and simultaneously adapts to all levels of a multi-level memory hierarchy. The assumption of an optimal cache replacement policy can be reasonably approximated by a standard cache replacement method such as LRU. 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 this paper we introduce a cache-oblivious framework, which we call *GEP* or the *Gaussian Elimination Paradigm*. This framework applies to 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 $\mathcal{O}(n^3)$ time, use $\mathcal{O}(n^2)$ space and incur $\mathcal{O}(\frac{n^3}{B})$ cache-misses. We present two versions of our cache-oblivious framework:

- In-place cache-oblivious I-GEP, which executes generalized versions of several important special cases of GEP including Gaussian elimination and LU-decomposition without pivoting, Floyd-Warshall all-pairs shortest paths and matrix multiplication. This framework takes full advantage of both spatial and temporal locality of data to incur only $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$ cache-misses while still running in $\mathcal{O}(n^3)$ time and without using any extra space.
- Cache-oblivious C-GEP, which executes GEP in its full generality with the same time and cache-bounds as I-GEP while using $\mathcal{O}(n^2)$ space.

We present a parallel version of I-GEP (as well as C-GEP), and we analyze the parallel running time as well as the caching performance under both distributed and shared caches. In both cases our parallel algorithm is cache-oblivious and matches the sequential cache-complexity while achieving good speed-up.

We present extensive experimental results. Our experimental results show the following:

- 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.
- A `pthreads` implementation of parallel I-GEP on an 8-core CMP gives good speed-up.
- Experimental results comparing performance of I-GEP with that of highly optimized cache-aware BLAS routines for square matrix multiplication and Gaussian elimination without pivoting show that our implementation of I-GEP runs moderately slower than native BLAS; however, I-GEP incurs fewer number of cache misses. It should also be noted that I-GEP is much simpler to code, easily supports multithreading and is portable across machines.

One potential application of the 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 Fig. 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 $\langle i, j, k \rangle$ 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

$\mathrm{G}(c, 1, n)$

(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 ← 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], c[i, k], c[k, j], c[k, k])
```

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

Gaussian Elimination Paradigm or *GEP*. Many practical problems fall in this category, for example:

- LU decomposition and Gaussian elimination without pivoting with $\Sigma_G = \{\langle i, j, k\rangle : (1 \le k \le n-2) \wedge (k < i < n) \wedge (k < j \le n)\}$ and $f(x, u, v, w) = x - \frac{u}{w} \times v$.
- Floyd-Warshall all-pairs shortest paths with $\Sigma_G = \{\langle i, j, k\rangle : 1 \le i, j, k \le n\}$ and $f(x, u, v, \cdot) = \min\{x, u + v\}$.

Some other problems including matrix multiplication can be solved using GEP through structural transformation.

The running time of G is $\mathcal{O}(n^3)$ 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 cache complexity is $\mathcal{O}(\frac{n^3}{B})$ provided the cache misses incurred in line 4, if any, are only 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 $n = 2^q$ for some integer $q \ge 0$.

1.2 Organization of the Paper

In Sect. 2, we present and analyze an $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$ I/O in-place cache-oblivious algorithm, called I-GEP, which solves several important special cases of GEP. We prove some theorems relating the computation in I-GEP to the computation in GEP. In Sect. 3, we describe generalized versions of three major applications of I-GEP (Gaussian elimination without pivoting, matrix multiplication and Floyd-Warshall's APSP). Succinct proofs of correctness of these I-GEP implementations can be obtained using results from Sect. 2.

In Sect. 4, 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). In Sect. 5 we present parallel I-GEP (and C-GEP) and analyze its performance on both distributed and shared caches.

We consider the potential application of the GEP framework in compiler optimizations in Sect. 6. In Sect. 7 we present all of our experimental results: in Sect. 7.1 we present results comparing C-GEP, I-GEP and GEP for Floyd-Warshall, in Sect. 7.2 results comparing I-GEP to BLAS routines, and in Sect. 7.3 experimental results on parallel I-GEP using `pthreads`. Finally, we present some concluding remarks in Sect. 8.

1.3 Related Work

Known cache-oblivious algorithms for Gaussian elimination for solving systems of linear equations are based on LU decomposition. In [6, 33] cache-oblivious algorithms performing $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$ I/O operations are given for LU decomposition without pivoting; the algorithm in [30] 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 Sect. 3.1) 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.

Cache-oblivious multiplication of rectangular matrices is presented in [16]. The matrix multiplication algorithm for square matrices that we obtain with I-GEP is essentially the same as the one in [16].

A cache-oblivious algorithm for Floyd-Warshall's APSP algorithm is given in [27] (see also [13]). The algorithm runs in $\mathcal{O}(n^3)$ time and incurs $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$ cache misses. Our I-GEP implementation of Floyd-Warshall's APSP (see Sect. 3.3) produces exactly the same algorithm.

The main attraction of the Gaussian Elimination Paradigm 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 for all of them.

2 Cache-Oblivious I-GEP

In this section we introduce and analyze I-GEP, a recursive function F given in Fig. 2 that is cache-oblivious, computes in-place, and is a provably correct implementation of GEP in Fig. 1 for several important special cases of f and Σ_G including Floyd-Warshall's APSP, Gaussian elimination without pivoting and matrix multiplication. 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 other implementation (C-GEP) which we give in Sect. 4 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, where, the notation $[u, v]$ is used to represent the closed integer range $\{x \in Z | u \leq x \leq v\}$ in the standard interval notation:

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 $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$,

where, the notation $[u, v]$ represents the closed integer range $\{x \in Z | u \leq x \leq v\}$ in the standard interval notation.

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*** no update $\langle i, j, k \rangle \in \Sigma_G$ applicable on X with $k \in [k_1, k_2]$ exists ***then return*** {*Section 1.1 defines* Σ_G}
2. ***if*** $k_1 = k_2$ ***then*** {*Base case*}
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. $k_m \leftarrow \lfloor \frac{k_1 + k_2}{2} \rfloor$
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*}

Fig. 2 Cache-oblivious I-GEP. For several special cases of f and Σ_G in Fig. 1, we show that F performs the same computation as G (see Sect. 3), though there are some cases of f and Σ_G where the computations return different results

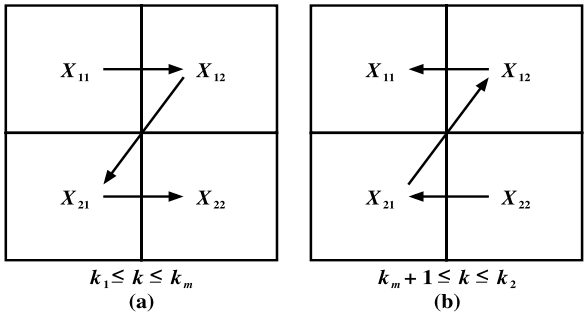

Fig. 3 Processing order of quadrants of X by F: (**a**) forward pass, (**b**) backward pass

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 Fig. 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) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ *and* $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$.

Let $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Then for every entry $c[i, j] \in X$, $c[i, k]$ can be found in U and $c[k, j]$ can be found in V. Input condition (a) requires that X, U and V must all be square matrices of the same size. Input condition (b) requires that $(X \equiv U) \vee (X \cap U = \emptyset)$, i.e., either U overlaps X completely, or does not intersect X at all. Similar constraints are imposed on V, 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 Fig. 3. The initial function call is $\mathrm{F}(c, 1, n)$.

Some Basic Properties of GEP 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 $j_1 < j_2$.

Properties of I-GEP We prove two theorems that reveal several important properties of F. Theorem 2.2 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 2.2 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 Sect. 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 Fig. 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 until Sect. 5). 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 2.2 *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 (a) $\langle i, j, k\rangle \in \Sigma_F \Rightarrow \langle i, j, k\rangle \in \Sigma_G$ holds by the check in line 1 of Fig. 2. The reverse direction can be proved by induction on q, where $2^q \times 2^q$ is the size of the matrix X input to F.

Let $\Sigma_{F(X,k_1,k_2)}$ denote the set of updates performed by F when called with parameters X, k_1 and k_2. Then $\Sigma_F = \Sigma_{F(c,1,n)}$ by definition. Also let $T_{X,[k_1,k_2]} = \{\langle i, j, k\rangle | c[i, j] \in X \wedge k \in [k_1, k_2]\}$, i.e., the set of all updates $\langle i, j, k\rangle$ with $k \in [k_1, k_2]$ that are applicable on X.

We will prove that $\langle i, j, k\rangle \in \Sigma_G \cap T_{X,[k_1,k_2]} \Rightarrow \langle i, j, k\rangle \in \Sigma_{F(X,k_1,k_2)}$. If $q = 0$, then X has only one entry, and clearly the proposition holds (base case). Now suppose the proposition holds for some value p (≥ 0) of q (inductive hypothesis) and consider $q = p + 1$. Function F recursively calls itself on each quadrant of X for $k \in [k_1, k_m]$ in line 6, and for $k \in [k_m + 1, k_2]$ in line 7, where $k_m = \lfloor \frac{k_1+k_2}{2} \rfloor$. Thus the recursive calls cover all entries of $T_{X,[k_1,k_2]}$, and also $\Sigma_{F(X,k_1,k_2)}$ is the union of all updates performed by them. Hence by inductive hypothesis $\langle i, j, k\rangle \in \Sigma_G \cap T_{X,[k_1,k_2]} \Rightarrow \langle i, j, k\rangle \in \Sigma_{F(X,k_1,k_2)}$. Since the initial call to F is made with $X = c[1 \ldots n, 1 \ldots n]$ and $[k_1, k_2] = [1, n]$, we have $\Sigma_G \subseteq T_{X,[k_1,k_2]}$ in that case, and therefore, $\langle i, j, k\rangle \in \Sigma_G \Rightarrow \langle i, j, k\rangle \in \Sigma_F$.

(b) Suppose $t_1 \neq t_2$. Observe that all recursive calls in lines 6 and 7 of F are made on mutually disjoint 3 dimensional subranges of $[i_1, i_2] \times [j_1, j_2] \times [k_1, k_2]$, and also that all updates to the input matrix c are performed when F is called with an input submatrix X consisting of a single cell of c, and each such call applies only one update to that cell (in line 3). Therefore, at some level of recursion $\langle i, j, k, t_1\rangle$ and $\langle i, j, k, t_2\rangle$ must have ended up in the subranges of two different recursive calls, i.e., the first three components (i, j, k) of $\langle i, j, k, t_1\rangle$ and $\langle i, j, k, t_2\rangle$ cannot be exactly the same, which is a contradiction. Hence $t_1 = t_2$.

(c) Observe that for all recursive calls in line 6, $k \in [k_1, k_m]$, and for those in line 7, $k \in [k_m + 1, k_2]$, where $k_1 \le k_m \le k_2$. Hence at some level of recursion $\langle i, j, k'_1, t_1\rangle$ will end up in a recursive call in line 6, and $\langle i, j, k'_2, t_2\rangle$ will end up in a recursive call in line 7. Since all updates due to the recursive calls in line 6 will be made before any of those due to the recursive calls in line 7, it follows that $t_2 > t_1$. □

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 2.3 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 $b - a + 1 = b' - a' + 1$.

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

Observation 2.4 *Consider the call* $\mathrm{F}(c, 1, n)$. *Every recursive call is on an aligned subsquare of* c, *and every aligned subsquare of* c *of width* 2^r *for* $r \le q$ *is invoked in exactly* $n/2^r$ *recursive calls on disjoint aligned subintervals* $[k_1, k_2]$ *of length* 2^r *each.*

Definition 2.5 Let x, y, and z be integers, $1 \le x, y, z \le n$.

(a) For $x \ne z$ or $y \ne z$, 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 $x \ne z$, *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 $x \ne z$, 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 2.5.

Observation 2.6

(a) *If* $x \ne z$ *or* $y \ne z$ *then* $\delta(x, y, z) \ge z$, *and if* $x \ne z$ *then* $\pi(x, z) \ge z$; $I(x, z)$ *and* $I(z, x)$ *have the same length while* $S(x, y, z)$ *and* $S'(x, y, z)$ *have the same size*; *and*

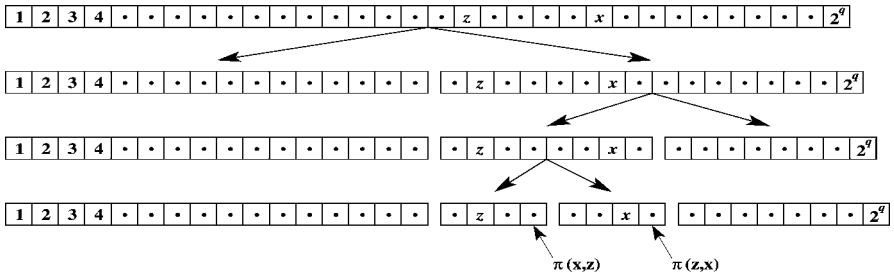

Fig. 4 Evaluating $\pi(x, z)$ and $\pi(z, x)$ for $x > z$: Given $x, z \in [1, 2^q]$ such that $x > z$, we start with an initial sequence of 2^q consecutive integers in $[1, 2^q]$, and keep splitting the segment containing both x and z at midpoint until x and z fall into different segments. The largest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

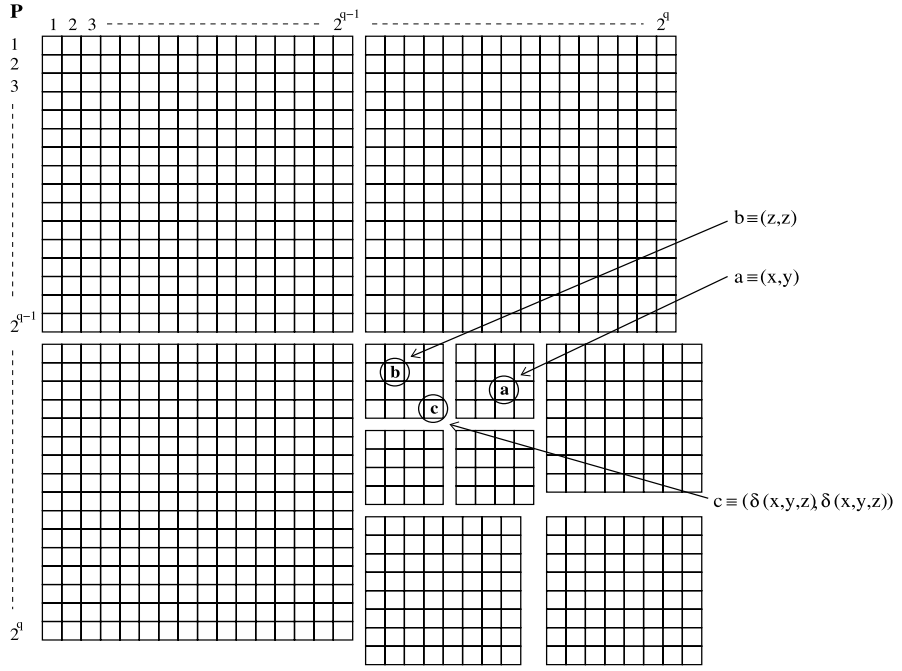

Fig. 5 Evaluating $\delta(x, y, z)$: Given $x, y, z \in [1, 2^q]$ (where $q \in \mathbb{Z}^+$), such that $x \neq z \vee y \neq z$, we start with an initial square $P[1 \ldots 2^q, 1 \ldots 2^q]$, and keep splitting the square (initially the entire square P) containing both $P[x, y]$ and $P[z, z]$ into 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 $\delta(x, y, z)$

$S(x, y, z)$ 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 2.4, this lemma can be proved by backward induction on q. As before the initial call is to $\mathrm{F}(c,1,n)$.

Lemma 2.7 *Let i,j,k be integers, $1 \le i,j,k \le n$, 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 $k' \le k$ 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 2.8 *Let δ and π be as defined in Definition 2.5. 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 one by one.

$c[i,j]$: By Theorem 2.2, 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 $c_{k-1}(i,j)$.

$c[k,k]$: 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 2.5, and consider the recursive call $F(X, k_1, k_2)$ 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 2.7). Also, as noted in Lemma 2.7, 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 2.7 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 2.5 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 2.5, $\delta(i, j, k)$ 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 $\delta(i, j, k)$.

$c[i, k]$ and $c[k, j]$: Similar to the proof for $c[k, k]$ but using parts (b) and (c) of Lemma 2.7. □

Cache Complexity Let $Q(n)$ be an upper bound on the number of cache-misses incurred by F for an input of size $n \times n$. The following recurrence follows from the observation that when the input is small enough to fit into the cache the only cache-misses incurred by F are those for reading in the initial input matrices to the cache and for writing out the final output to the main memory, otherwise the total number of cache-misses is simply the sum of the cache-misses incurred by the recursive calls:

$$Q(n) \leq \begin{cases} \mathcal{O}\left(n + \frac{n^2}{B}\right) & \text{if } n^2 \leq \gamma M, \\ 8Q\left(\frac{n}{2}\right) & \text{otherwise,} \end{cases} \tag{1}$$

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 $Q(n) = \mathcal{O}(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}) = \mathcal{O}(\frac{n^3}{B\sqrt{M}})$ (assuming a tall cache, i.e., $M = \Omega(B^2)$).

Since I-GEP can be used for multiplying matrices, it follows from the I/O lower bound of matrix multiplication [22] that the cache complexity of I-GEP is, in fact, tight for any algorithm that performs $\Theta(n^3)$ operations in order to implement the general version of the GEP computation as defined in Sect. 1.1.

Time and Space Complexities Since I-GEP is in-place, its space complexity is determined by the size of its input matrices which is clearly $\Theta(n^2)$. Time complexity of I-GEP is given by the following recurrence relation, where $T(n)$ denotes the running

time of I-GEP on an input of size $n \times n$.

$$T(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n \leq 1, \\ 8T\left(\frac{n}{2}\right) + \mathcal{O}(1) & \text{otherwise.} \end{cases} \tag{2}$$

Solving, we get $T(n) = \mathcal{O}(n^3)$.

Static Pruning of I-GEP In line 1 of Fig. 2, function $\mathrm{F}(X, k_1, k_2)$ performs dynamic pruning of its recursion tree by computing the set of all updates $\langle i, j, k\rangle \in \Sigma_G$ with $k \in [k_1, k_2]$ that are applicable on the input submatrix X. 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.

3 Applications of Cache-Oblivious I-GEP

In this section we consider I-GEP for three major GEP instances. Though the C-GEP implementation given in Sect. 4 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 Sect. 7 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.

- A class of applications that includes Gaussian elimination without pivoting, where we restrict Σ_G but allow f to be unrestricted.
- 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.
- Path computations over closed semirings which includes Floyd-Warshall's APSP algorithm [15] and Warshall's algorithm for finding transitive closures [31]. For this class of problems we specify both f and Σ_G.

3.1 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 [12]. 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 ith $(1 \leq i < n)$ row represents the equation $a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n-1}x_{n-1} = b_i$. The method proceeds in 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

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 for Gaussian elimination without pivoting $f(x,u,v,w) = x - \frac{u}{w} \times v$. The set of updates is $\Sigma_G = \{\langle i,j,k\rangle : (1 \le k \le n-2) \wedge (k < i < n) \wedge (k < j \le n)\}$ which is applied in step 4 of the algorithm.)

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*** $(k \le n-2) \wedge (k < i < n) \wedge (k < j)$ ***then*** $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$

Fig. 6 A general form of the first phase of Gaussian elimination without pivoting

matrix by back substitution. It is straight-forward to implement this second phase in $\mathcal{O}(n^2)$ time and $\mathcal{O}(\frac{n^2}{B})$ I/Os, so we will concentrate on the first phase.

The first phase is an instantiation of the GEP code in Fig. 1. In Fig. 6 we give a computation that is a general form of the computation in the first phase of Gaussian elimination without pivoting in the sense that the update function f in Fig. 6 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., $\Sigma_G = \{\langle i, j, k\rangle : (1 \le k \le n-2) \wedge (k < i < n) \wedge (k < j \le n)\}$.

The correctness of the I-GEP implementation of the code in Fig. 6 can be proved by induction on k using Theorem 2.8 and by observing that each $c[i, j]$ $(1 \le i, j \le n)$ settles down (i.e., is never modified again) before it is ever used on the right hand side of an update.

As described in 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 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 [6, 30, 33].

3.2 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 shown in Fig. 7(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.

The algorithm in Fig. 7(b) generalizes the computation in step 4 of Fig. 7(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 matrix c is disjoint from matrices a, b, and d.

The correctness of the I-GEP implementation of the code in Fig. 7(b) follows from Theorem 2.2 and from the observation that matrices a, b and d remain unchanged throughout the computation.

```
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)

Fig. 7 (**a**) Modified matrix multiplication algorithm, (**b**) a more general form of the algorithm in Fig. 7(a)

Initial Values:

$$\forall_{1\le i,j\le n} c[i,j] = \begin{cases} 1 & \text{if } i = j, \\ l(v_i, v_j) & \text{otherwise.} \end{cases}$$

(a)

Computation of Path Costs:

```
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] ⊕ (c[i, k] ⊙ c[k, j])
```

(b)

Fig. 8 Computation of path costs over a closed semiring $(S, \oplus, \odot, 0, 1)$: (**a**) initialization of c, (**b**) computation of path costs

3.3 Path Computations over a Closed Semiring

An algebraic structure known as a *closed semiring* [3] serves as a general framework for solving path problems in directed graphs. In [3], 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 [15] and Warshall's algorithm for finding transitive closures [31] 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 presence of cycles), $c[i, j]$ will still be well-defined due to the properties of a closed semiring.

The algorithm given in Fig. 8(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 $\mathcal{O}(n^3)$ operations and uses $\mathcal{O}(n^2)$ space. Floyd-Warshall's APSP is a specialization of the algorithm in Fig. 8(b) in that it performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

Correctness of I-GEP Implementation of Fig. 8(b) Recall that $c_0(i, j)$ is the initial value of $c[i, j]$ received by the I-GEP function F in Fig. 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 Fig. 8(b) follows from the following lemma.

Lemma 3.1 *For all* $i, j, k \in [1, n]$, $Q^k_{i,j} \supseteq P^k_{i,j}$.

Proof The proof is by induction on k. The proposition holds trivially for $k = 0$, since for all $i, j \in [1, n]$, $c^0(i, j)$ is just the cost of edge (v_i, v_j), and $P^0_{i,j}$ contains only the edge (v_i, v_j) (observe that because of the initialization in Fig. 8(a), we can assume that (v_i, v_j) always exists with the cost of this edge being 1 if $i = j$, and $l(v_i, v_j)$ otherwise).

Now suppose the proposition holds for all $k \in [0, k']$, where $k' \in [0, n-1]$. We will prove that it holds for $k = k' + 1$ and thus for all $k \in [0, k' + 1]$.

For any $i, j \in [1, n]$, consider the update $\langle i, j, k' + 1\rangle$, i.e., $c[i, j] = c[i, j] \oplus (c[i, k' + 1] \odot c[k' + 1, j])$. We know from Theorem 2.2 that immediately before this update $c[i, j] = c_{k'}(i, j)$, and immediately after this update $c[i, j] = c_{k'+1}(i, j)$. We also know from Theorem 2.8 that immediately before this update $c[i, k' + 1] = c_{\pi(j,k'+1)}(i, k' + 1)$ and $c[k' + 1, j] = c_{\pi(i,k'+1)}(k' + 1, j)$ hold. Since $\pi(j, k' + 1) \geq k'$ and $\pi(i, k' + 1) \geq k'$ follow from the definition of π, we have $Q^{\pi(j,k'+1)}_{i,k'+1} \supseteq Q^{k'}_{i,k'+1}$ and $Q^{\pi(i,k'+1)}_{k'+1,j} \supseteq Q^{k'}_{k'+1,j}$. From inductive hypothesis we know that $Q^{k'}_{i,j} \supseteq P^{k'}_{i,j}$, $Q^{k'}_{i,k'+1} \supseteq P^{k'}_{i,k'+1}$ and $Q^{k'}_{k'+1,j} \supseteq P^{k'}_{k'+1,j}$. Hence, the addition of $c[i, k' + 1] \odot c[k' + 1, j]$ to $c[i, j]$ ensures that all paths that first go from v_i to $v_{k'+1}$ and then from $v_{k'+1}$ to v_j such that neither subpath has an intermediate vertex higher than $v_{k'}$ are also considered in the computation of $c_{k'+1}(i, j)$. Therefore, $Q^{k'+1}_{i,j} \supseteq P^{k'+1}_{i,j}$. Thus, the proposition holds for all $i, j \in [1, n]$ and all $k \in [0, k' + 1]$.

Now proceeding up to $k' = n - 1$, we conclude that $Q^k_{i,j} \supseteq P^k_{i,j}$ holds for all $i, j \in [1, n]$ and all $k \in [0, n]$. □

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 3.1, results in $Q^n_{i,j} = P^n_{i,j}$.

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

In this section we present a completely general cache-oblivious framework for GEP that matches the time and cache complexity of I-GEP. In order to express mathematical expressions with conditionals in compact form, in this section we will use *Iverson's convention* [23, 24] 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.

4.1 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 $i, j \in [1, n]$

Table 1 States of $c[i, j]$, $c[i, k]$, $c[k, j]$ and $c[k, k]$ immediately before applying $\langle i, j, k\rangle \in \Sigma_G$

Cell	G	F
$c[i, j]$	$\hat{c}_{k-1}(i, j)$	$c_{k-1}(i, j)$
$c[i, k]$	$\hat{c}_{k-\lvert j\le k\rvert}(i, k)$	$c_{\pi(j,k)}(i, k)$
$c[k, j]$	$\hat{c}_{k-\lvert i\le k\rvert}(k, j)$	$c_{\pi(i,k)}(k, j)$
$c[k, k]$	$\hat{c}_{k-\lvert(i<k)\vee(i=k\wedge j\le k)\rvert}(k, k)$	$c_{\delta(i,j,k)}(k, k)$

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 kth iteration of the outer ***for*** loop in G, where $i, j \in [1, n]$ and $k \in [0, n]$.

In Table 1, we tabulate the exact states of $c[i, j]$, $c[i, k]$, $c[k, j]$ and $c[k, k]$ immediately before G or F applies an update $\langle i, j, k\rangle \in \Sigma_G$. Entries in the 2nd column are determined by inspecting the code in Fig. 1, while those in the 3rd column follows from Theorem 2.8.

It follows from Definition 2.5 that for $i, j < k, \pi(j, k) \neq k - |j \le k|, \pi(i, k) \neq k - |i \le k|$ and $\delta(i, j, k) \neq k - |(i < k) \vee (i = k \wedge j \le k)|$. Therefore, though both G and F start with the same input matrix, at certain points in the 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 could return different output values. Whether the final output matrix returned by the two algorithms are the same depends on f, Σ_G and the input values.

As an example (see Fig. 9), consider a 2×2 input matrix c, and let $\Sigma_G = \{\langle i, j, k\rangle | 1 \le i, j, k \le 2\}$. 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 $f(x, y, z, w) = x + w$, then F will output $c[2, 1] = 8$, while G will output $\hat{c}[2, 1] = 2$.

4.2 C-GEP Using $4n^2$ Additional Space

We first define a quantity τ_{ij}, which plays a crucial role in the extension of I-GEP to the completely general C-GEP.

Definition 4.1 For $1 \le i, j, l \le n$, we define $\tau_{ij}(l)$ to be the largest integer $l' \le l$ such that $\langle i, j, l'\rangle \in \Sigma_G$ provided such an update exists, and 0 otherwise. More formally, for all $i, j, l \in [1, n]$, $\tau_{ij}(l) = \max_{l'}\{l' \mid l' \le l \wedge \langle i, j, l'\rangle \in \Sigma_G \cup \{\langle i, j, 0\rangle\}\}$.

The significance of τ of can be explained as follows. We know from Theorem 2.2 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 $c_l(i, j)$ ($\hat{c}_l(i, j)$,

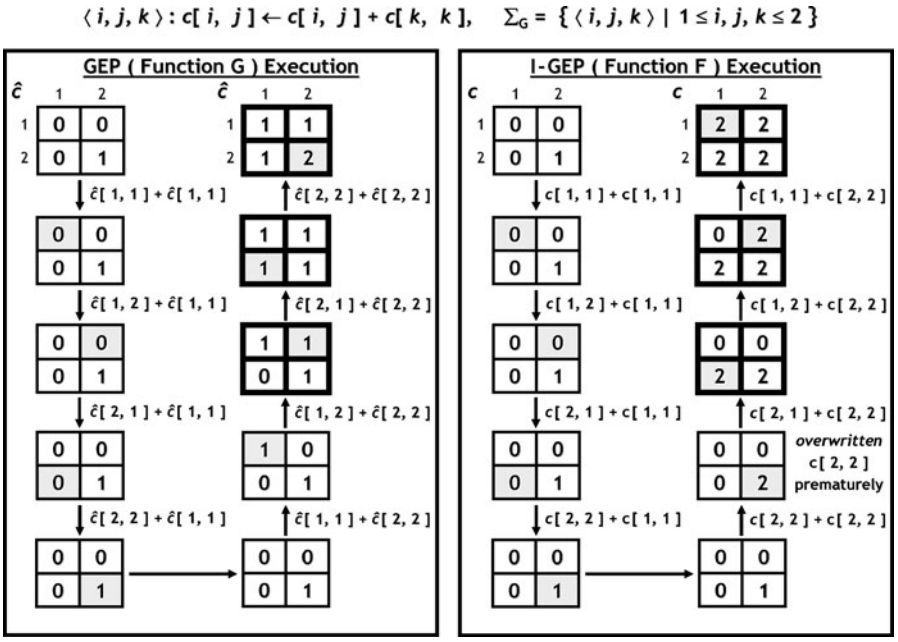

Fig. 9 An instance of GEP for which GEP (function G in Fig. 1) and I-GEP (function F in Fig. 2) compute different output values for the same input matrix. The input is a 2×2 matrix c, $\Sigma_G = \{\langle i, j, k\rangle | 1 \le i, j, k \le 2\}$, and $f(x, y, z, w) = x + w$. The figure shows step-by-step execution of both G and F on the same input matrix, and the steps in which the two functions produce different output values are drawn with *thick lines*

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 2.5) and τ: we know from Theorem 2.8 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. A similar observation holds for δ defined in Definition 2.5.

We extend I-GEP to full generality by modifying F in Fig. 2 so that it performs updates according to the second column of Table 1 instead of the third column. 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 $c_{k-|j\le k|}(i, k)$, $c_{k-|i\le k|}(k, j)$ and $c_{k-|(i<k)\vee(i=k\wedge j\le k)|}(k, k)$ before it computes $c_k(i, j)$ since we know from Observation 2.6 that $\pi(j, k) \ge k - |j \le k|$, $\pi(i, k) \ge k - |i \le k|$ and $\delta(i, j, k) \ge k - |(i < k) \vee (i = k \wedge j \le k)|$ 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 2.2 and the definition of τ.

Observation 4.2 *Immediately before* F *applies the update* $\langle i, j, k\rangle \in \Sigma_G$:

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

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

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

If the condition in Observation 4.2(a) holds, we must save $c_{k-|j\le k|}(i,k)$ as soon as it is generated so that it can be used later by $\langle i,j,k\rangle$. However, $c_{k-|j\le k|}(i,k)$ is not necessarily generated by $\langle i,k,k-|j\le k|\rangle$ since this update may not exist in Σ_G in the first place. If $\tau_{ij}(k-|j\le k|)\neq 0$, then $\langle i,k,\tau_{ij}(k-|j\le k|)\rangle$ is the update that generates $c_{k-|j\le k|}(i,k)$, and we must save this value after applying this update and before some other update modifies it. If $\tau_{ij}(k-|j\le k|)=0$, then $c_{k-|j\le k|}(i,k) = c_0(i,k)$, 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 Fig. 1.

Observation 4.3 *The GEP code in Fig. 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 4.3(a) do not need to be stored separately, since we know from Theorem 2.2 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 4.3. By inspecting the code in Fig. 1 (see also the second column of Table 1), we observe that immediately before G applies the update $\langle i,j',j\rangle$ in Observation 4.3(b), $c[i,j] = \hat{c}_{j-1}(i,j) = \hat{c}_{\tau_{ij}(j-1)}(i,j)$ if $j' \le j$, 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 4.3(c), $c[i,j] = \hat{c}_{i-1}(i,j) = \hat{c}_{\tau_{ij}(i-1)}(i,j)$ if $i' \le i$, 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 4.3(d), $c[i,j] = \hat{c}_{i-1}(i,j) = \hat{c}_{\tau_{ij}(i-1)}(i,j)$ if $i' < i \vee (i'=i \wedge j' \le j)$, 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.

The modified version of F, which we call H, is shown in Fig. 10. Function 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. After it reaches the base case (i.e., $i_1 = i_2$, $j_1 = j_2$ and $k_1 = k_2$) and computes the

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*** no update $\langle i, j, k \rangle \in \Sigma_G$ applicable on X with $k \in [k_1, k_2]$ exists ***then return*** {*Section 1.1 defines* Σ_G}
2. ***if*** $k_1 = k_2$ ***then*** {*Base case*}
3. $i \leftarrow i_1,\ j \leftarrow j_1,\ k \leftarrow k_1$
4. $c[i,j] \leftarrow f(c[i,j], u_{|j>k|}[i,k], v_{|i>k|}[k,j], u_{|(i>k)\ \vee\ (i=k\ \wedge\ j>k)|}[k,k])$ {*Update* $c[i,j]$}
5. {*Update appropriate* $u_0/u_1/v_0/v_1$ *values*}
if $k = \tau_{ij}(j-1)$ ***then*** $u_0[i,j] \leftarrow c[i,j]$ {$\tau_{ij}(l) = \max_{l'} \{l' \mid l' \leq l \wedge \langle i,j,l' \rangle \in \Sigma_G \cup \{\langle i,j,0 \rangle\}\}$}
if $k = \tau_{ij}(j)$ ***then*** $u_1[i,j] \leftarrow c[i,j]$
if $k = \tau_{ij}(i-1)$ ***then*** $v_0[i,j] \leftarrow c[i,j]$
if $k = \tau_{ij}(i)$ ***then*** $v_1[i,j] \leftarrow c[i,j]$
6. ***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.*}
7. $k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$ {*Steps 7–9 are similar to steps 5–7 of function* F *in Figure 2.*}
8. $H(X_{11}, k_1, k_m),\ H(X_{12}, k_1, k_m),\ H(X_{21}, k_1, k_m),\ H(X_{22}, k_1, k_m)$ {*forward pass*}
9. $H(X_{22}, k_m+1, k_2),\ H(X_{21}, k_m+1, k_2),\ H(X_{12}, k_m+1, k_2),\ H(X_{11}, k_m+1, k_2)$ {*backward pass*}

Fig. 10 C-GEP: A cache-oblivious implementation of GEP (i.e., G in Fig. 1) that works for all f and Σ_G

value of $c[i,j]$ (assuming $i = i_1 = i_2$, $j = j_1 = j_2$ and $k = k_1 = k_2$), it saves $c[i,j]$ to $u_{|1-l|}[i,j]$ provided $k = \tau_{ij}(j-l)$, and to $v_{|1-l|}[i,j]$ provided $k = \tau_{ij}(i-l)$, where $l \in \{0,1\}$. Moreover, during the computation of $c[i,j]$ in the base case, instead of using the current values of $c[i,k]$, $c[k,j]$ and $c[k,k]$ directly from matrix c, it extracts them from $u_{|j>k|}[i,k]$, $v_{|i>k|}[k,j]$ and $u_{|(i>k)\vee(i=k\wedge j>k)|}[k,k]$, respectively. We assume that each of the tests comparing k to $\tau_{ij}(\cdot)$ can be performed in constant time without incurring any additional cache misses.

Cache Complexity and Running Time The number of cache misses incurred by H can be described using the same recurrence relation (1) that was used to describe the cache misses incurred by F in Sect. 2, and hence the cache complexity remains the same, i.e., $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$. Function H also has the same $\mathcal{O}(n^3)$ running time as F, since it only incurs a constant overhead per update applied.

Correctness Since Theorems 2.2 and 2.8 in Sect. 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 column 2 of Table 1 and thus G, follows directly from the following lemma, which can be proved by induction on k using Theorems 2.2 and 2.8, and by observing that H saves all required intermediate values in lines 5–8.

Lemma 4.4 *Immediately before* H *performs the update* $\langle i,j,k \rangle$, *the following hold*: $c[i,j] = \hat{c}_{k-1}(i,j)$, $u_{|j>k|}[i,k] = \hat{c}_{k-|j\leq k|}(i,k)$, $v_{|i>k|}[k,j] = \hat{c}_{k-|i\leq k|}(k,j)$ *and* $u_{|(i>k)\vee(i=k\wedge j>k)|}[k,k]) = \hat{c}_{k-|(i<k)\vee(i=k\wedge j\leq k)|}(k,k)$.

4.3 Reducing the Additional Space

We can reduce the amount of extra space used by H (see Fig. 10) 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 step 8 of this call can be implemented using only $n^2 + n$ extra space. A similar argument applies to the backward pass (step 9) as well.

The first recursive call $\mathrm{H}(X_{11}, k_1, k_2)$ in step 8 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 step 8 can be implemented using 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 an arbitrary f and arbitrary Σ_G at the expense of only $n^2 + n$ extra space. The running time and the cache complexity of this implementation remain $\mathcal{O}(n^3)$ and $\mathcal{O}(\frac{n^3}{B\sqrt{M}})$, respectively.

5 Parallel I-GEP and C-GEP

In this section we consider parallel implementations of I-GEP and C-GEP. We observe that the second and third calls to F in line 5 of the pseudocode for I-GEP given in Fig. 2 can be executed in parallel while maintaining correctness and all properties we have established for I-GEP; similarly the second and third calls to F in line 6 can be performed in parallel. A similar observation holds for lines 11 and 12 of H (see Fig. 10). The resulting parallel code performs a sequence of 6 parallel calls (four calling F or H once and two calling F or H twice), and hence with p processors its parallel execution time is $\mathcal{O}(\frac{n^3}{p} + n^{\log_2 6})$.

In Figs. 11–14 we present a better parallel implementation of I-GEP. This implementation explicitly refers to the different types of functions invoked by I-GEP based on the relative values of the i, j, and k intervals. We assume that $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$, $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$, $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$ and

F(X, U, V, W) {F *can be any of the 9 functions* $(A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, D_4)$ *in column 1 of Figure 12*}
($X \equiv c[i_1..i_2, j_1..j_2]$, $U \equiv c[i_1..i_2, k_1..k_2]$, $V \equiv c[k_1..k_2, j_1..j_2]$ and $W \equiv c[k_1..k_2, k_1..k_2]$. Function F assumes the following: (*a*) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$, for some integer $q \geq 0$.
(*b*) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ 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 13)

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

1. ***if*** no update $\langle i, j, k \rangle \in \Sigma_G$ with $(i, j) \in X$, $(i, k) \in U$, $(k, j) \in V$ exists ***then return*** {*Section 1.1 defines* Σ_G}
2. ***if*** X is a 1×1 matrix ***then*** $X \leftarrow f(X, U, V, W)$ {*Base case*}
   else {*The following function calls are determined from the table in Figure 12. 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.*}
3. ***for*** $r \leftarrow 0$ ***to*** 3 ***do*** {*forward pass*}
   $F_{ij}(X_{ij}, U_{ik}, V_{kj}, W_{kk})$ ***with*** $i \leftarrow 1 + \lfloor \frac{r}{2} \rfloor$, $j \leftarrow 3 + r - 2i$, $k \leftarrow 1$
4. ***for*** $r \leftarrow 3$ ***downto*** 0 ***do*** {*backward pass*}
   $F'_{ij}(X_{ij}, U_{ik}, V_{kj}, W_{kk})$ ***with*** $i \leftarrow 1 + \lfloor \frac{r}{2} \rfloor$, $j \leftarrow 3 + r - 2i$, $k \leftarrow 2$

Fig. 11 Cache-oblivious I-GEP reproduced from Fig. 2, but here F is assumed to be a template function that can be instantiated to any of the 9 functions given in Fig. 13. The recursive calls in lines 6 and 7 are replaced with appropriate instantiations of F which can be determined from Fig. 12

F	F_{11}	F_{12}	F_{21}	F_{22}	F'_{22}	F'_{21}	F'_{12}	F'_{11}
A	A 1	B_1 2	C_1 2	D_1 3	A 4	B_2 5	C_2 5	D_4 6
B_l $(l = 1, 2)$	B_l 1	B_l 1	D_l 2	D_l 2	B_l 3	B_l 3	D_{l+2} 4	D_{l+2} 4
C_l $(l = 1, 2)$	C_l 1	D_{2l-1} 2	C_l 1	D_{2l-1} 2	C_l 3	D_{2l} 4	C_l 3	D_{2l} 4
D_l $(l \in [1, 4])$	D_l 1	D_l 1	D_l 1	D_l 1	D_l 2	D_l 2	D_l 2	D_l 2

Fig. 12 Different instantiations of F (from Fig. 11), and the parallelism in each. Each row lists the instantiations of the functions called in steps 3 and 4 of F when F is instantiated to the function in column 1. Functions in columns 2–9 are executed by F in nondecreasing order of the sequence numbers given inside the *small boxes*. Two functions with the same sequence number can be executed in parallel

Fig. 13 Function specific pre-condition $P(F)$ for F in Fig. 2

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	$i_2 < k_1 \wedge j_2 < k_1$

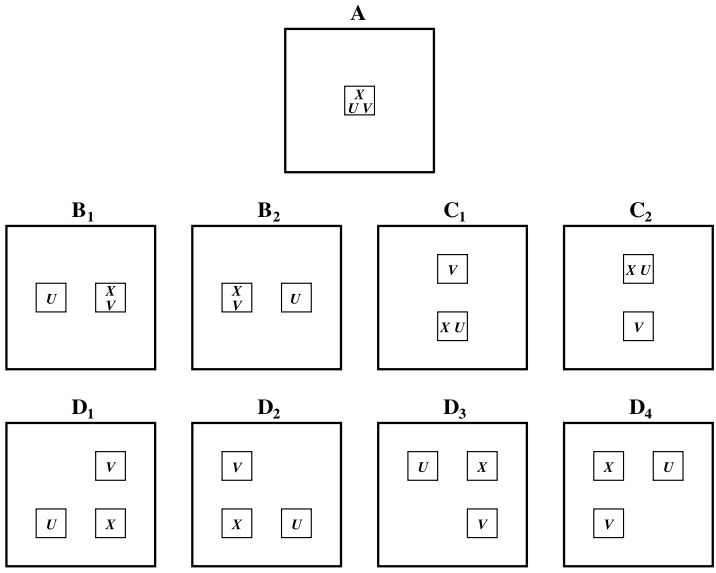

Fig. 14 Relative positions of $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$ w.r.t. $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ assumed by different instantiations of F

$W \equiv c[k_1 \ldots k_2, k_1 \ldots k_2]$, where $[i_1, i_2]$, $[j_1, j_2]$ and $[k_1, k_2]$ are the ranges of i, j and k values, respectively, supplied to F. Then for every entry $c[i, j] \in X$, $c[i, k]$ can be found in U, $c[k, j]$ in V, and $c[k, k]$ can be found in W. Note that only the diagonal entries of W are used.

Input Condition 2.1(a) implies that X, U and V must all be square matrices of the same dimensions. Input condition 2.1(b) requires that each of U and V 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, U and V. For different arrangements of these matrices we give a different name to F. Figure 14 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. Observe that the four types of functions (i.e., A, B_l, C_l and D_l) differ in the amount and type of overlap the input matrices X, U and V have among them. Function A assumes that all three matrices overlap, while function D_l expects completely non-overlapping matrices. Function B_l assumes that only X and V overlap, while C_l assumes overlap only between X and U. Intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, thus leading to better parallelism.

In Fig. 11 we reproduce F from Fig. 2, but replace the recursive calls in lines 6 and 7 of Fig. 2 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 3 of Fig. 11), and by F'_{pq} $(p, q \in [1, 2])$ we denote the same in the backward pass (line 4 of Fig. 11). For each of the nine instantiations of the calling function F, Fig. 12 associates F_{pq} and F'_{pq} $(p, q \in [1, 2])$ with appropriate instantiations, and also identifies the recursive

calls that can be executed in parallel. The initial call is to a function of type A, where the input matrices X, U, V and W completely overlap.

We now analyze the parallel execution time for I-GEP on function A. Let $T_A(n) = T_\infty$ denote the parallel running time when A is invoked with an unbounded number of processors on an $n \times n$ matrix. Let $T_B(n)$, $T_C(n)$ and $T_D(n)$ denote the same for B_i, C_i and D_i, respectively. We will assume for simplicity that $T_A(1) = T_B(1) = T_C(1) = T_D(1) = \mathcal{O}(1)$. Hence we have the following recurrences:

$$T_A(n) \le 2\left(T_A\left(\frac{n}{2}\right) + \max\left\{T_B\left(\frac{n}{2}\right), T_C\left(\frac{n}{2}\right)\right\} + T_D\left(\frac{n}{2}\right)\right) + \mathcal{O}(1),$$

$$T_B(n) \le 2\left(T_B\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right)\right) + \mathcal{O}(1),$$

$$T_C(n) \le 2\left(T_C\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right)\right) + \mathcal{O}(1),$$

$$T_D(n) \le 2T_D\left(\frac{n}{2}\right) + \mathcal{O}(1).$$

Solving these recurrences we obtain $T_\infty = \mathcal{O}(n \log^2 n)$, and thus the following theorem:

Theorem 5.1 *When executed with p processors, multithreaded I-GEP performs $T_1 = \mathcal{O}(n^3)$ work and terminates in $\frac{T_1}{p} + T_\infty = \mathcal{O}(\frac{n^3}{p} + n \log^2 n)$ parallel steps on an $n \times n$ input matrix.*

A similar parallel algorithm with the same parallel time bound applies to C-GEP.

For specific applications of I-GEP, the actual recursive function calls may not take the most general form analyzed above (see Appendix B). For instance, only a subset of the calls are made for Gaussian elimination without pivoting. However, the parallel time bound remains the same as in Theorem 5.1 for this problem as well as for all-pairs shortest paths. On the other hand, for matrix multiplication, we can perform all four recursive calls in each of steps 3 and 4 of Fig. 11 in parallel and hence the parallel time bound is reduced to $\mathcal{O}(\frac{n^3}{p} + n)$. Note that this matrix multiplication computation does not assume associativity of addition.

We have implemented this multithreaded version of I-GEP for Floyd-Warshall's APSP, square matrix multiplication and Gaussian elimination without pivoting using `pthreads`, and we report some experimental results in Sect. 7.3.

5.1 Cache Complexity

We first consider distributed caches, where each processor has its own private cache, and then a shared cache, where all processors share the same cache.

Distributed Caches The following lemma is obtained by considering the schedule that executes each subproblem of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ entirely on a single processor. Since there are $p^{3/2}$ such subproblems, and according to recurrence (1) the sequential execution of each of them on a processor with a private cache of size M and block size B causes $\mathcal{O}(\frac{(n/\sqrt{p})^3}{B\sqrt{M}} + \frac{(n/\sqrt{p})^2}{B})$ cache-misses, the lemma follows.

Lemma 5.2 *There exists a deterministic schedule that incurs only* $\mathcal{O}(\frac{n^3}{B\sqrt{M}} + \sqrt{p} \cdot \frac{n^2}{B})$ *cache misses when executing multithreaded I-GEP on a machine with* p *processors, each with a private cache of size* M *and block size* B.

Using results from [1, 18] (e.g., Theorem 2 and equation (4) of [18]) one can show that I-GEP incurs $\mathcal{O}(\frac{n^3}{B\sqrt{M}} + \frac{(p \cdot n \log^2 n)^{1/3} n^2}{B} + p \cdot n \log^2 n)$ cache misses w.h.p. under the state-of-the-art general-purpose work-stealing Cilk scheduler [17] for distributed caches. The bound in Lemma 5.2 is much better.

Shared Caches Here we consider the case when the p processors share a single cache of size M_p. Part (a) of Lemma 5.3 below is obtained using a general result for shared caches given in [5] for a PDF (*parallel depth first search*) schedule. Better bounds are obtained in part (b) of Lemma 5.3 through the following hybrid depth-first schedule.

Let G denote the computation DAG of I-GEP (i.e., function A), and let $\mathcal{C}(G)$ denote a new DAG obtained from G by contracting each subDAG of G corresponding to a recursive function call on an $r \times r$ submatrix to a supernode, where r is a power of 2 such that $\sqrt{p} \leq r < 2\sqrt{p}$. The subDAG in G corresponding to any supernode v is denoted by $\mathcal{S}(v)$.

Now the hybrid scheduling scheme is applied on G as follows. The scheduler executes the nodes (i.e., supernodes) of $\mathcal{C}(G)$ under 1DF-schedule (sequential depth-first schedule) [5]. However, for each supernode v, the scheduler uses a PDF-schedule with all p processors in order to execute the subDAG $\mathcal{S}(v)$ of G before moving to the next supernode. This leads to the following.

Lemma 5.3 *For* $p \geq 1$ *let multithreaded I-GEP execute* T_p *parallel steps and incur* Q_p *cache misses with* p *processors and on a shared ideal cache of* M_p *blocks. Then under the hybrid depth-first schedule,*

(a) i. $Q_p \leq Q_1$ *if* $M_p \geq M_1 + \Theta(p)$,
ii. *If* $M_1 = M_p$ *then* $Q_p = O(Q_1)$ *provided* $p = O(M_p)$.

(b) $T_p = \mathcal{O}(\frac{n^3}{p} + n \log^2 n)$.

Proof (a.i) Since for each supernode v in $\mathcal{C}(G)$ the subDAG $\mathcal{S}(v)$ in G accesses at most $\Theta(r^2)$ locations of the input matrix, when executing $\mathcal{S}(v)$ under PDF-schedule no more than $\Theta(r^2) = \Theta(p)$ nodes can become *premature* [5] simultaneously. Since supernodes are executed one at a time, having $M_p \geq M_1 + \Theta(p)$ ensures that there is always enough space in the shared cache to accommodate the premature nodes without ever incurring any extra cache misses. Therefore, $Q_p \leq Q_1$.

(a.ii) Suppose $M_p = M_1 = M$. Since the hybrid schedule never creates more than $\Theta(p)$ simultaneous premature nodes (see part (a)), we can set aside $\Theta(p)$ locations in the shared cache for holding the premature nodes. The effective cache size thus reduces to $M - \Theta(p)$, and assuming $M - \Theta(p) = \Omega(M) \Rightarrow p = \mathcal{O}(M)$, the number of cache misses incurred by multithreaded I-GEP is $Q_p \leq \mathcal{O}(\frac{n^3}{B\sqrt{M - \Theta(p)}}) = \mathcal{O}(Q_1)$.

(b) The number of parallel steps for PDF-schedule follows from the results in [5]. Therefore, we restrict our attention to the hybrid scheduler below.

Table 2 Properties of supernodes in $\mathcal{C}(G)$: for a given supernode v, $\mathcal{F}(v)$ denotes the recursive function represented by subDAG $\mathcal{S}(v)$ in G while $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ denote the number of supernodes in $\mathcal{C}(G)$ representing $\mathcal{F}(v)$ and the number of parallel steps required to execute $\mathcal{F}(v)$, respectively

$\mathcal{F}(v)$	A	B_i $(i = 1, 2)$	C_i $(i = 1, 2)$	D_i $(i \in [1, 4])$
$n(\mathcal{F}(v))$	$\frac{n}{r}$	$(\frac{n}{r})^2 - \frac{n}{r}$	$(\frac{n}{r})^2 - \frac{n}{r}$	$(\frac{n}{r})^3 - 2(\frac{n}{r})^2 + \frac{n}{r}$
$s(\mathcal{F}(v))$	$\mathcal{O}(r \log^2 r)$	$\mathcal{O}(r \log r)$	$\mathcal{O}(r \log r)$	$\mathcal{O}(r)$

Observe that G has $\Theta(n^3)$ nodes, and each subDAG in G corresponding to supernodes in $\mathcal{C}(G)$ has $\Theta(r^3)$ nodes. Therefore, $\mathcal{C}(G)$ has only $\Theta((\frac{n}{r})^3)$ nodes.

For a given supernode v, let $\mathcal{F}(v)$ denote the recursive function represented by subDAG $\mathcal{S}(v)$ in G. Let $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ denote the number of supernodes in $\mathcal{C}(G)$ representing $\mathcal{F}(v)$ and the number of parallel steps required to execute $\mathcal{F}(v)$, respectively. The values of $n(\mathcal{F}(v))$ and $s(\mathcal{F}(v))$ for $\mathcal{F}(v) \in \{A, B_i, C_i, D_i\}$ are tabulated in Table 2 (the calculations are not difficult and are omitted for brevity). Therefore, the number of parallel steps required to execute all supernodes is

$$\sum_{\mathcal{F}(v) \in \{A, B_i, C_i, D_i\}} n\big(\mathcal{F}(v)\big) \times s\big(\mathcal{F}(v)\big) = \mathcal{O}\left(\frac{n^3}{r^2} + \frac{n^2}{r} \log r + n \log r\right)$$

$$= \mathcal{O}\left(\frac{n^3}{p} + n \log^2 n\right)$$

(since $p \leq n^2$).

Since $\mathcal{C}(G)$ has only $\Theta((\frac{n}{r})^3)$ nodes, the number of steps required to execute $\mathcal{C}(G)$ under 1DF-schedule is $\mathcal{O}((\frac{n}{r})^3) = \mathcal{O}(\frac{n^3}{p\sqrt{p}})$. Therefore, the total number of parallel steps required to execute multithreaded I-GEP under the hybrid depth-first schedule is

$$\mathcal{O}\left(\frac{n^3}{p} + \frac{n^3}{p\sqrt{p}} + n \log^2 n\right) = \mathcal{O}\left(\frac{n^3}{p} + n \log^2 n\right) = \mathcal{O}\left(\frac{T_1}{p} + T_\infty\right)$$

since $T_1 = n^3$ and $T_\infty = \mathcal{O}(n \log^2 n)$. □

When executing I-GEP under the state-of-the-art PDF-scheduler [5] for shared caches, $Q_p \leq Q_1$ provided $M_p \geq M_1 + \Theta(p \cdot T_\infty(n)) = M_1 + \Theta(pn \log^2 n)$, which is much weaker than the bound given above for the hybrid depth-first scheduler.

In a recent work [4] we have introduced the *multicore-cache model* that reflects the reality that multicore processors have both per-processor private (L_1) caches and a large shared (L_2) cache on chip, and presented an online scheduler for cache-efficient execution of a broad class of parallel divide-and-conquer algorithms (that includes I-GEP) on this model. The new scheduler is competitive with the standard sequential scheduler in the following sense. Given any dynamically unfolding computation DAG from this class of algorithms, the cache complexity on the multicore-cache model under our new scheduler is within a constant factor of the sequential cache complexity for both L_1 and L_2, while the time complexity is within a constant factor

of the sequential time complexity divided by the number of processors p. In a more recent work [10] we have shown that for both shared and distributed caches the depth of any GEP computation can be improved to $\mathcal{O}(n)$ while still matching its optimal sequential cache complexity by choosing tile sizes that depend only on the number of cores/processors and thus still remaining cache-oblivious. This is the maximum parallelism achievable when staying within the GEP framework. There is a well-known purely parallel NC algorithm with lower depth for matrix multiplication (a specific GEP problem), but that algorithm uses extra space.

Very recently [11] we have extended the 3-level multicore-cache model described in [4] to the *hierarchical multi-level caching model* (*HM*) for multicores.[1] The HM model consists of a collection of cores sharing an arbitrarily large main memory through a hierarchy of caches of finite but increasing sizes that are successively shared by larger groups of cores. We have also introduced the notion of *multicore-oblivious* (*MO*) algorithms for the HM model, i.e., algorithms that make no mention of the number of cores and the cache parameters. For improved performance, however, an MO algorithm is allowed to provide advice or "hints" to the run-time scheduler through a small set of instructions on how to schedule the parallel tasks it spawns. We have shown that I-GEP can be solved multicore-obliviously on the HM model in time proportional to its sequential time complexity divided by the number of cores, while still remaining within a constant factor of its optimal sequential cache-complexity at each level of the cache hierarchy.

6 Cache-Oblivious GEP and Compiler Optimization

'Tiling' is a powerful loop transformation technique employed by optimizing compilers for improving temporal locality in nested loops [25]. 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 15(b) shows a tiled version of the triply nested loop shown in Fig. 15(a) that occurs in matrix multiplication [25].

Cache performance of a tiled loop depends on the chosen tile size T. The 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 [25, 32]. 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 Fig. 2 and the C-GEP code given in Fig. 10 can be viewed as cache-oblivious versions of tiling for the triply nested loops of the form as shown in Fig. 1. The nested loop in Fig. 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

[1] We briefly described the HM model in [10].

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

(a)

```
1. for i ← 1 to n by T do                                   {T is the tile size}
2.     for j ← 1 to n by T do
3.         for k ← 1 to n by T do
4.             for i' ← i to min(i + T − 1, n) do
5.                 for j' ← j to min(j + T − 1, n) do
6.                     for k' ← k to min(k + T − 1, n) do
7.                         c[i', j'] ← c[i', j'] + a[i', k'] × b[k', j']
```

(b)

Fig. 15 (**a**) Traditional matrix multiplication algorithm, (**b**) tiled version of the matrix multiplication algorithm of part (a) [25]

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 Fig. 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 Sect. 3.3), 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 Sect. 7) suggest that I-GEP outperforms C-GEP in both in-core and out-of-core computations.

We will now look at some general conditions under which I-GEP is a legal transformation for a given GEP code. Consider the general GEP code in Fig. 1. Recall the definition of π from Sect. 2, and the definition of τ_{ij} from Sect. 4.2 (Definition 4.1). The following lemma follows from Observations 4.2 and 4.3 in Sect. 4.2, 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 6.1 *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 Fig.* 1.

7 Experimental Results

We ran our experiments on the three architectures listed in Table 3. Each machine can perform at most two double precision floating point operations per clock cycle.

Table 3 Machines used for experiments. All block sizes (B) are 64 bytes

Model	Processors/ Cores	Speed	Peak GFLOPS (per core)	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	6.12	8 KB (4-way)	512 KB (8-way)	4 GB
AMD Opteron 250	2	2.4 GHz	4.8	64 KB (2-way)	1 MB (8-way)	4 GB
AMD Opteron 875	8 (4 dual-cores)	2.2 GHz	4.4	64 KB (2-way)	1 MB (16-way)	32 GB

The *peak performance* of each machine is thus measured in terms of *GFLOPS* (or Giga FLoating point Operations Per Second) which is two times the clock speed of the machine in GHz.

The Intel P4 Xeon machine is also equipped with a 73.5 GB Fujitsu MAP3735NC hard disk (10K RPM, 4.5 ms avg. seek time, 64.1 to 107.86 MB/s data transfer rate) [19]. Our out-of-core experiments were run on this machine. All machines run Ubuntu Linux 5.10 "Breezy Badger". Each machine was exclusively used for experiments.

We used the *Cachegrind* profiler [29] for simulating cache effects. For in-core computations all algorithms were implemented in C using a uniform programming style and compiled using *gcc* 3.3.4 with optimization parameter -O3 and limited loop unrolling.

We summarize our results below.

7.1 GEP, I-GEP and C-GEP for APSP

In this section we present experimental results comparing GEP, I-GEP and C-GEP implementations of Floyd-Warshall's APSP algorithm [15, 31] for both *in-core* and *out-of-core* computations.

Out-of-Core Computation For out-of-core computations we implemented GEP, I-GEP and C-GEP in C++, and compiled using g++ 3.3.4 compiler with optimization level -O3 and STXXL software library version 0.9 [14] for external memory accesses. The STXXL library maintains its own fully associative cache in RAM with pages from the disk, and allows users set the size of the cache (M) and the block transfer size (B) manually. We compiled STXXL with DIRECT-I/O turned on so that the OS does not cache data from hard disk.

When the computation is out-of-core I/O wait times dominate computation times. In Fig. 16(a) we keep n and B fixed and vary M. We observe that M has almost no effect on the I/O wait time of GEP while that of both I-GEP and C-GEP decrease as M increases. This result is consistent with theoretical predictions since the cache complexity of GEP is independent of M and that of I-GEP and C-GEP vary inversely with $\sqrt{M}$. In general, the I/O wait time of GEP is several hundred times more than that of I-GEP and C-GEP; for example, when only half of the input matrix fits in internal memory GEP waits 500 times more than I-GEP, and almost 180 times more than both variants of C-GEP. In Fig. 16(b) we keep n and M fixed, and vary M/B

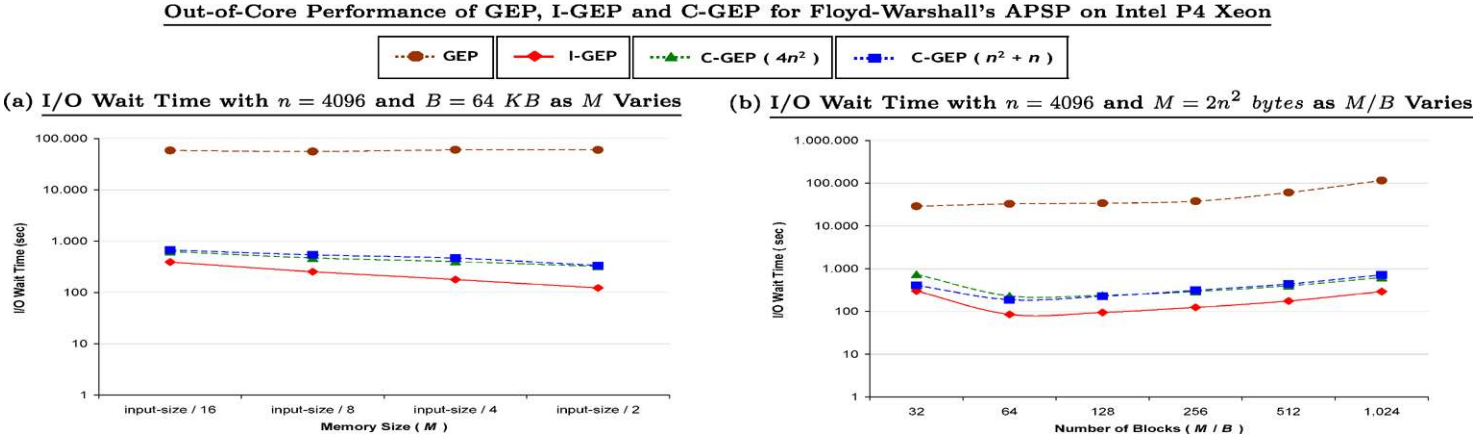

Fig. 16 Comparison of out-of-core performance of GEP, I-GEP and C-GEP on Intel Pentium 4 Xeon with a fast hard disk (10K RPM, 4.5 ms avg. seek time, 64 to 107 MB/s transfer rate)

(by varying B), and observe that in general, I/O wait times increase linearly with the increase of M/B. The theoretical I/O complexities of all these algorithms vary inversely with B, which explains the observed trend. However, when M/B is small, the number of page faults increases which affects the cache performance of all algorithms.

In-Core Computation In Fig. 17 we plot the performance of GEP and I-GEP on both Intel Xeon and AMD Opteron 250. We optimized I-GEP as described in Sect. 7.2. On Intel Xeon I-GEP runs around 5 times faster than GEP while on AMD Opteron it runs around 4 times faster.

In Fig. 18 we plot the relative performance of I-GEP and C-GEP on Intel Xeon. As expected, both versions of C-GEP run slower and incur more L2 misses than I-GEP, since they perform more write operations. However, this overhead diminishes as n becomes larger. The $(n^2 + n)$-space variant of C-GEP performs slightly worse than the $4n^2$-space variant which we believe is due to the fact that the $(n^2 + n)$-space C-GEP needs to perform more initializations and re-initializations of the temporary matrices (i.e., u_0, u_1, v_0 and v_1) compared to the $4n^2$-space C-GEP.

7.2 Comparison of I-GEP and BLAS Routines

We compared the performance of our I-GEP code for square matrix multiplication and Gaussian elimination without pivoting on double precision floats with algorithms based on highly fine-tuned *Basic Linear Algebra Subprograms* (BLAS). We applied the following major optimizations on our basic I-GEP routines before the comparison:

- In order to reduce the overhead of recursion we solve the problem directly using a GEP-like iterative kernel as the input submatrix X received by the recursive functions becomes very small. We call the size of X at which we switch to the iterative kernel the *base-size*. On each machine the best value of *base-size*, i.e., for which the implementation ran the fastest, was determined empirically. On Intel Xeon it is 128×128 and on AMD Opteron it is 64×64.
- We use SSE2 ("Streaming SIMD Extension 2") instructions for increased throughput.
- For Gaussian elimination without pivoting we use a standard technique for reducing the number of division operations to $o(n^3)$ (i.e., by moving the division operations out of the innermost loops).
- We use the *bit-interleaved* layout (e.g., see [7, 16]) for reduced TLB misses. More specifically, we arrange the base case size blocks in the bit-interleaved layout with data within the blocks arranged in row-major layout. We include the cost of converting to and from this format in the time bounds.

In Fig. 19 we show the performance of square matrix multiplication on AMD Opteron 250 with GEP (an optimized version), I-GEP and *Native BLAS*, i.e., BLAS generated for the native machine using the automated empirical optimization tool ATLAS [28]. We report the results in '% peak', e.g., an algorithm executing at 'x% of peak' spends x% of its execution time performing floating point operations while

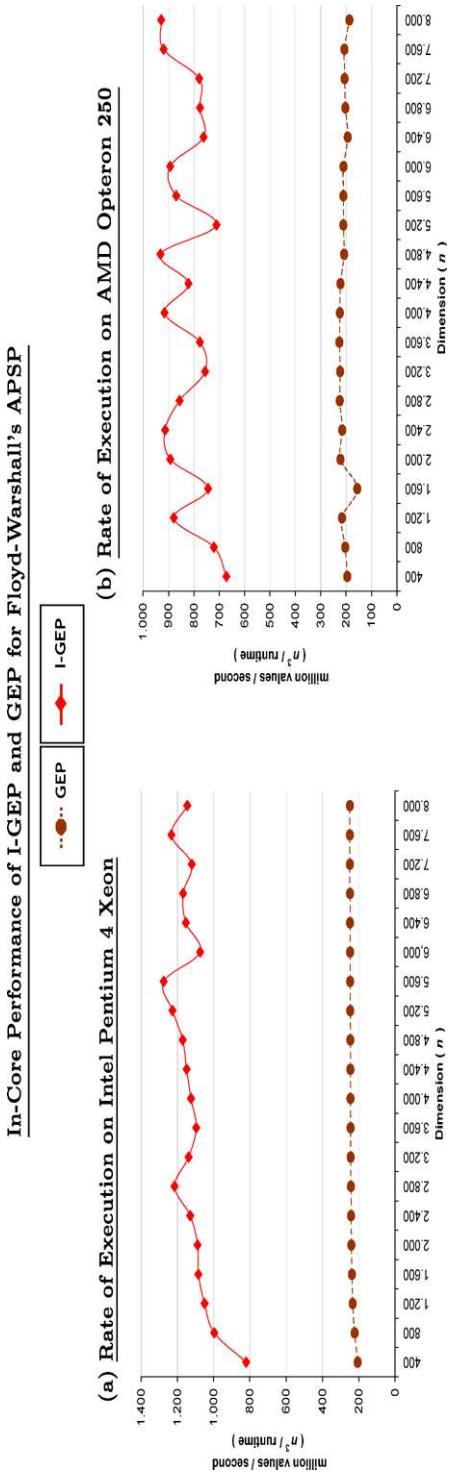

Fig. 17 Comparison of I-GEP and GEP on Intel Xeon and AMD Opteron for computing Floyd-Warshall's all-pairs shortest paths. Both machines have two processors, but only one was used

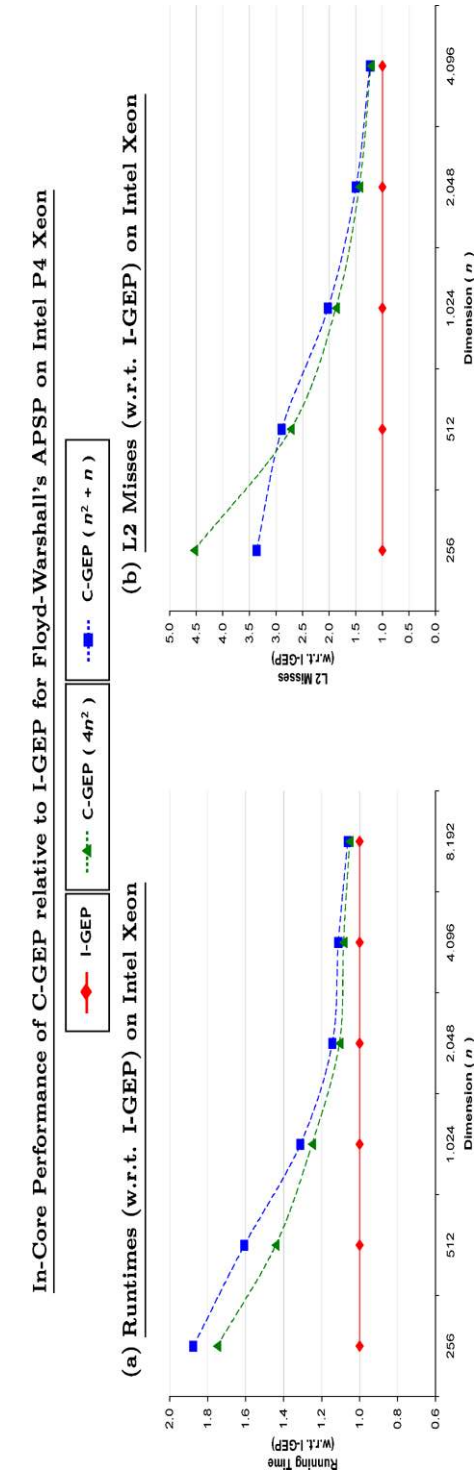

Fig. 18 Comparison of in-core performance of I-GEP and C-GEP on Intel Pentium 4 Xeon

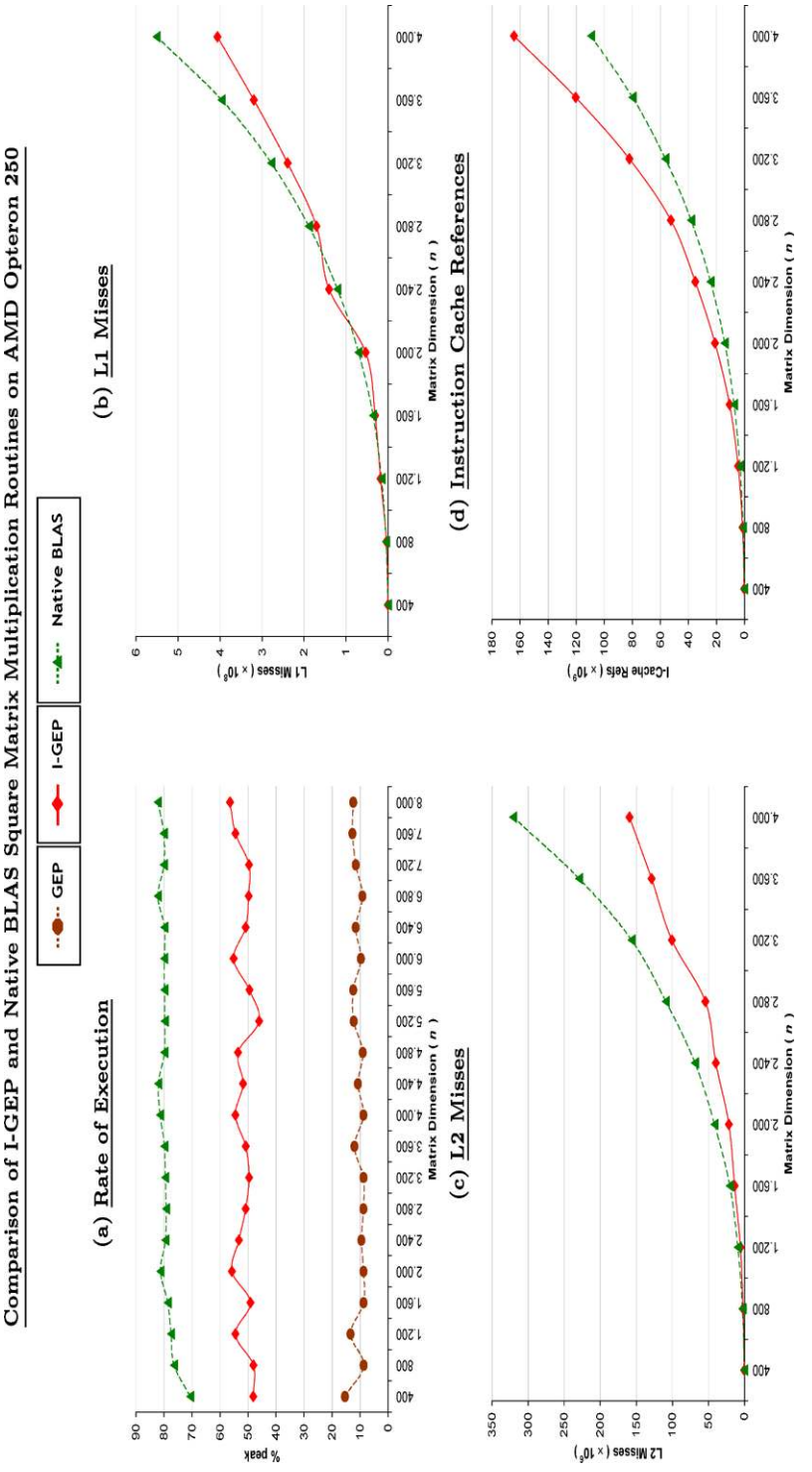

Fig. 19 Comparison of I-GEP and native BLAS square matrix multiplication routines on a 2.4 GHz dual processor AMD Opteron 250 (only one processor was used)

remaining time is spent in other overheads including recursion, loops, cache misses, etc. From the plots in Fig. 19 we observe:

- Native BLAS executes at 78–83% of peak while I-GEP executes at 50–56% of peak. Traditional GEP reaches only 9–13% of peak. The GotoBLAS [20] which is usually the fastest BLAS available for most machines (not shown in the plots) runs at 85–88% of peak.
- I-GEP incurs fewer L1 and L2 misses than native BLAS.
- I-GEP executes more instructions than native BLAS.

We obtained similar results on Intel P4 Xeon.

In Fig. 20 we plot the performance of Gaussian elimination without pivoting using GEP, I-GEP and GotoBLAS [20] on both Intel Xeon and AMD Opteron 250. (Recall that GotoBLAS is the fastest BLAS available for most machines.) We used the LU decomposition (without pivoting) routine available in FLAME [21] to implement Gaussian elimination without pivoting using GotoBLAS. On both machines GotoBLAS executes at around 75–83% of peak while I-GEP runs at around 45–55% of peak. Traditional GEP reaches only 7–9% of peak.

Recursive square matrix multiplication using an iterative base case similar to our implementations is studied in [34]. The experimental results in [34] report performance level of only about 35% of peak for Intel P4 Xeon which is significantly lower than what we obtain for the same machine (50–58%). We conjecture that our improved performance is partly due to our use of SSE2 instructions, especially since [34] obtained performance levels of 60–75% for SUN UltraSPARC IIIi, IBM Power 5 and Intel Itanium 2 using FMA instructions. These latter results nicely complement our results for Intel P4 Xeon and AMD Opteron and further suggest that reasonable performance levels can be reached for square matrix multiplication on different architectures using relatively simple code that does not directly depend on cache parameters.

Both our implementations and the ones in [34] experimentally determined the best base-size since the overhead of recursion becomes excessive if the recursion extends all the way down to size 1. In [34] this is viewed as not being purely cache-oblivious; however we consider the fine-tuning of the base-size in recursive algorithms to be a standard optimization during implementation.

7.3 Multithreaded I-GEP

We implemented multithreaded I-GEP using the standard `pthreads` library. We varied the number of concurrent threads from 1 to 8 on a 4×dual-core AMD Opteron 875 (with private L1 and L2 caches for each core) and used I-GEP to perform matrix multiplication, Gaussian elimination without pivoting and Floyd-Warshall's APSP on input 5000×5000 matrices. We used the default scheduling policy on Linux. In Fig. 21 we plot the speed-up factors achieved by multithreaded I-GEP over its unthreaded version as the number of concurrent threads is increased. For matrix multiplication and Gaussian elimination without pivoting we also plot the speed-up factors achieved by multithreaded GotoBLAS.

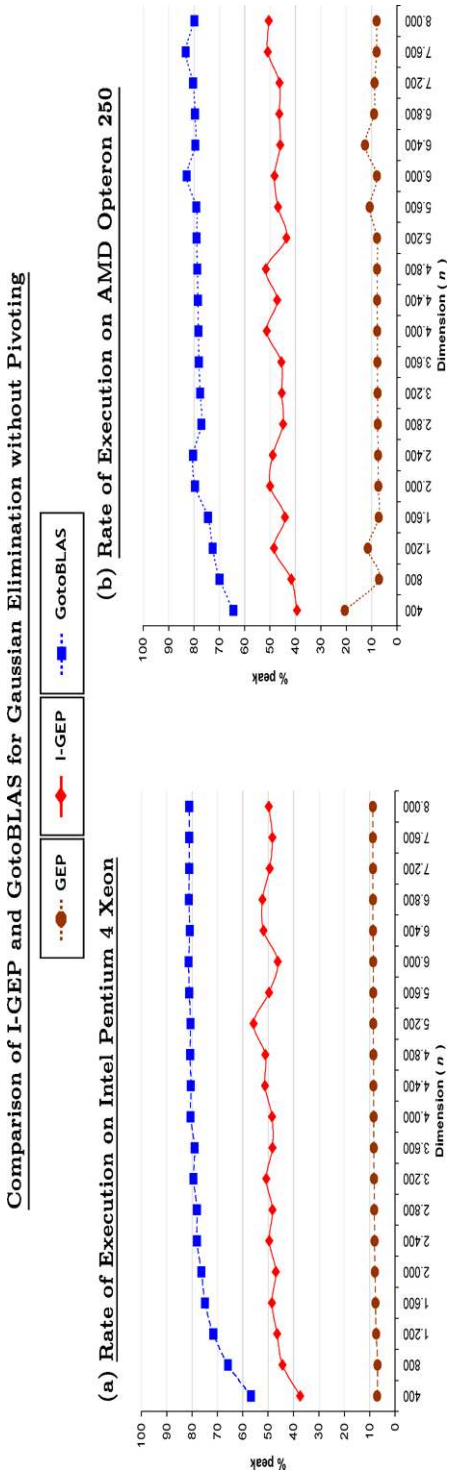

Fig. 20 Comparison of I-GEP and GotoBLAS on Intel Xeon and AMD Opteron for performing Gaussian elimination without pivoting. Both machines have two processors, but only one was used

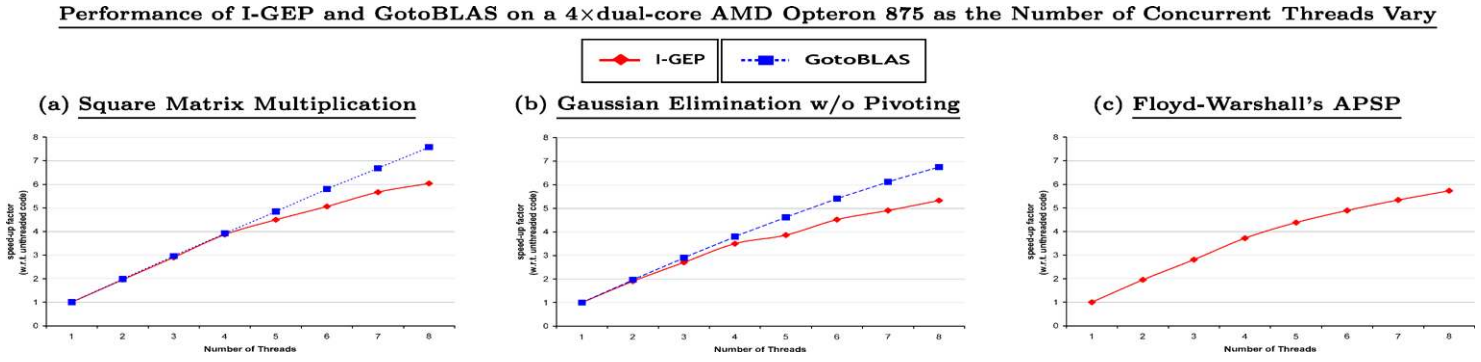

Fig. 21 Performance of I-GEP on a 4×dual-core AMD Opteron 875 for square matrix multiplication, Gaussian elimination w/o pivoting and Floyd-Warshall's all-pairs shortest paths on 5000 × 5000 matrices as number of concurrent threads is varied. For matrix multiplication and Gaussian elimination w/o pivoting we compare IGEPs performance with that of GotoBLAS

For square matrix multiplication I-GEP speeds up by a factor of 6 when the number of concurrent threads increases from 1 to 8, while for Gaussian elimination without pivoting and Floyd-Warshall's APSP the speed-up factors are smaller, i.e., 5.33 and 5.73, respectively. As mentioned in Sect. 5, I-GEP for matrix multiplication has more parallelism than I-GEP for Gaussian elimination without pivoting and Floyd-Warshall's APSP, which could explain the better speed-up factor for matrix multiplication. We also observe from Fig. 21 that the I-GEP's performance gain with each additional thread drops when the number of threads exceeds 4. This happens because the two cores in each Opteron 875 processor share one memory controller. Since there are 4 processors, when the number of threads is at most 4, the scheduler can assign each thread to a different processor so that each thread can utilize the full memory controller throughput of the processor it is assigned to. However, when the number of threads exceeds 4, both cores of one or more processors will have threads assigned to them, and thus the memory controller of each such processor will be shared by at least two threads. As a result performance gain drops when a memory controller can not keep up with the combined memory bandwidth requirements of the threads it is serving. This situation can be improved by assigning concurrent threads that share input data to the same processor whenever two or more such threads must be executed by the same processor.

GotoBLAS scales up better than our current implementation of multithreaded I-GEP as the number of threads increases, and with 8 threads it reaches speed-up factors of 7.6 and 6.75 for matrix multiplication (Fig. 21(a)) and Gaussian elimination without pivoting (Fig. 21(b)), respectively. Though in the first case GotoBLAS scales up almost linearly up to 8 threads, in the second case its performance gain drops more noticeably after 4 threads.

We consider it likely that the performance of multithreaded I-GEP can be improved by using the scheduling policies described in Sect. 5 instead of the default policy on Linux.

7.4 Summary of Experimental Results

We draw the following conclusions from our results:

- In our experiments I-GEP always outperformed both variants of C-GEP (see Sect. 7.1). The $4n^2$-space variant of C-GEP almost always outperformed the $(n^2 + n)$-space variant, and it is also easier to implement. Therefore, if disk space is not at a premium, the $4n^2$-space C-GEP should be used instead of the $(n^2 + n)$-space variant, and I-GEP is preferable to both variants of C-GEP whenever applicable.
- When the computation is in-core, I-GEP runs about 5–6 times faster than even some reasonably optimized versions of GEP. It has been reported in [26] that I-GEP runs slightly slower than GEP on Intel P4 Xeon for Floyd-Warshall's APSP when the prefetchers are turned on. We believe that we get dramatically better results for I-GEP in part because unlike [26] we arrange the entries of each base-case submatrix in a prefetcher-friendly layout, i.e., in row-major order (see Sect. 7.2). Note that we include the cost of converting to and from this layout in the time bounds we report.

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

(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])
```

Fig. 22 Two simple variants of GEP (Fig. 1) obtained by rearranging the ***for*** loops

- BLAS routines run about 1.5 times faster than I-GEP. However, I-GEP is cache-oblivious and is implemented in a high level language, while BLAS routines are cache-aware and employ numerous low-level machine-specific optimizations in assembly language. The cache-miss results in Sect. 7.2 indicate that the cache performance of I-GEP is at least as good as that of native BLAS. Hence the performance gain of native BLAS over I-GEP is most likely due to optimizations other than cache-optimizations.
- Our I-GEP/C-GEP code for in-core computations can be used for out-of-core computations without any changes, while BLAS is optimized for in-core computations only.
- I-GEP/C-GEP can be parallelized very easily, and speeds up reasonably well as the number of processors (i.e., concurrent threads) increases. However, current systems offer very limited flexibility in scheduling tasks to processors, and we believe that performance of multithreaded I-GEP can be improved further if better scheduling policies are used.

8 Conclusion

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 GEP 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 GEP 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 [25]. However, several important open questions still exist. For example, can we solve GEP in its full generality without using any extra space, or at least using $o(n^2)$ space? Also can we obtain general cache-oblivious frameworks for other variants of GEP (for example, for those shown in Fig. 22).

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

Appendix A: Formal Definitions of δ and π

In Sect. 2 we defined functions π and δ (see Definition 2.5) 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 2.5(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 2.7(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) $(z, z) \in X_{11}$ and $(x, y) \notin X_{11}$: $X_{11} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^{q-1}$ by definition.
  (ii) $(z, z) \in X_{22}$ and $(x, y) \notin X_{22}$: $X_{22} \equiv S(x, y, z)$ and $\delta(x, y, z) = 2^q$ by definition.
  (iii) $(z, z) \in X_{11}$ and $(x, y) \in X_{11}$: $S(x, y, z) \in X_{11}$, and compute $\delta(x, y, z)$ recursively from X_{11}.
  (iv) $(z, z) \in X_{22}$ 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 $y \neq z$.

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

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

Then

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

Now we can formally define function $\delta : [1, 2^q] \times [1, 2^q] \times [1, 2^q] \to [0, 2^q]$ 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 (A.1).

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

From Definition 2.5(b), we have that function $\pi : [1, 2^q] \times [1, 2^q] \to [0, 2^q]$ 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:

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

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

Appendix B: Static Pruning of I-GEP

In Sect. 2, the test in line 1 of Fig. 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. 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 $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$. Input conditions 2.1 imply nine possible arrangements (i.e., relative positions) of X, U and V. For each of these arrangement we give a different name (A, B_1, B_2, C_1, C_2, D_1, D_2, D_3 or D_4) to F (see Fig. 14) which we call an instantiation of F. Given an instantiation F′ of F, Fig. 13 expresses the corresponding arrangement of X, U and V as a relationship $P(F')$

among the indices i_1, i_2, j_1, j_2, k_1 and k_2. Figure 11 reproduces F from Fig. 2, but replaces the recursive calls in lines 6 and 7 of Fig. 2 with instantiations of F (in lines 3 and 4 of Fig. 11). A given computation need not necessarily make all recursive calls in lines 3 and 4. Whether a specific recursive call to a function F′ (say) will be made or not depends on $P(F')$ (see Fig. 13) 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 $P(C_2)$. 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.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 1–12. ACM, New York (2000)
2. Aggarwal, A., Vitter, J.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
3. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
4. Blelloch, G., Chowdhury, R., Gibbons, P., Ramachandran, V., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, pp. 501–510 (2008)
5. Blelloch, G., Gibbons, P.: Effectively sharing a cache among threads. In: Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures, Barcelona, Spain, pp. 235–244 (2004)
6. Blumofe, R., Frigo, M., Joerg, C., Leiserson, C., Randall, K.: An analysis of DAG-consistent distributed shared-memory algorithms. In: Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures, pp. 297–308 (1996)
7. Chatterjee, S., Lebeck, A., Patnala, P., Thotethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures, pp. 222–231 (1999)
8. Chowdhury, R., Ramachandran, V.: Cache-oblivious dynamic programming. In: Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms, Miami, Florida, pp. 591–600 (2006)
9. Chowdhury, R., Ramachandran, V.: The cache-oblivious Gaussian Elimination Paradigm: Theoretical framework, parallelization and experimental evaluation. In: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, pp. 71–80 (2007)
10. Chowdhury, R., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, pp. 207–216 (2008)
11. Chowdhury, R., Silvestri, F., Blakeley, B., Ramachandran, V.: Oblivious algorithms for multicores and network of processors. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium, Atlanta, Georgia, April 2010
12. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
13. D'Alberto, P., Nicolau, A.: R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica **47**(2), 203–213 (2007)
14. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: Proceedings of the 13th Annual European Symposium on Algorithms. LNCS, vol. 1004, pp. 640–651. Springer, Berlin (2005)
15. Floyd, R.: Algorithm 97 (SHORTEST PATH). Commun. ACM **5**(6), 345 (1962)
16. Frigo, M., Leiserson, C., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pp. 285–297 (1999)

17. Frigo, M., Leiserson, C., Randall, K.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, pp. 212–223 (1998)
18. Frigo, M., Strumpen, V.: The cache complexity of multithreaded cache oblivious algorithms. In: Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, pp. 271–280 (2006)
19. Fujitsu MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP disk drives product/maintenance manual
20. Goto, K.: GotoBLAS (2005). http://www.tacc.utexas.edu/resources/software
21. Gunnels, J., Gustavson, F., Henry, G., van de Geijn, R.: FLAME: Formal linear algebra methods environment. ACM Trans. Math. Softw. **27**(4), 422–455 (2001)
22. Hong, J., Kung, H.: I/O complexity: the red-blue pebble game. In: Proceedings of the 13th Annual ACM Symposium on Theory of Computing, pp. 326–333 (1981)
23. Iversion, K.: A Programming Language. Wiley, New York (1962)
24. Knuth, D.: Two notes on notation. Am. Math. Mon. **99**, 403–422 (1992)
25. Muchnick, S.: Advanced Compiler Design & Implementation. Morgan Kaufmann, San Mateo (1997)
26. Pan, S., Cherng, C., Dick, K., Ladner, R.: Algorithms to take advantage of hardware prefetching. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, pp. 91–98 (2007)
27. Park, J., Penner, M., Prasanna, V.: Optimizing graph algorithms for improved cache performance. IEEE Trans. Parallel Distrib. Syst. **15**(9), 769–782 (2004)
28. Powell, D., Allison, L., Dix, T.: Automated empirical optimization of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001). http://math-atlas.sourceforge.net
29. Seward, J., Nethercote, N.: Valgrind (debugging and profiling tool for x86-Linux programs). http://valgrind.kde.org/index.html
30. Toledo, S.: Locality of reference in LU decomposition with partial pivoting. SIAM J. Matrix Anal. Appl. **18**(4), 1065–1081 (1997)
31. Warshall, S.: A theorem on boolean matrices. J. ACM **9**(1), 11–12 (1962)
32. Wolf, M., Lam, M.: A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp. 30–44 (1991)
33. Womble, D., Greenberg, D., Wheat, S., Riesen, R.: Beyond core: Making parallel computer I/O practical. In: Proceedings of the 1993 DAGS/PC Symposium, pp. 56–63 (1993)
34. Yotov, K., Roeder, T., Pingali, K., Gunnels, J., Gustavson, F.: An experimental comparison of cache-oblivious and cache-aware programs. In: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, pp. 93–104 (2007)