The Cache-oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation [*]

Rezaul Alam Chowdhury
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
shaikat@cs.utexas.edu

Vijaya Ramachandran
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
vlr@cs.utexas.edu

ABSTRACT

The Gaussian Elimination Paradigm (GEP) was introduced by the authors in [6] to represent the triply-nested loop computation that occurs in several important algorithms including Gaussian elimination without pivoting and Floyd-Warshall's all-pairs shortest paths algorithm. An efficient cache-oblivious algorithm for these instances of GEP was presented in [6]. In this paper we establish several important properties of this cache-oblivious framework, and extend the framework to solve GEP in its full generality within the same time and I/O bounds. We then analyze a parallel implementation of the framework and its caching performance for both shared and distributed caches. We present extensive experimental results for both in-core and out-of-core performance of our algorithms. We consider both sequential and parallel implementations of our algorithms, 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.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; F.3.3 [**Studies of Program Constructs**]: Program and recursion schemes

General Terms

Algorithms, Theory, Performance, Experimentation

Keywords

Cache-oblivious algorithm, Gaussian elimination, all-pairs shortest path, matrix multiplication, tiling

1. INTRODUCTION

Cache-efficient algorithms improve execution time by exploiting data parallelism inherent in the transfer of blocks of useful data between adjacent memory levels. By increasing locality in their memory access patterns, these algorithms try to keep the number of block transfers small. The *ideal-cache model* [11] is a further refinement that enables the development of system-independent cache-efficient algorithms that simultaneously adapt to all levels of a multi-level memory hierarchy. This leads to fuller use of data parallelism and also produces portable code. The ideal-cache model represents the memory hierarchy with two memory levels — a cache of size M with an optimal offline cache replacement policy and an unlimited main memory partitioned into blocks of size B. The *cache complexity* of an algorithm is the number of *I/Os* (i.e., block transfers) performed between these two levels. Algorithms designed for this model are known as *cache-oblivious* algorithms since they do not use knowledge of M and B.

In [6] we introduced the *Gaussian Elimination Paradigm* (*GEP*) for triply-nested loop computations similar to that in Gaussian elimination without pivoting. Traditional GEP implementations run in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and incur $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/Os. In [6] we presented a framework for in-place cache-oblivious execution of several important algorithms in GEP including Gaussian elimination and LU-decomposition without pivoting, all-pairs shortest paths and matrix multiplication; we also adapted this framework to solve important non-GEP problems such as sequence alignment with arbitrary gap function, and a class of dynamic programs termed as 'simple-DP' [5]. This cache-oblivious implementation, which we call here I-GEP, incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os while still running in $\mathcal{O}\left(n^3\right)$ time and without using any extra space. However, there exist some examples of GEP triply-nested loops for which I-GEP fails to produce correct results.

In this paper we establish several properties of I-GEP. We then build on these results to derive C-GEP, which has the same time and I/O bounds as I-GEP, but unlike I-GEP is a provably correct cache-oblivious implementation of GEP in its full generality. C-GEP uses a modest amount of extra space. We present experimental results that show that both I-GEP and C-GEP significantly outperform GEP especially in out-of-core computations, although improvements in computation time are already realized during in-core computations. We present a parallel version of I-GEP, and analyze its parallel running time as well as its caching performance under both distributed and shared caches, and we present some experimental results on our `pthreads` implementation

[*]This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609.

of parallel I-GEP. Finally we present experimental results comparing performance of I-GEP with that of highly optimized cache-aware BLAS routines for matrix multiplication and Gaussian elimination without pivoting. Our results show that our implementation of I-GEP runs moderately slower than native BLAS; however, I-GEP performs fewer number of cache misses, is much simpler to code, easily supports `pthreads` and is portable across machines.

Two papers related to our work appear in these proceedings [3, 26]. Experimental studies similar to some of our experimental work are reported in [26] and are discussed in Section 4.2. The parallel performance of cache-aware dense linear algebra matrix computations on SMP architectures is considered in [3].

Organization of the Paper. In the rest of this section we summarize the earlier results from [6] on I-GEP, and related work. In Section 2 we derive some key properties of I-GEP, and use these to derive the fully general C-GEP. We also briefly address the potential application of I-GEP and C-GEP in optimizing compilers in Section 2.3. In Section 3 we present and analyze parallel I-GEP (and C-GEP). Finally, in Section 4 we present all of our experimental results: in Section 4.1 we present results comparing C-GEP, I-GEP and GEP for Floyd-Warshall, in Section 4.2 results comparing I-GEP to BLAS routines, and in Section 4.3 experimental results on parallel I-GEP using `pthreads`.

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 $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ be an arbitrary function. By GEP (or the *Gaussian Elimination Paradigm*), we refer to the computation in Figure 1, where the algorithm G 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 $i,j,k \in [1,n]$; here $\langle i,j,k\rangle$ $(1 \le i,j,k \le n)$ denotes 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.

As noted in [6] many practical problems can be solved using the GEP construct, including all-pairs shortest paths, LU decomposition, and Gaussian elimination without pivoting. Other problems such as simple dynamic program [5] can be solved using GEP through structural transformation.

We note the following properties of G, which are easily verified by inspection: Given Σ_G, G applies each $\langle i,j,k\rangle \in \Sigma_G$ on c exactly once, and in a specific order. Given any two distinct updates $\langle i_1,j_1,k_1\rangle \in \Sigma_G$ and $\langle i_2,j_2,k_2\rangle \in \Sigma_G$, the update $\langle i_1,j_1,k_1\rangle$ will be applied before $\langle i_2,j_2,k_2\rangle$ if $k_1 < k_2$, or if $k_1 = k_2$ and $i_1 < i_2$, or if $k_1 = k_2$ and $i_1 = i_2$ but $j_1 < j_2$. The running time of G is $\mathcal{O}\left(n^3\right)$ and its I/O complexity is $\mathcal{O}\left(n^3/B\right)$.

Figure 2 presents a recursive function F, also from [6], which we will refer to as I-GEP. As noted in [6], I-GEP is a provably correct implementation of GEP for several important special cases of f and Σ_G, including all cases corresponding to the problems mentioned above. (I-GEP does not solve GEP in its full generality, however.)

I-GEP is cache-oblivious with I/O complexity $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ assuming a tall cache (i.e., $M = \Omega\left(B^2\right)$) [6]. This bound is tight for general GEP [6]. It also computes in-place.

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

1.2 Related Work

As discussed in [6] there exist $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithms for several problems that can be solved using GEP: Floyd-Warshall's APSP [20, 8], Gaussian elimination w/o pivoting [23, 2], matrix multiplication [11] and simple DP [5]. The main attraction of our cache-oblivious approach to 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. I-GEP AND C-GEP

We start by analyzing F from [6] given in Figure 2, which we call I-GEP. The inputs to F are a square submatrix X of $c[1\ldots n, 1\ldots n]$, and indices k_1 and k_2, which satisfy the following constraints:

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

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

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

The base case occurs when $k_1 = k_2$, and F 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 each quadrant in two passes: forward (line 5) and backward (line 6). The initial function call is $\mathrm{F}(c, 1, n)$.

2.1 Properties of I-GEP

We state two theorems that reveal several important properties of F that will be used in Section 2.2 to extend I-GEP to all GEP computations.

Recall that in Section 1.1 we defined Σ_G to be the set of all updates $\langle i,j,k\rangle$ performed by the original GEP algorithm G in Figure 1. Analogously, let Σ_F be the set of all updates performed by $\mathrm{F}(c,1,n)$. Assume each instruction executed by F receives a unique time stamp, and let $\langle i,j,k,t\rangle$ denote an update $\langle i,j,k\rangle$ applied by F 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 in Σ_G exactly once, and no other updates; it also identifies a partial order on the updates F performs. For the proof see technical report [7].

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

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

(b) $\langle i,j,k,t_1\rangle \in \Pi_F \;\wedge\; \langle i,j,k,t_2\rangle \in \Pi_F \;\Rightarrow\; t_1 = t_2$, *i.e.,* F *performs each* $\langle i,j,k\rangle$ *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.,* F *updates each* $c[i,j]$ *in increasing order of* k *values.*

We now introduce some terminology as well as two functions π and δ which will be used in the next theorem. More formal definitions of δ and π are in the technical report [7].

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

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

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

```
G($c$, $n$, $f$, $\Sigma_G$)
(Input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix, $f(\cdot,\cdot,\cdot,\cdot)$ is an arbitrary problem-specific function, and $\Sigma_G$ is a problem-specific set
of triplets such that $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$ is executed in line 4 if $\langle i,j,k\rangle \in \Sigma_G$.)
1. for $k \leftarrow 1$ to $n$ do
2.     for $i \leftarrow 1$ to $n$ do
3.         for $j \leftarrow 1$ to $n$ do
4.             if $\langle i,j,k\rangle \in \Sigma_G$ then $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$
```

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

```
F($X, k_1, k_2$)
($X[1 \ldots 2^q, 1 \ldots 2^q] \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ for some integer $q \geq 0$. Function $f$ and set $\Sigma_G$ are as defined in Figure 1 and assumed
to be available globally. Function F assumes input conditions 2.1. The initial call to F is F($c, 1, n$) for an $n \times n$ input matrix
$c$, where $n$ is assumed to be a power of 2.)

1. if $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ then return          {$T_{X,[k_1,k_2]} = \{\langle i,j,k\rangle | i \in [i_1,i_2] \wedge j \in [j_1,j_2] \wedge k \in [k_1,k_2]\}$}
2. if $k_1 = k_2$ then $c[i_1,j_1] \leftarrow f(c[i_1,j_1], c[i_1,k_1], c[k_1,j_1], c[k_1,k_1])$
3. else          {quadrants of $X$: $X_{11}$ (top-left), $X_{12}$ (top-right), $X_{21}$ (bottom-left) and $X_{22}$ (bottom-right)}
4.     $k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
5.     $F(X_{11}, k_1, k_m)$, $F(X_{12}, k_1, k_m)$, $F(X_{21}, k_1, k_m)$, $F(X_{22}, k_1, k_m)$          {forward pass}
6.     $F(X_{22}, k_m+1, k_2)$, $F(X_{21}, k_m+1, k_2)$, $F(X_{12}, k_m+1, k_2)$, $F(X_{11}, k_m+1, k_2)$          {backward pass}
```

Figure 2: Cache-oblivious I-GEP. For several special cases of f and Σ_G in Figure 1, F **performs the same computation as** G **[6], though there are exceptions (see Section 2.2.1).**

Definition 2.2. *Let $x, y, z \in [1, n]$ be integers.*

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

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

Let $c_k(i, j)$ denote the value of $c[i, j]$ after all updates $\langle i, j, k'\rangle \in \Sigma_G$ with $k' \leq k$, and no other updates have been applied on it by F. The following theorem (proof in tech report [7]) identifies the exact states of $c[i, k]$, $c[k, j]$ and $c[k, k]$ immediately before $\langle i, j, k\rangle$ is applied. One implication of this theorem is that the total order of the updates as applied by F and G can be different (see Section 2.2.1).

Theorem 2.2. *Let δ and π be as defined in Definition 2.2. Then immediately before function* F *performs the update $\langle i, j, k\rangle$ (i.e., before it executes $c[i, j] \leftarrow f(c[i, j], c[i, k], c[k, j], c[k, k])$), the following hold: $c[i, j] = c_{k-1}(i, j)$, $c[i, k] = c_{\pi(j,k)}(i, k)$, $c[k, j] = c_{\pi(i,k)}(k, j)$ and $c[k, k] = c_{\delta(i,j,k)}(k, k)$.*

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

In order to express mathematical expressions with conditionals in compact form, in this section we will use *Iverson's convention* [16] in which $|\mathcal{E}|$ is used 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.

2.2.1 A Closer Look at I-GEP

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

In the following table, 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 Figure 1, while those in the 3rd column follows from Theorem 2.2.

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

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

It follows from Definition 2.2 that for $i, j < k$, $\pi(j, k) \neq k - |j \leq k|$, $\pi(i, k) \neq k - |i \leq k|$ and $\delta(i, j, k) \neq k - |(i < k) \vee (i = k \wedge j \leq k)|$. 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, consider a 2×2 matrix c, and let $\Sigma_G = \{\langle i, j, k\rangle | 1 \leq i, j, k \leq 2\}$. If initially $c[1, 1] = c[1, 2] = c[2, 1] = 0$ and $c[2, 2] = 1$, and f returns the sum of its input values, then F will output $c[2, 1] = 8$, while G will output $c[2, 1] = 2$.

2.2.2 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 2.3. *For $1 \leq i, j, l \leq n$, we define $\tau_{ij}(l)$ to be the largest integer $l' \leq l$ such that $\langle i, j, l'\rangle \in \Sigma_G$ provided such an update exists, and 0 otherwise.*

The significance of τ can be explained as follows. We know from Theorem 2.1 that both F and G apply the updates $\langle i, j, k\rangle$ in increasing order of k values. Hence, at any point of time during the execution of F (or G) if $c[i, j]$ is in state

```
H($X, k_1, k_2$)
($X[1 \ldots 2^q, 1 \ldots 2^q] \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ 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. Function $f$ and set $\Sigma_G$ are as defined in Figure 1 and assumed to be available globally. Similar to F, H assumes input
conditions 2.1. The initial call is H($c, 1, n$) for an $n \times n$ input matrix $c$, where $n$ is assumed to be a power of 2.)

1.  if $T_{X,[k_1,k_2]} \cap \Sigma_G = \emptyset$ then return        $\{T_{X,[k_1,k_2]} = \{\langle i,j,k\rangle | i \in [i_1,i_2] \wedge j \in [j_1,j_2] \wedge k \in [k_1,k_2]\}\}$
2.  if $k_1 = k_2$ then
3.      $i \leftarrow i_1,\ j \leftarrow j_1,\ k \leftarrow k_1$
4.      $c[i,j] \leftarrow f(c[i,j], u_{|j>k|}[i,k], v_{|i>k|}[k,j], u_{|(i>k)\ \vee\ (i=k\ \wedge\ j>k)|}[k,k])$
5.      if $k = \tau_{ij}(j-1)$ then $u_0[i,j] \leftarrow c[i,j]$        $\{\tau_{ij}(l) = \max_{l'} \{l' \mid l' \leq l \wedge \langle i,j,l'\rangle \in \Sigma_G \cup \{\langle i,j,0\rangle\}\}\}$
6.      if $k = \tau_{ij}(j)$ then $u_1[i,j] \leftarrow c[i,j]$
7.      if $k = \tau_{ij}(i-1)$ then $v_0[i,j] \leftarrow c[i,j]$
8.      if $k = \tau_{ij}(i)$ then $v_1[i,j] \leftarrow c[i,j]$
9.  else        {quadrants of $X$: $X_{11}$ (top-left), $X_{12}$ (top-right), $X_{21}$ (bottom-left) and $X_{22}$ (bottom-right)}
10.     $k_m \leftarrow \left\lfloor \frac{k_1+k_2}{2} \right\rfloor$
11.     $H(X_{11}, k_1, k_m)$, $H(X_{12}, k_1, k_m)$, $H(X_{21}, k_1, k_m)$, $H(X_{22}, k_1, k_m)$        {forward pass}
12.     $H(X_{22}, k_m+1, k_2)$, $H(X_{21}, k_m+1, k_2)$, $H(X_{12}, k_m+1, k_2)$, $H(X_{11}, k_m+1, k_2)$        {backward pass}
```

Figure 3: C-GEP: A cache-oblivious implementation of GEP (in Figure 1) that works for all f and Σ_G.

$c_l(i,j)$ ($\hat{c}_l(i,j)$, resp.), where $l \neq 0$, then $\langle i, j, \tau_{ij}(l)\rangle$ is the update that has left $c[i,j]$ in this state.

We extend I-GEP to full generality by modifying F in Figure 2 so that it performs updates according to the second column of Table 1 instead of the third column. For this, it suffices to modify F to save the value of $c[i,j]$ immediately after applying the update $\langle i,j,k\rangle \in \Sigma_G$ for $k \in \{\tau_{ij}(i-1), \tau_{ij}(i), \tau_{ij}(j-1), \tau_{ij}(j)\}$, and then access the appropriate saved values when evaluating f (see tech. report [7] for more details). Since there are exactly n^2 possible (i,j) pairs, we need to save at most $4n^2$ intermediate values.

In Figure 3 we present the modified version of F, which we call H. The only difference between F and H is in the way the updates are performed. In line 2, F updates $c[i,j]$ using entries directly from c, i.e., it updates $c[i,j]$ using whatever values $c[i,j]$, $c[i,k]$, $c[k,j]$ and $c[k,k]$ have at the time of the update. In contrast, H uses four $n \times n$ matrices u_0, u_1, v_0 and v_1 for saving appropriate intermediate values computed for the entries of c as discussed above, which it uses for future updates. For more details on correctness of H see tech. report [7]. Assuming that each of the tests in lines 5–8 can be performed in constant time without additional cache misses, the I/O and time complexity of H are the same as those of F. We will refer to H as C-GEP.

Reducing the Additional Space. The amount of extra space used by H (Figure 3) can be reduced 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 tech. report [7] we use this observation to implement H using only four additional $\frac{n}{2} \times \frac{n}{2}$ matrices and two vectors of length $\frac{n}{2}$ each. Its time and I/O complexity remain unchanged.

2.3 Cache-Oblivious GEP and Compiler Optimization

Tiling is a powerful cache-aware loop transformation technique employed by optimizing compilers for improving temporal locality in nested loops [18, 24]. In contrast, both I-GEP and C-GEP can be viewed as cache-oblivious tiling for the triply nested loops of the form as shown in Figure 1.

Though C-GEP is a legal transformation for any loop in GEP format, I-GEP is not. In tech report [7] we discuss several conditions under which I-GEP is a legal transformation for a given GEP instance. I-GEP performs better than C-GEP in practice (see Section 4) and is in-place. However, the full generality of C-GEP along with the fact that it is a simple variant of I-GEP makes C-GEP a potentially more useful compiler optimization technique than I-GEP.

3. PARALLEL I-GEP AND C-GEP

In this section we consider parallel implementations of I-GEP and C-GEP. It is not difficult to observe that the second and third calls to F in line 5 of the pseudocode for I-GEP given in Figure 2 can be performed 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. 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 $\mathcal{O}\left(n^3/p + n^{\log_2 6}\right)$.

In Figure 6 we present a better parallel implementation of I-GEP. In this figure we have explicitly referred to the different types of functions invoked by I-GEP based on the relative values of the i, j, and k intervals. Here we use the notation introduced in [6] and summarized in Figures 4 and 5 (and in Figures 13 and 14 in the Appendix). 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 (note that only the diagonal entries of W are used). 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, and thus leading

```
F(X, k_1, k_2)          {F ∈ {A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, D_4}}
    ...  ...  ...  ...  ...  ...  ...  ...  ...
  5. $F_{11}(X_{11}, k_1, k_m)$,  $F_{12}(X_{12}, k_1, k_m)$,
     $F_{21}(X_{21}, k_1, k_m)$,  $F_{22}(X_{22}, k_1, k_m)$
  6. $F'_{22}(X_{22}, k_m + 1, k_2)$,  $F'_{21}(X_{21}, k_m + 1, k_2)$,
     $F'_{12}(X_{12}, k_m + 1, k_2)$,  $F'_{11}(X_{11}, k_m + 1, k_2)$
```

Figure 4: I-GEP (partial) from Figure 2, but here F is a template function. Calls in lines 5 & 6 are determined from Figure 5.

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

Figure 5: Functions recursively called by F in Figure 4.

```
A( X, U, V, W )
(Each of $X$, $U$, $V$ and $W$ points to the same $2^q \times 2^q$ square submatrix of $c$ for some integer $q \geq 0$. The initial call to A 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 $T_{XUV} \cap \Sigma_G = \emptyset$ then return        {$T_{XUV}$ = { updates on $X$ using $(i, k) \in U$ and $(k, j) \in V$ },
                                                        and $\Sigma_G$ is the set of updates performed by iterative GEP}
  2. if $X$ is a $1 \times 1$ matrix then $X \leftarrow f(X, U, V, W)$
     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.}
  3.     A( $X_{11}$, $U_{11}$, $V_{11}$, $W_{11}$ )
  4.     parallel : B$_1$( $X_{12}$, $U_{11}$, $V_{12}$, $W_{11}$ ),  C$_1$( $X_{21}$, $U_{21}$, $V_{11}$, $W_{11}$ )
  5.     D$_1$( $X_{22}$, $U_{21}$, $V_{12}$, $W_{11}$ )
  6.     A( $X_{22}$, $U_{22}$, $V_{22}$, $W_{22}$ )
  7.     parallel : B$_2$( $X_{21}$, $U_{22}$, $V_{21}$, $W_{22}$ ),  C$_2$( $X_{12}$, $U_{12}$, $V_{22}$, $W_{22}$ )
  8.     D$_4$( $X_{11}$, $U_{12}$, $V_{21}$, $W_{22}$ )
```

```
B$_l$( X, U, V, W )                                   { $l \in \{1, 2\}$ }
($X \equiv V \equiv c[i_1..i_2, j_1..j_2]$ and $U \equiv W \equiv c[i_1..i_2, k_1..k_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[i_1, i_2] = [k_1, k_2]$ and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.)

  1. if $T_{XUV} \cap \Sigma_G = \emptyset$ then return
  2. if $X$ is a $1 \times 1$ matrix then $X \leftarrow f(X, U, V, W)$
     else
  3.     parallel : B$_l$( $X_{11}$, $U_{11}$, $V_{11}$, $W_{11}$ )
                    B$_l$( $X_{12}$, $U_{11}$, $V_{12}$, $W_{11}$ )
  4.     parallel : D$_l$( $X_{21}$, $U_{21}$, $V_{11}$, $W_{11}$ )
                    D$_l$( $X_{22}$, $U_{21}$, $V_{12}$, $W_{11}$ )
  5.     parallel : B$_l$( $X_{21}$, $U_{22}$, $V_{21}$, $W_{22}$ )
                    B$_l$( $X_{22}$, $U_{22}$, $V_{22}$, $W_{22}$ )
  6.     parallel : D$_{l+2}$( $X_{11}$, $U_{12}$, $V_{21}$, $W_{22}$ )
                    D$_{l+2}$( $X_{12}$, $U_{12}$, $V_{22}$, $W_{22}$ )
```

```
C$_l$( X, U, V, W )                                   { $l \in \{1, 2\}$ }
($X \equiv U \equiv c[i_1..i_2, j_1..j_2]$ and $V \equiv W \equiv c[k_1..k_2, j_1..j_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[j_1, j_2] = [k_1, k_2]$ and $[i_1, i_2] \cap [k_1, k_2] = \emptyset$.)

  1. if $T_{XUV} \cap \Sigma_G = \emptyset$ then return
  2. if $X$ is a $1 \times 1$ matrix then $X \leftarrow f(X, U, V, W)$
     else
  3.     parallel : C$_l$( $X_{11}$, $U_{11}$, $V_{11}$, $W_{11}$ )
                    C$_l$( $X_{21}$, $U_{21}$, $V_{11}$, $W_{11}$ )
  4.     parallel : D$_{2l-1}$( $X_{12}$, $U_{11}$, $V_{12}$, $W_{11}$ )
                    D$_{2l-1}$( $X_{22}$, $U_{21}$, $V_{12}$, $W_{11}$ )
  5.     parallel : C$_l$( $X_{12}$, $U_{12}$, $V_{22}$, $W_{22}$ )
                    C$_l$( $X_{22}$, $U_{22}$, $V_{22}$, $W_{22}$ )
  6.     parallel : D$_{2l}$( $X_{11}$, $U_{12}$, $V_{21}$, $W_{22}$ )
                    D$_{2l}$( $X_{21}$, $U_{22}$, $V_{21}$, $W_{22}$ )
```

```
D$_l$( X, U, V, W )                                   { $l \in \{1, 2, 3, 4\}$ }
($X \equiv c[i_1..i_2, j_1..j_2]$, $U \equiv c[i_1..i_2, k_1..k_2]$, $V \equiv c[k_1..k_2, j_1..j_2]$ and $W \equiv c[k_1..k_2, k_1..k_2]$, where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$, $[i_1, i_2] \cap [k_1, k_2] = \emptyset$, and $[j_1, j_2] \cap [k_1, k_2] = \emptyset$.)

  1. if $T_{XUV} \cap \Sigma_G = \emptyset$ then return
  2. if $X$ is a $1 \times 1$ matrix then $X \leftarrow f(X, U, V, W)$
     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.}
  3.     parallel : D$_l$( $X_{11}$, $U_{11}$, $V_{11}$, $W_{11}$ ),  D$_l$( $X_{12}$, $U_{11}$, $V_{12}$, $W_{11}$ ),
                    D$_l$( $X_{21}$, $U_{21}$, $V_{11}$, $W_{11}$ ),  D$_l$( $X_{22}$, $U_{21}$, $V_{12}$, $W_{11}$ )
  4.     parallel : D$_l$( $X_{11}$, $U_{12}$, $V_{21}$, $W_{22}$ ),  D$_l$( $X_{12}$, $U_{12}$, $V_{22}$, $W_{22}$ ),
                    D$_l$( $X_{21}$, $U_{21}$, $V_{21}$, $W_{22}$ ),  D$_l$( $X_{22}$, $U_{22}$, $V_{22}$, $W_{22}$ )
```

Figure 6: Multithreaded I-GEP. Initial call is A(c, c, c, c) on an $n \times n$ matrix c, where n is a power of 2.

to better parallelism. The initial call is to an input of type A, where the intervals for i, j and k are identical.

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:

$$T_A(n) \leq 2(T_A(n/2) + \max\{T_B(n/2), T_C(n/2)\} + T_D(n/2)) + 8$$
$$T_B(n) \leq 2(T_B(n/2) + T_D(n/2)) + 8$$
$$T_C(n) \leq 2(T_C(n/2) + T_D(n/2)) + 8$$
$$T_D(n) \leq 2T_D(n/2) + 8$$

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

Theorem 3.1. *When executed with p processors, multithreaded I-GEP performs $T_1 = \mathcal{O}(n^3)$ work and terminates in $\frac{T_1}{p} + T_\infty = \mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ 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. For instance as noted in [6], only a subset of the calls are made for Gaussian elimination without pivoting. However, the parallel time bound remains the same as in Theorem 3.1 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 5 and 6 of Figure 4 in parallel and hence the parallel time bound is reduced to $\mathcal{O}\left(\frac{n^3}{p} + n\right)$. Note that this matrix multiplication computation does not assume associativity of addition.

We have implemented this multithreaded version of I-GEP for Floyd-Warshalls's APSP, square matrix multiplication and Gaussian elimination w/o pivoting in `pthreads`, and we report some experimental results in Section 4.3.

3.1 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. Part (b) of 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. This schedule gives a better result than the one given in part (a) for the work-stealing scheduler Cilk [12]; the bound in part (a) is obtained by applying a result in [13] on the caching performance of parallel algorithms whose sequential cache complexity is a concave function of work.

Lemma 3.1. *Consider multithreaded I-GEP executed with p processors, each with a private cache of size M and block size B.*

(a) *When executed by Cilk, with high probability I-GEP incurs* $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{(p\cdot n\log^2 n)^{\frac{1}{3}} n^2}{B} + p \cdot n\log^2 n\right)$ *cache misses.*

(b) *There exists a deterministic schedule which incurs only* $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \sqrt{p}\cdot\frac{n^2}{B}\right)$ *cache misses.*

Shared Caches. Here we consider the case when the p processors share a single cache of size M_p. Part (a) of lemma 3.2 below is obtained using a general result for shared caches given in [1] for a PDF *(parallel depth first search)* schedule. Better bounds are obtained in part (b) of lemma 3.2 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 [1]. 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 3.2. *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*

(a) *With a* PDF*-schedule,*
$Q_p \leq Q_1$ *if* $M_p \geq M_1 + \frac{9}{2}n(\log^2 n + 2)(p-1)$.

(b) *With the hybrid depth-first schedule,*
(i) $Q_p \leq Q_1$ *if* $M_p \geq M_1 + 16p^{\frac{3}{2}}$,
(ii) *If* $M_1 = M_p$ *then* $Q_p = O(Q_1)$ *provided* $p = O(M_p^{\frac{2}{3}})$.

(c) *For both schedules,* $T_p = \mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$.

4. EXPERIMENTAL RESULTS

We ran our experiments on the three architectures listed in Table 2. Each machine can perform at most two double precision floating point operations per clock cycle. 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) [17]. 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 [21] 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; all plots appear at the end of this section.

4.1 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 [10, 22] 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 [9] for external memory accesses. The STXXL library maintains its own fully associative cache in RAM with pages from the

Model	Processors	Speed	Peak GFLOPS (per processor)	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	6.12	8 KB (4-way, $B = 64$ B)	512 KB (8-way, $B = 64$ B)	4 GB
AMD Opteron 250	2	2.4 GHz	4.8	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	4 GB
AMD Opteron 850	8 (4 dual-core)	2.2 GHz	4.4	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	32 GB

Table 2: Machines used for experiments.

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 Figure 7(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. 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 Figure 7(b) we keep n and M fixed, and vary M/B (by varying B), and observe that in general, I/O wait times increase linearly with the increase of M/B. However, when M/B is small, the number of page faults increases which affects the I/O performance of all algorithms.

In-Core Computation. In Figure 8 we plot the performance of GEP and I-GEP on both Intel Xeon and AMD Opteron 250. We optimized I-GEP as described in Section 4.2. On Intel Xeon I-GEP runs around 5 times faster than GEP while on AMD Opteron it runs around 4 times faster.

In Figure 9 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 reinitializations of the temporary matrices (i.e., u_0, u_1, v_0 and v_1) compared to the $4n^2$-space C-GEP.

4.2 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 $o\left(n^3\right)$ (i.e., by moving the division operations out of the innermost loops).

– We use the *bit-interleaved* layout (e.g., see [11, 4]) 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 Figure 11 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 [25]. 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 remaining time is spent in other overheads including recursion, loops, cache misses, etc. From the plots in Figure 11 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 [15] 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 Figure 10 we plot the performance of Gaussian elimination without pivoting using GEP, I-GEP and GotoBLAS [15] 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 [14] 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 [26]. The experimental results in [26] 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 [26] 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 [26] 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 [26] 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.

4.3 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 an 8-processor AMD Opteron 850 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 Figure 12 we plot the speed-up factors achieved by multithreaded I-GEP over its unthreaded version as the number of concurrent threads is increased.

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 Section 3, 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.

4.4 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 Section 4.1). 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 [19] 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 [19] we arrange the entries of each base-case submatrix in a prefetecher-freindly layout, i.e., in row-major order (see Section 4.2). Note that we include the cost of converting to and from this layout in the time bounds we report.

– 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 Section 4.2 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.

5. REFERENCES

[1] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. *Proc. SPAA*, pp. 235–244, 2004.

[2] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. *Proc. SPAA*, pp. 297–308, 1996.

[3] E. Chan, E. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. *Proc. SPAA*, 2007.

[4] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thotethodi. Recursive array layouts and fast parallel matrix multiplication. *Proc. SPAA*, pp. 222–231, 1999.

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

[6] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. *Proc. SODA*, pp. 591–600, 2006.

[7] R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework and experimental evaluation. CS TR-06-04, UT Austin, 2006.

[8] P. D'Alberto and A. Nicolau. 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.

[9] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. *Proc. ESA*, pp. 640–651, 2005.

[10] R. Floyd. Algorithm 97 (SHORTEST PATH). *CACM*, 5(6):345, 1962.

[11] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *Proc. FOCS*, pp. 285–297, 1999.

[12] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. *Proc. PLDI*, pp. 212–223, 1998.

[13] M. Frigo and V. Strumpen. The cache-complexity of multithreaded cache-oblivious algorithms. *Proc. SPAA*, pp. 271–280, 2006.

[14] J. Gunnels, F. Gustavson, G. Henry, and R. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM TOMS*, 27(4):422–455, 2001.

[15] K. Goto. GotoBLAS, 2005. http://www.tacc.utexas.edu/resources/software.

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

[17] MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP disk drives product/maintenance manual. http://www.fujitsu.com/downloads/COMP/fcpa/hdd/.

[18] S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[19] S. Pan, C. Cherng, K. Dick, and R. Ladner. Algorithms to take advantage of hardware prefetching. *Proc. ALENEX*, pp. 91–98, 2007.

[20] J. Park, M. Penner and V. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE TPDS*, 15(9):769–782, 2004.

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

[22] S. Warshall. A theorem on boolean matrices. *JACM*, 9(1):11–12, 1962.

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

[24] M. Wolf and M. Lam. A data locality optimizing algorithm. *Proc. PLDI*, pp. 30–44, 1991.

[25] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[26] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-aware programs. *Proc. SPAA*, 2007.

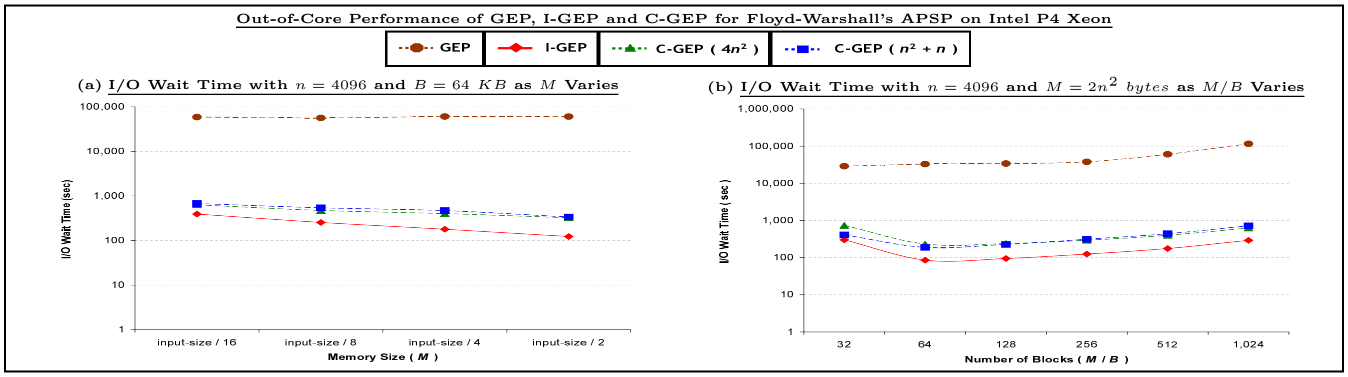

Figure 7: Comparison of out-of-core performance of GEP, I-GEP and C-GEP on Intel Pentium 4 Xeon with a fast hard disk (10K RPM, 4.5 ms avg. seek time, 64 to 107 MB/s transfer rate).

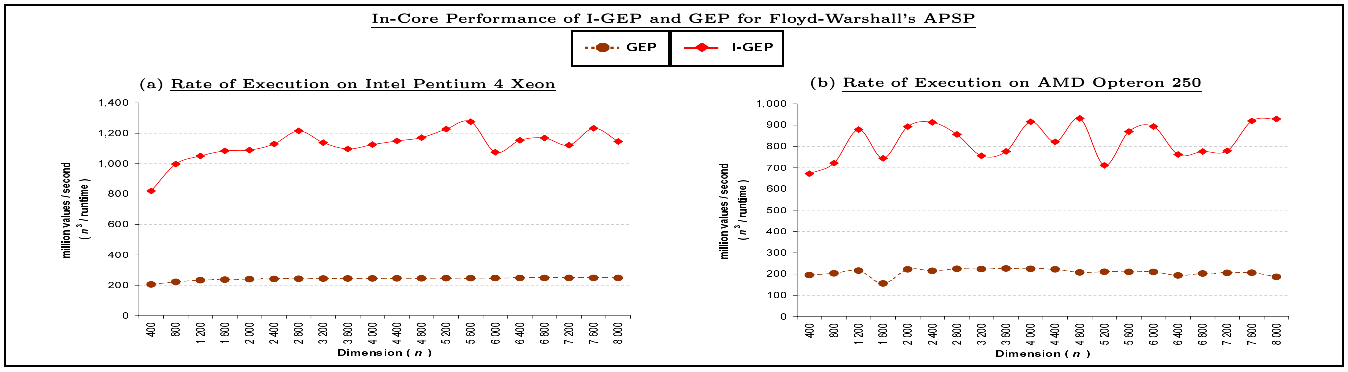

Figure 8: Comparison of I-GEP and GEP on Intel Xeon and AMD Opteron for computing Floyd-Warshall's all-pairs shortest paths. Both machines have two processors, but only one was used.

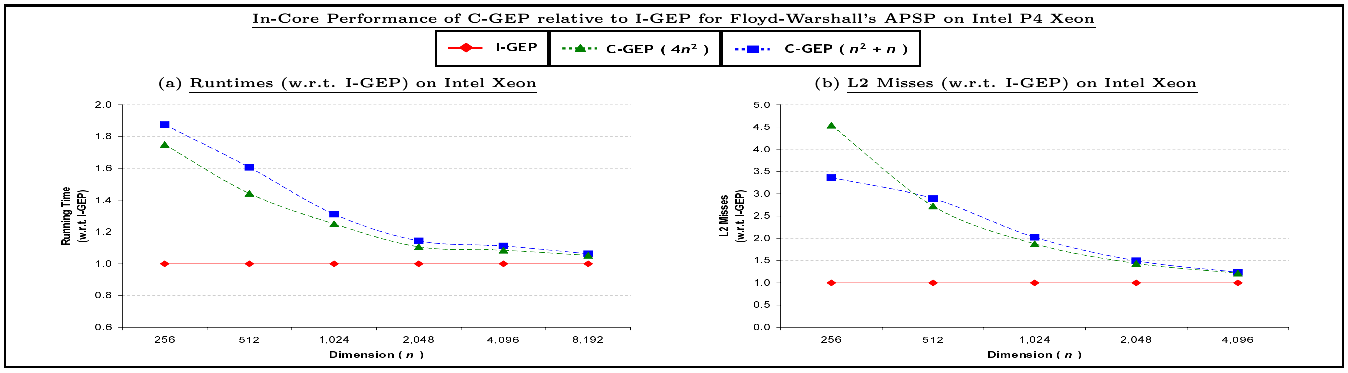

Figure 9: Comparison of in-core performance of I-GEP and C-GEP on Intel Pentium 4 Xeon.

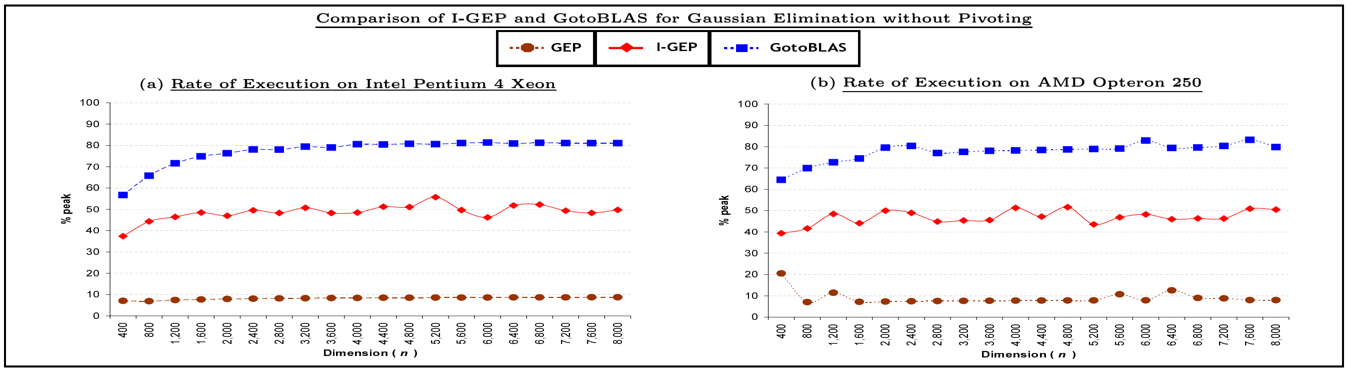

Figure 10: Comparison of I-GEP and GotoBLAS on Intel Xeon and AMD Opteron for performing Gaussian elimination without pivoting. Both machines have two processors, but only one was used.

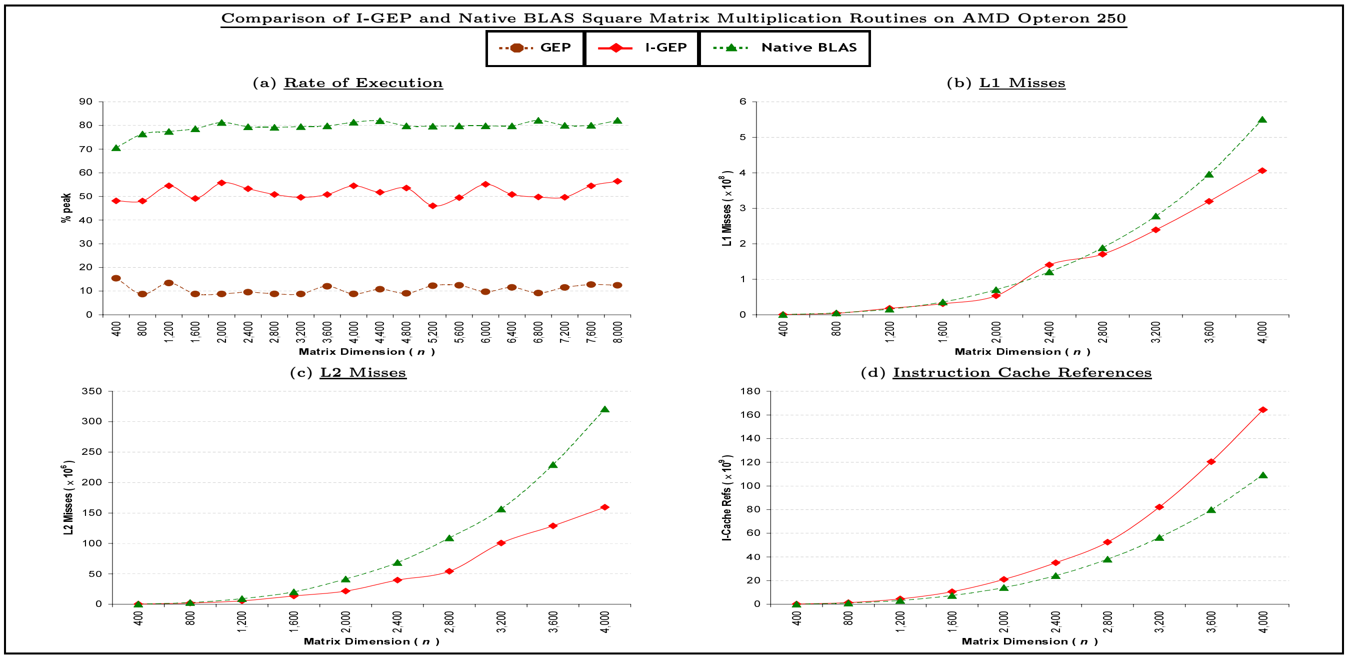

Figure 11: Comparison of I-GEP and native BLAS square matrix multiplication routines on a 2.4 GHz dual processor AMD Opteron 250 (only one processor was used).

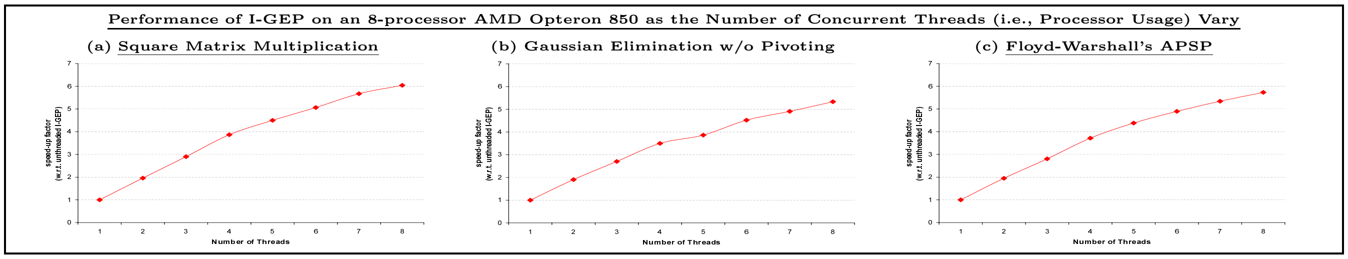

Figure 12: Performance of I-GEP on an 8-processor AMD Opteron 850 for square matrix multiplication, Gaussian elimination w/o pivoting and Floyd-Warshall's all-pairs shortest paths on 5000×5000 matrices as number of concurrent threads is varied.

APPENDIX

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

Figure 13: Function specific pre-condition $P(F)$ for F in Figure 4 assuming $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ (from [6]).

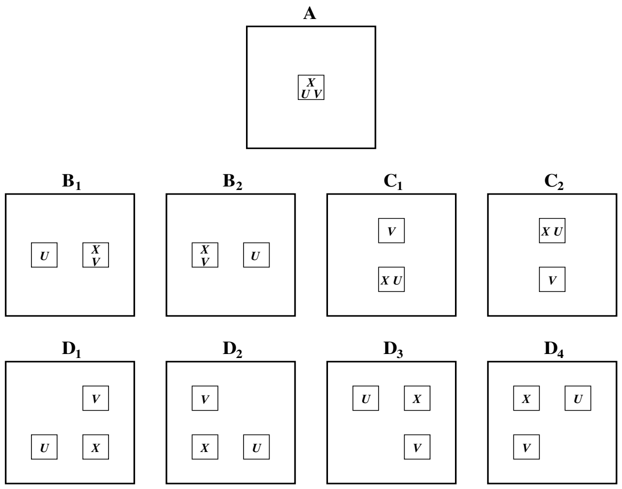

Figure 14: Relative positions of $U \equiv c[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv c[k_1 \ldots k_2, j_1 \ldots j_2]$ w.r.t. $X \equiv c[i_1 \ldots i_2, j_1 \ldots j_2]$ assumed by different instantiations of F.