HPC 2012 Final Project

Implementing the QR Factorization in Parallel

Authors:

Jacqueline Bush

Paul Torres

December 21, 2012

Contents

1 Introduction **1**

2 Background **1**

2.1 Basics on QR Decomposition 1

2.2 Why QR? 2

2.2.1 Least Squares and QR Factorization 3

2.2.2 The QR Algorithm 4

2.3 QR Factorization in Serial 5

2.3.1 Modified Gram Schmidt 5

2.3.2 Givens Rotations 7

2.3.3 Householder Reflections 8

3 WY Implementation **9**

3.1 Performance: Naive Householder QR Implementation vs WY Householder QR Implementation 11

4 Tiled QR Factorization: Version 1 **12**

4.1 Implementation 15

5 Tiled QR Factorization: Version 2 **16**

6 Blocked QR Performance: **20**

7 Conculsion **25**

8 Andreas's Questions = ANSWERED **27**

List of Algorithms

2.1 The QR Algorithm (without shifts) 4

2.2 Modified Gram-Schmidt 6

2.3 Givens QR Factorization 8

2.4 Householder QR Factorization 9

3.1 Computes matrices W and Y such that $Q = I + WY^T$ in Householder QR Factorization 11

3.2 $v_k = \text{House}(x)$ Computes Householder Vector. 11

4.1 Tiled QR Factorization: Version 1 13

5.1 Tiled QR Factorization: Version 2 19

1 Introduction

We explore the use of OpenMP to create an optimized QR matrix factorization algorithm. A combination of tiling of the input matrices and the computation of Householder reflectors utilizing what is known as the ”WY” representation yields improved performance over a naive Householder algorithm. The attractiveness of the WY algorithm stems from its lower memory cost and slightly better performance even if the complexity is similar.

The WY algorithm works best upon small matrices of a fixed size. Because of the small size of the submatrices, and despite the fact that certain steps require matrix multiplication and transposition of them, further tiling of submatrices for multiplication–in the manner of class assigment #6–caused performance to suffer. This in retrospect is unsurprising, yet it was initially unanticipated. Finally we explored two approaches yielding successively better performance in the manner in which the submatrices of the input matrices are accessed and updated.

2 Background

2.1 Basics on QR Decomposition

In linear algebra, a QR decomposition of a matrix is a decomposition of a matrix $A \in \mathbb{C}^{m\times n}$ into a product

$$A = QR \tag{2.1}$$

where $Q \in \mathbb{C}^{m\times m}$ is a unitary orthogonal matrix and $R \in \mathbb{C}^{m\times n}$ is an upper triangular matrix. Since Q is unitary

$$\det(Q) = \pm 1 \quad \text{and} \quad Q^*Q = I.$$

If $m \geq n$ then R has the following form,

$$R = \begin{pmatrix} * & * & * & \cdots & * & * & * \\ 0 & * & * & \cdots & * & * & * \\ 0 & 0 & * & \cdots & * & * & * \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & * & * & * \\ 0 & 0 & 0 & \cdots & 0 & * & * \\ 0 & 0 & 0 & \cdots & 0 & 0 & * \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{pmatrix} = \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}.$$

The factorization

$$A = \hat{Q}\hat{R} \tag{2.2}$$

$\hat{Q} \in \mathbb{C}^{m\times n}$ and $\hat{R} \in \mathbb{C}^{n\times n}$ is called the reduced QR factorization.

2.2 Why QR?

QR decompositions can be used for many things; they can be used to find other matrix factorizations, such as SVD, to find the eigenvalues and eigenvectors of the matrix A, they can be used to solve the least squares problem and they are a fundamental part of the QR algorithm.

2.2.1 Least Squares and QR Factorization

One example of how QR factorizations are used is the least squares problem,

$$\min_{x\in\mathbb{R}^n} ||Ax-y||_2^2. \tag{2.3}$$

Suppose we know the QR factorization of $A \in \mathbb{C}^{m\times n}$, i.e. $A = QR$. Since the least squares problem doesn't make sense if A is under determined, without loss of generality we can assume that $m \geq n$. Then (2.3) reduces to

$$\begin{aligned}
\min_{x\in\mathbb{R}^n} ||Ax-y||_2^2 &= \min_{x\in\mathbb{R}^n} ||QRx-y||_2^2 \\
&= \min_{x\in\mathbb{R}^n} \left|\left| Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x - y \right|\right|_2^2 \\
&= \min_{x\in\mathbb{R}^n} \left|\left| Q \left(\begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x - Q^*y \right) \right|\right|_2^2 \\
&= \min_{x\in\mathbb{R}^n} \left|\left| \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x - Q^*y \right|\right|_2^2 \qquad (2.4)
\end{aligned}$$

Since Q is unitary $||Qx||_2^2 = ||x||_2^2$. Now suppose $Q^*y = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$ where $c_1 = \hat{Q}^*y \in \mathbb{C}^n$, $c_2 \in \mathbb{C}^{m-n}$ and plug into (2.4) to get

$$\begin{aligned} \min_{x\in\mathbb{R}^n} ||Ax-y||_2^2 &= \min_{x\in\mathbb{R}^n} \left|\left| \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x - \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \right|\right|_2^2 \\ &= \min_{x\in\mathbb{R}^n} \left(||\hat{R}x - c_1||_2^2 + ||c_2||_2^2 \right) \end{aligned} \tag{2.5}$$

Clearly the minimum of (2.5) occurs when $\hat{R}x = c_1$. So,

$$\min_{x\in\mathbb{R}^n} ||Ax-y||_2^2 = ||c_2||_2^2. \tag{2.6}$$

Hence by finding a QR decomposition of A we have reduced the least squares problem to solving the upper triangular linear system

$$\hat{R}x = \hat{Q}^*y. \tag{2.7}$$

2.2.2 The QR Algorithm

The QR algorithm is the most widely used algorithm for finding eigenvalues of small and dense matrices. It is also a stable way of finding QR factorizations of matrix powers A^2, A^3, $\cdots$. Consider the most basic version of the QR Algorithm,

Algorithm 2.1 The QR Algorithm (without shifts)

```
$A^{(0)} = A$
**for** $k = 1, 2, \cdots$ **do**
    $Q^{(k)}R^{(k)} = A^{(k-1)}$
    $A^{(k)} = R^{(k)}Q^{(k)}$
**end for**
```

Under suitable assumptions, this simple algorithm converges to a shur normal form

T for the matrix A, where $A = UTU^T$, U unitary, T is upper triangular if A is arbitrary, diagonal if A is hermitian. Since the Shur form T is similar to the original matrix A it has the same eigenvalues. These eigenvalues are the elements in the diagonal of T.

Notice that the QR algorithm without shifts has two basic procedures. It first factors A into its QR factorization, a $O(n^3)$ operation count. It then multiplies R and Q, also an $O(n^3)$ operation count if carried out naively. For this reason the QR algorithm is not generally used on large matrices. By performing QR factorization in parallel we can speed up this algorithm allowing it to become feasible for larger matrices.

2.3 QR Factorization in Serial

To compute a QR decomposition of a matrix A sequentially the standard approach is to either use the Modified Gram-Schmidt algorithm or to apply Orthogonal Transformations such as Givens Rotations or Householder Reflections.

2.3.1 Modified Gram Schmidt

Let a_i denote the ith column of A, let q_i denote the ith column of Q, and as usual let r_{ij} denote the element in the (i, j) position in R. Finally let v_i denote the ith column of Q before it is normalized. Suppose q is an orthogonal normalized vector. Then

$$P_q = qq^* \tag{2.8}$$

is an rank one orthogonal projector that isolates the component in the q direction. The complement of P_q,

$$P_{\perp q} = I - qq^* \tag{2.9}$$

is a rank $m-1$ orthogonal projector that eliminates the component in the direction of q. Let,

$$P_j = P_{\perp q_{j-1}} \cdots P_{\perp q_2} P_{\perp q_1} \tag{2.10}$$

with $P_1 = I$. The modified Gram Schmidt algorithm is based on the following equation,

$$v_j = P_{\perp q_{j-1}} \cdots P_{\perp q_2} P_{\perp q_1} a_j. \tag{2.11}$$

If (2.11) is applied sequentially we get,

$$\begin{aligned}
v_j^{(1)} &= a_j \\
v_j^{(2)} &= P_{\perp q_1} v_j^{(1)} = v_j^{(1)} - q_1 q_1^* v_j^{(1)} \\
v_j^{(3)} &= P_{\perp q_2} v_j^{(2)} = v_j^{(2)} - q_2 q_2^* v_j^{(2)} \\
&\vdots \qquad \vdots \qquad\qquad \vdots \\
v_j = v_j^{(j)} &= P_{\perp q_{j-1}} v_j^{(j-1)} = v_j^{(j-1)} - q_{j-1} q_{j-1}^* v_j^{(j-1)}
\end{aligned}$$

This process is illustrated in the modified Gram Schmidt pseudo code given below.

Algorithm 2.2 Modified Gram-Schmidt

```
for $i = 1$ to $n$ do
  $v_i = a_i$
end for
for $i = 1$ to $n$ do
  $r_{ii} = ||v_i||$
  $q_i = v_i / r_{ii}$
  for $j = i + 1$ to $n$ do
    $r_{ij} = q_i^* v_i$
    $v_j = v_j - r_{ij} q_i$
  end for
end for
```

If $A \in \mathbb{C}^{m \times n}$ then this algorithm has an $O(mn^2)$ operation count. The main disadvantage of the modified Gram Schmidt algorithm is that it is susceptible to precision problems in which the columns of Q are not orthonormal.

2.3.2 Givens Rotations

Givens Rotations act on only two rows at a time. So they can be described by a matrix of the form

$$G = \begin{pmatrix} s_1 & s_2 \\ -s_2 & s_1 \end{pmatrix}$$

where $s_1^2 + s_2^2 = 1$. Notice that G is orthogonal, since $GG^* = I$ and $\det G = 1$. Also notice that for any vector $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{C}^2$,

$$Gx = \begin{pmatrix} ||x|| \\ 0 \end{pmatrix} \quad \text{if} \quad s_1 = \frac{x_1}{||x||}, \text{ and } s_2 = \frac{x_2}{||x||}$$

A succession of Givens rotations on varying pairs of rows is applied to A to reduce A to upper triangular form R. Observe that only one element is zeroed out on each application of a givens rotation.

Algorithm 2.3 Givens QR Factorization

```
$Q = I$; $R = A$
for  $i = 1$ to $n$ do
  for  $k = i + 1$ to $m$ do
    $s_1 = r_{ii}$ and $s_2 = r_{ki}$
    if  $s_2 \neq 0$ then
      $s = \sqrt{s_1^2 + s_2^2} = ||(s_1, s_2)||_2$
      $s_1 = s^{-1} s_1$ and $s_2 = s^{-1} s_2$
      $\begin{pmatrix} e_i^* R \\ e_k^* R \end{pmatrix} = \begin{pmatrix} s_1 & s_2 \\ -s_2 & s_1 \end{pmatrix} \begin{pmatrix} e_i^* R \\ e_k^* R \end{pmatrix}$      Givens rotation on rows i, k
      $\begin{pmatrix} e_i^* Q \\ e_k^* Q \end{pmatrix} = \begin{pmatrix} s_1 & s_2 \\ -s_2 & s_1 \end{pmatrix} \begin{pmatrix} e_i^* Q \\ e_k^* Q \end{pmatrix}$      Givens rotation on rows i, k
    end if
  end for
end for
```

This algorithm runs in $O(mn^2)$ flops. In general Givens QR factorization is easier to program then Householder QR factorization but, unless A is sparse, tends to perform slower.

2.3.3 Householder Reflections

Householder Reflections are special unitary matrices Q_i such that $Q_n \cdots Q_2 Q_1 A = Q^* A$ is upper triangular. Q_i has the form,

$$Q_i = \begin{pmatrix} I & 0 \\ 0 & P_i \end{pmatrix} \tag{2.12}$$

where

$$P_i = I - 2\frac{v_i v_i^*}{v_i^* v_i} \tag{2.13}$$

and

$$v_i = \text{sign}(a_{ii})||A_{i:m,i}||_2 e_1 + A_{i:m,i}. \tag{2.14}$$

This allows each Q_i to zero out all on the elements in the ith column of A below the diagonal.

Algorithm 2.4 Householder QR Factorization

```
for $k = 1$ to $n$ do
    $x = A_{k:m,k}$
    $v_k = \text{sign}(x_1)||x||_2 e_1 + x$
    $v_k = v_k/||v_k||_2$
    $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})$
end for
```

This algorithm runs in $O(2n^2m - \frac{2n^3}{3})$ flops. In general this algorithm runs faster then Givens algorithm, it is also more stable then Gram-Schmidt. We use a version of this algorithm later in our code.

3 WY Implementation

Notice that if we want to obtain a QR factorization via householder reflections as given earlier (algorithm 2.3.3) we have an unfavorable ratio of vector loads and unloads compared to actually computation. Also notice that we would need to do additional n matrix multiplications to recover Q bringing to the operation count of the naive householder QR algorithm to $O(2n^2m - n^3/3)$. To solve this problem we represent products of Householder matrices,

$$Q_k = P_1 \cdots P_k$$

in the form

$$Q_k = I + W_k Y_k^T$$

where W_k and Y_k are n by k matrices and

$$P_i = I - \beta v_i v_i^T, \qquad \beta = -\frac{2}{v_i^T v_i}.$$

is a rank one update. Then

$$Q_k^T A = (I + Y_k W_k^T)A = A + Y_k W_k^T A$$

and

$$\begin{aligned}
Q_k = Q_{k-1}P_k &= (I + W_{k-1}Y_{k-1}^T)(I - \beta v_k v_k^T) \\
&= I + W_{k-1}Y_{k-1}^T - \beta Q_{k-1} v_k v_k^T \\
&= I + \begin{pmatrix} W_{k-1} & -\beta Q_{k-1} v_k \end{pmatrix} \begin{pmatrix} Y_{k-1}^T \\ v_k^T \end{pmatrix} \\
&= I + \begin{pmatrix} W_{k-1} & -\beta Q_{k-1} v_k \end{pmatrix} \begin{pmatrix} Y_{k-1} & v_k \end{pmatrix}^T \\
\Rightarrow \ W_k &= \begin{pmatrix} W_{k-1} & -\beta Q_{k-1} v_k \end{pmatrix} \\
\text{and } Y_k &= \begin{pmatrix} Y_{k-1} & v_k \end{pmatrix}.
\end{aligned}$$

From this we generate the following algorithm.

Algorithm 3.1 Computes matrices W and Y such that $Q = I + WY^T$ in Householder QR Factorization

```
Initialize $W \in \mathbb{R}^{m \times n}$ and $Y^T \in \mathbb{R}^{n \times m}$ as zero matrices.
for $k = 1$ to $m$ do
    $a_k = (I + WY^T)^T a_k$,    i.e $a_k = Q_{k-1}^T a_k$    skip if $k = 1$.
    $v_k = \text{House}(a_k)$
    $z = -2(I + WY^T)v_k$
    $W = \begin{pmatrix} W & z \end{pmatrix}$ i.e replace kth zero column with z.
    $Y^T = \begin{pmatrix} Y \\ v_k \end{pmatrix}$ i.e. replace kth zero row with $v_k$.
end for
```

where

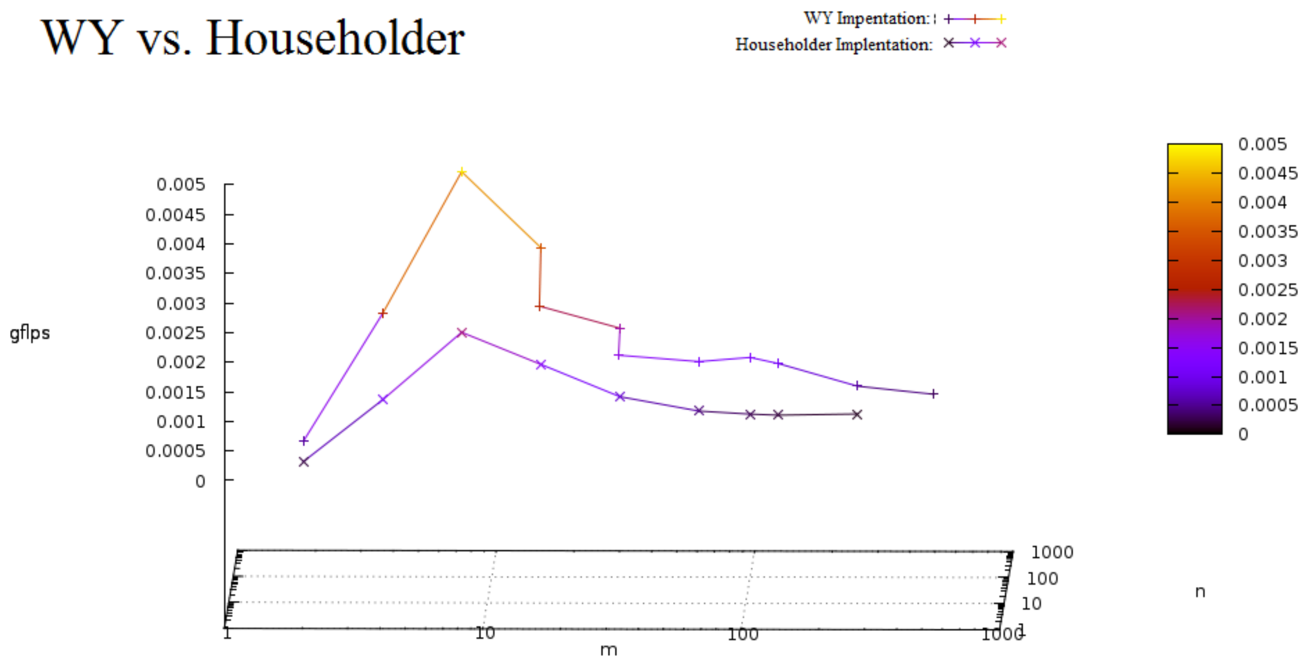
Algorithm 3.2 $v_k = \text{House}(x)$ Computes Householder Vector.

```
$v_k(i) = 0$ if $k < i$, and $v_k(i) = x(i)$ if $k \geq i$.
$v_k = \text{sign}(x_k)||x||_2 e_k + x$
$v_k = v_k / ||v_k||_2$
```

This code runs in O$(2n^2m - n^3/3)$ flops, approximately the same number of flops as the previous algorithm however the ratio of loading and unloading blocks to actual computation is better.

3.1 Performance: Naive Householder QR Implementation vs WY Householder QR Implementation

To check that the WY householder QR implementation is an improvement over the regular householder implementation we ran both codes over the same square matrices of size n on a node of Bowery and plotted the results in terms of GFlops per second.



Observe that for any size the WY implementation is faster than the navie householder implentation. For this reason we use the WY implentation to calculate the QR factorization on blocks. Also notice that the WY code is optimized with block size 8. This implies that our Tiled QR algorithms should be run with block size 8 not 16 (as we initially thought). We investigated this further later in the paper.

4 Tiled QR Factorization: Version 1

Recall that the goal of this paper is to implement QR factorization in parallel. In class we saw that one way to preform matrix matrix multiplication in parallel is to tile the matrices A and B. Using the tiled matrix matrix multiply algorithm as inspiration we created a tiled QR factorization algorithm. Our idea is demonstrated in the pseudo code below.

Algorithm 4.1 Tiled QR Factorization: Version 1

```
$Q = I$
for $k = 1$ to $\min(hn = \text{number of blocks in column}, wn = \text{number of blocks in row})$ do
  for $i = k + 1$ to $hn$ do
    Preform $QR$ factorization on $\begin{bmatrix} A_{kk} \\ A_{ik} \end{bmatrix}$ to get $\begin{bmatrix} R_{kk} \\ 0 \end{bmatrix}$ and $Q_{ik}$
    Update $Q$ matrix
    OMP PARALLEL FOR LOOP:
    for $j = k + 1$ to $wn$ do
      Factor out $Q_{ik}$ from $\begin{bmatrix} A_{kj} \\ A_{ij} \end{bmatrix}$ to get $\begin{bmatrix} B_{kj} \\ B_{ij} \end{bmatrix}$.
    end for
  end for
  if $k = hn - 1$ and $hn = \min(hn, wn)$ then
    Preform $QR$ factorization on $A_{kk}$, resulting in $Q_{kk}$ and $R_{kk}$
    Update $Q$
    OMP PARALLEL FOR LOOP:
    for $j = k + 1$ to $wn$ do
      Factor out $Q_{kk}$ from $A_{kj}$ to get $B_{kj}$.
    end for
  end if
end for
```

Where for block size b, $A_{ij} \in M_{b\times b}(\mathbb{C})$, $Q_{ii} \in M_{b\times b}(\mathbb{C})$, $Q_{ij} \in M_{2b\times 2b}(\mathbb{C})$ and Q_{ij} is a unitary matrix. To understand why this algorithm works mathematically we consider a 3 by 4 blocked matrix A.

$$\begin{aligned}
A &= \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \end{pmatrix} \\
&= \begin{pmatrix} (Q_{12})_{11} & (Q_{12})_{12} & 0 \\ (Q_{12})_{21} & (Q_{12})_{22} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ 0 & B_{22} & B_{23} & B_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \end{pmatrix} \\
&= \begin{pmatrix} (Q_{12})_{11} & (Q_{12})_{12} & 0 \\ (Q_{12})_{21} & (Q_{12})_{22} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} (Q_{13})_{11} & 0 & (Q_{13})_{12} \\ 0 & I & 0 \\ (Q_{13})_{21} & 0 & (Q_{13})_{22} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & B_{22} & B_{23} & B_{24} \\ 0 & B_{32} & B_{33} & B_{34} \end{pmatrix} \\
&= \begin{pmatrix} (Q_{12})_{11} & (Q_{12})_{12} & 0 \\ (Q_{12})_{21} & (Q_{12})_{22} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} (Q_{13})_{11} & 0 & (Q_{13})_{12} \\ 0 & I & 0 \\ (Q_{13})_{21} & 0 & (Q_{13})_{22} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & (Q_{23})_{11} & (Q_{23})_{12} \\ 0 & (Q_{23})_{21} & (Q_{23})_{22} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & B_{33} & B_{34} \end{pmatrix} \\
&= \begin{pmatrix} (Q_{12})_{11} & (Q_{12})_{12} & 0 \\ (Q_{12})_{21} & (Q_{12})_{22} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} (Q_{13})_{11} & 0 & (Q_{13})_{12} \\ 0 & I & 0 \\ (Q_{13})_{21} & 0 & (Q_{13})_{22} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & (Q_{23})_{11} & (Q_{23})_{12} \\ 0 & (Q_{23})_{21} & (Q_{23})_{22} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & Q_{33} \end{pmatrix} \\
&\quad \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \end{pmatrix}
\end{aligned}$$

Clearly

$$R = \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \end{pmatrix}$$

is an upper triangular matrix. So we just need to check that

$$
\begin{aligned}
Q = &\begin{pmatrix} (Q_{12})_{11} & (Q_{12})_{12} & 0 \\ (Q_{12})_{21} & (Q_{12})_{22} & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} (Q_{13})_{11} & 0 & (Q_{13})_{12} \\ 0 & I & 0 \\ (Q_{13})_{21} & 0 & (Q_{13})_{22} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & (Q_{23})_{11} & (Q_{23})_{12} \\ 0 & (Q_{23})_{21} & (Q_{23})_{22} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & Q_{33} \end{pmatrix} \\
=&Q_1Q_2Q_3Q_4
\end{aligned}
$$

is unitary. Notice that each Q_i is unitary. Hence

$$
\begin{aligned}
Q^*Q &= (Q_1Q_2Q_3Q_4)^*Q_1Q_2Q_3Q_4 \\
&= Q_4^*Q_3^*Q_2^*Q_1^*Q_1Q_2Q_3Q_4 \\
&= I
\end{aligned}
$$

In other words Q is unitary and our algorithm successfully finds a QR factorization of A.

4.1 Implementation

While on the surface our algorithm appears to have only two sub functions, performing QR factorization on a block and factoring out the Q from the other blocks in the row, the second operation can be broken down into two more basic sub functions. Since Q is unitary if

$$A = QR \quad \text{then} \quad Q^*A = R.$$

So our algorithm really needs three main sub functions, WY QR factorization, matrix transpose and matrix matrix multiply.

Notice that the majority of the work in this algorithm is done by matrix multiplication. After WY performs we need to update both the rows to the right of the column being operated on as well as the relevant columns of the matrix Q. Both updates require matrix multiplication. So it makes sense when optimizing BlockedQR to first optimize matrix matrix multiplication. For this reason we initially wrote tiled QR version one using a blocked matrix matrix multiplication subprogram. However when we ran perf we found that eighty percent of the work was being done blocking and unblocking submatrices. In addition we realized that because we picked blocked size 16 to optimize the WY code, blocking the matrix multiply code was pointless. So we replaced the blocked matrix matrix multiply code with the simple three loop version. To take advantage of pipelining in the matrix multiplication code we ordered the loops j, k, i.

5 Tiled QR Factorization: Version 2

Version two of our tiled QR algorithm introduces parallelization on the column updates, i.e. when we zeroed out blocks below the diagonal. However unlike the row updates that version one introduced, zeroing out blocks below the diagonal can not be done fully in parallel. To get around this we broke the blocks in the column into groups made up of succesively smaller 2^p blocks as shown in the diagram below.

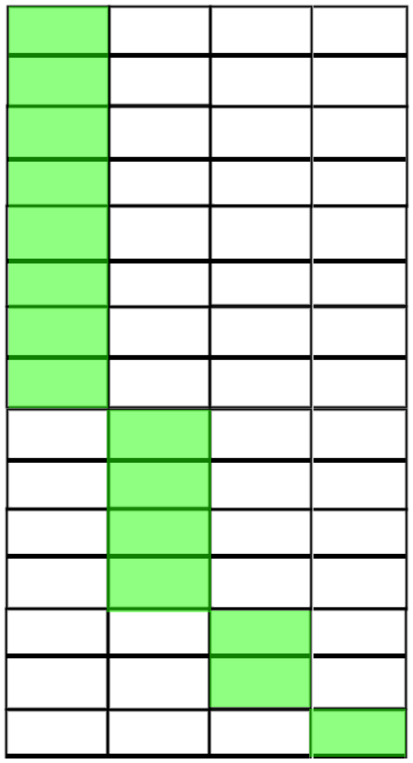
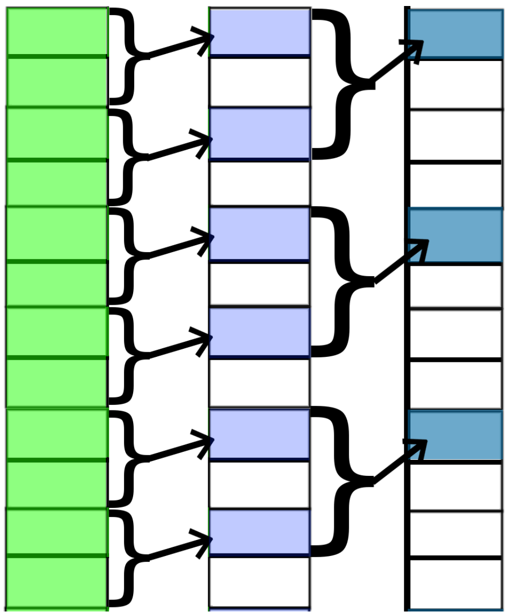


Figure 1: This diagram denotes one column being broken up and zeroed out over multiple iterations. Notice that the each green set consists of the largest power of two number of blocks of those blocks in the column remaining to be zeroed out.

We zero out (or merge) each green set using a binary tree. Each level of the binary tree is zeroed out in parallel as see in the diagram below.



Finally we update the block k of the column, zeroing out the root of each binary tree below block k.

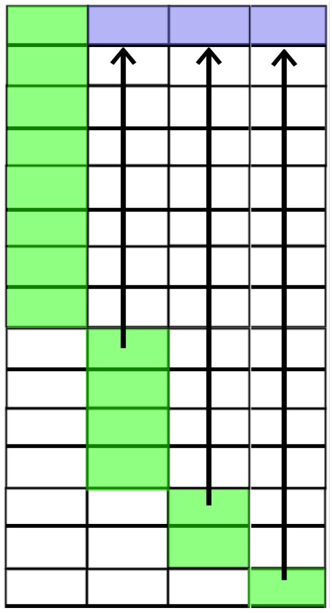


Figure 2: Note: The first block in this example is block k.

Repeating this procedure for each column yeilds the algorithm given below.

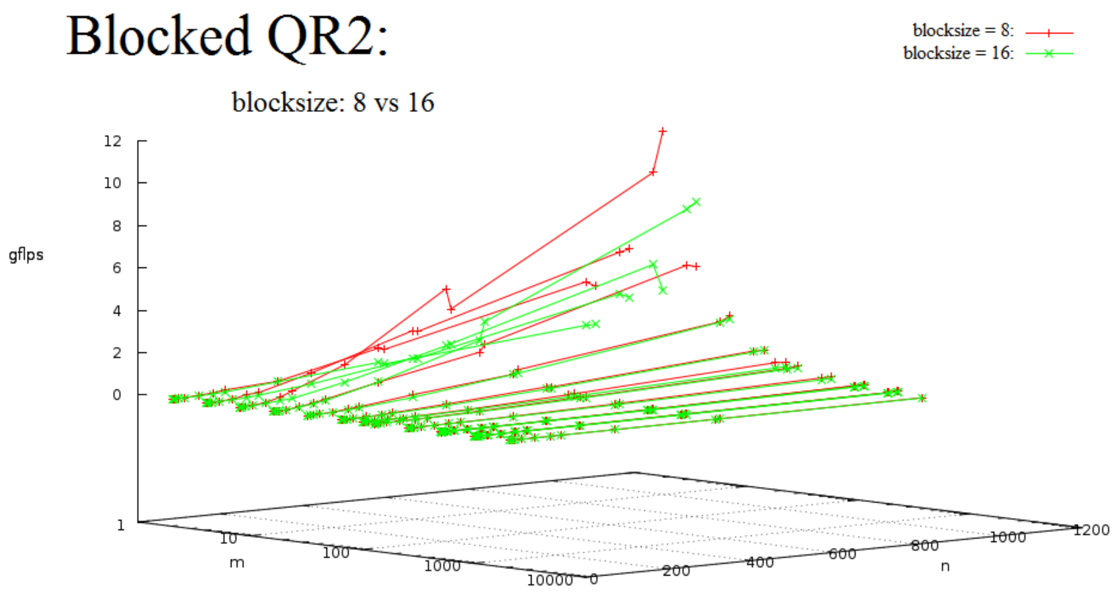
Algorithm 5.1 Tiled QR Factorization: Version 2

```
$Q = I$
for $k = 1$ to $\min(hn, wn)$ do
    Initialize variables: boundary = $k$
    while boundary < hn do
        Compute the largest $c$ such that for an integer $p$, $c = 2^p <$ hn - boundary + 1
        Initialize variables $c1 = 1$ and $c2 = 2$.
        while c2 $\leq c$ do
            OMP PARALLEL FOR LOOP:
            for $i =$ boundary; $i <$ boundary $+c - c1$ ; $i+ = c2$ do
                Preform $QR$ factorization on $\begin{bmatrix} A_{i,k} \\ A_{i+c1,k} \end{bmatrix}$ to get $\begin{bmatrix} R_{i,k} \\ 0 \end{bmatrix}$ and $Q_{i,i+c1}$
                Update $Q$ matrix
                OMP PARALLEL FOR LOOP:
                for $j = k + 1$ to $wn$ do
                    Factor out $Q_{i,i+c1}$ from $\begin{bmatrix} A_{ij} \\ A_{i+c1,j} \end{bmatrix}$ to get $\begin{bmatrix} B_{ij} \\ B_{i+c1,j} \end{bmatrix}$.
                end for
            end for
            Update variables $c1* = 2$, $c2* = 2$.
        end while
        if boundary > 0 and $k \neq$boundary then
            Preform $QR$ factorization on $\begin{bmatrix} A_{k,k} \\ A_{\text{boundary},k} \end{bmatrix}$ to get $\begin{bmatrix} R_{k,k} \\ 0 \end{bmatrix}$ and $Q_{k,\text{boundary}}$
            Update $Q$ matrix
            OMP PARALLEL FOR LOOP:
            for $j = k + 1$ to $wn$ do
                Factor out $Q_{k,\text{boundary}}$ from $\begin{bmatrix} A_{kj} \\ A_{\text{boundary},j} \end{bmatrix}$ to get $\begin{bmatrix} B_{kj} \\ B_{k,\text{boundary}} \end{bmatrix}$.
            end for
        end if
        boundary += c
    end while
    if $k = hn - 1$ and $hn = \min(hn, wn)$ then
        Preform $QR$ factorization on $A_{kk}$, resulting in $Q_{kk}$ and $R_{kk}$
        Update $Q$
        OMP PARALLEL FOR LOOP:
        for $j = k + 1$ to $wn$ do
            Factor out $Q_{kk}$ from $A_{kj}$ to get $B_{kj}$.
        end for
    end if
end for
```

Notice the only difference between version 1 and version 2 is order that the blocks below the diagonal are zeroed out. Hence mathematically they are the same and since version one correctly performs QR decomposition, so will version two.

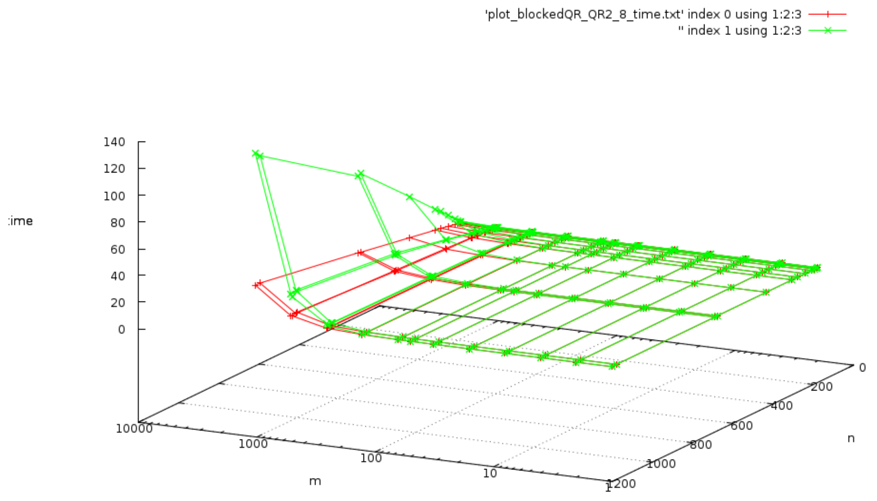
6 Blocked QR Performance:

In the preformance section on WY we saw that the optimal block size was 8 by 8. However in homework 6 we saw that the optimal block size to use the full C1 cache size was 16 by 16. We plotted the GFlops per second required to run Blocked QR version 2 for both blocked sizes on Bowery to see which block size optimizes our algorithm.

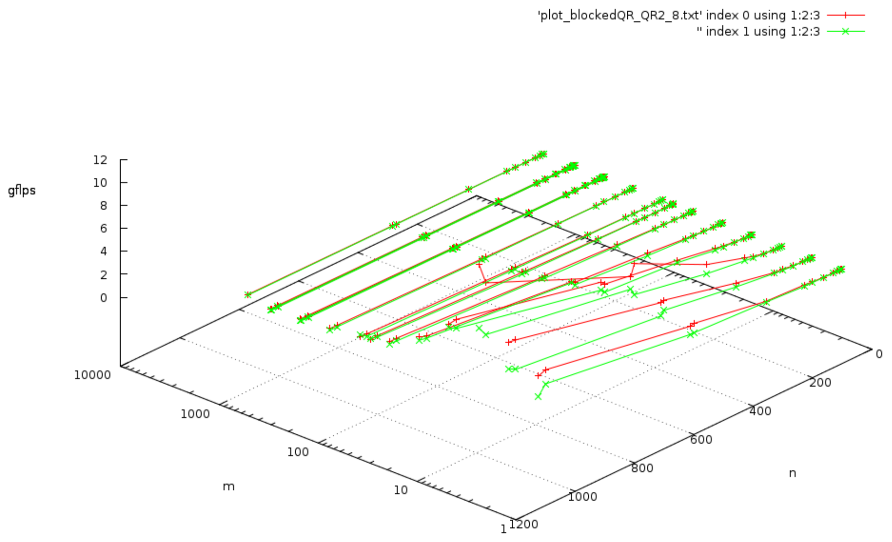


As you can see the block size $b = 8$ is clearly faster than the blocksize $b = 16$. This seems to mean that the code is dominated by the WY algorithm, not the matrix multiplication, even though perf tell us that most of the time is being spent in matrix multiplication. For the rest of our preformance study we will use block size equal to eight.

Next we compared version one to version to see what performance improvement if any we get from updating version one with more parallelization.



Observe that for $m \geq 1000$ version two becomes significantly better than version one in terms of seconds on Bowery. To compare performance in terms of GFlops per second we look at the next graph.



From this graph we see that version two either performs better or the same as version one. This makes sense because we are doing the same amount of work in both versions. The second version simply performs faster in general then the first. Hence updating version one with additional parallelization is an worth while update.

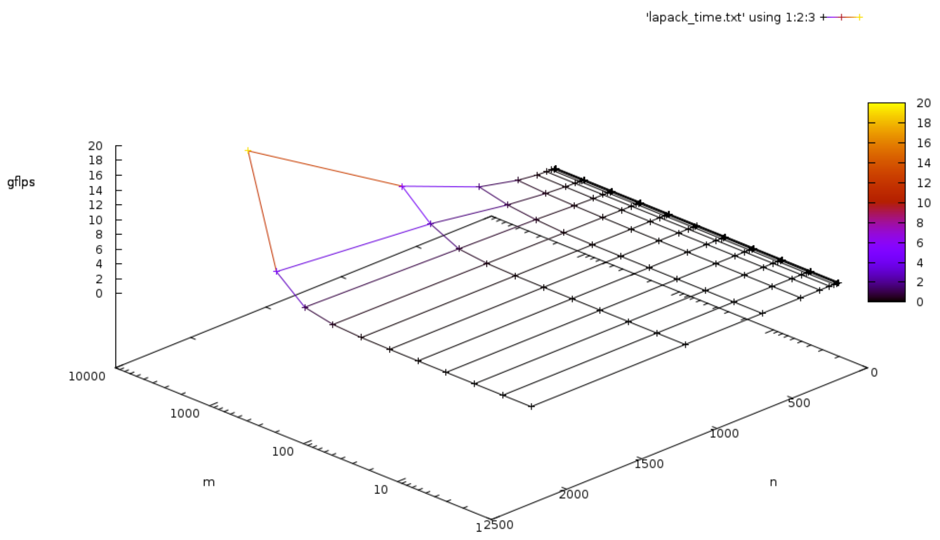
Now we consider the scaling measurement. The following list the time in seconds and the GFlops per second required to run Blocked QR version two over a different number of threads for a 2,048 by 2,048 matrix.

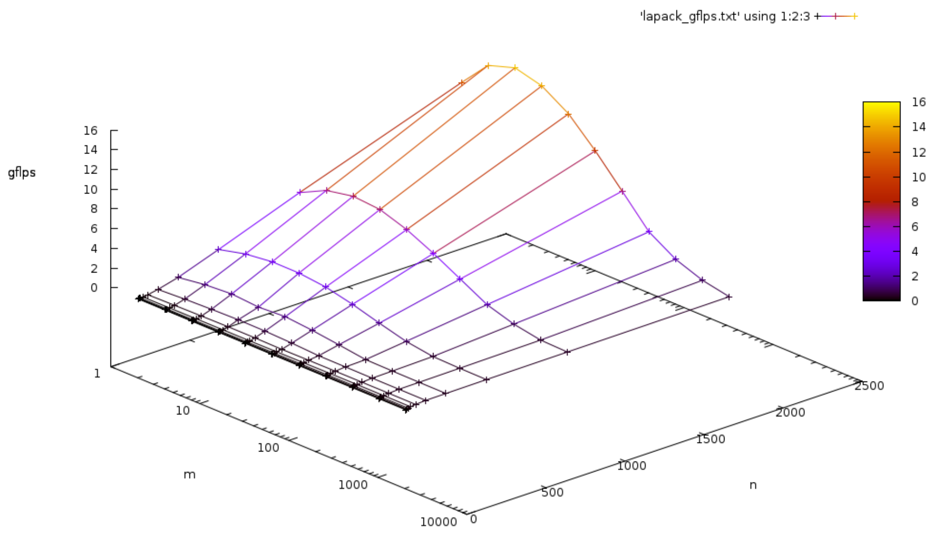
SCALING MEASUREMENTS:

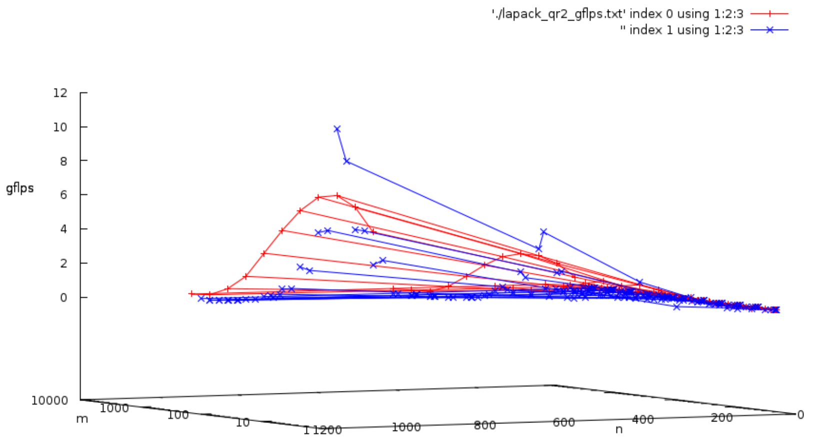
Number of threads	time (s)	Glfps
1	238.951351	0.035948466
2	273.221443	0.03143946
3	227.997841	0.037675508
4	210.125298	0.040880059
5	198.128890	0.043355286
6	189.512168	0.04532656
7	186.347311	0.04609637
8	188.394108	0.045595559

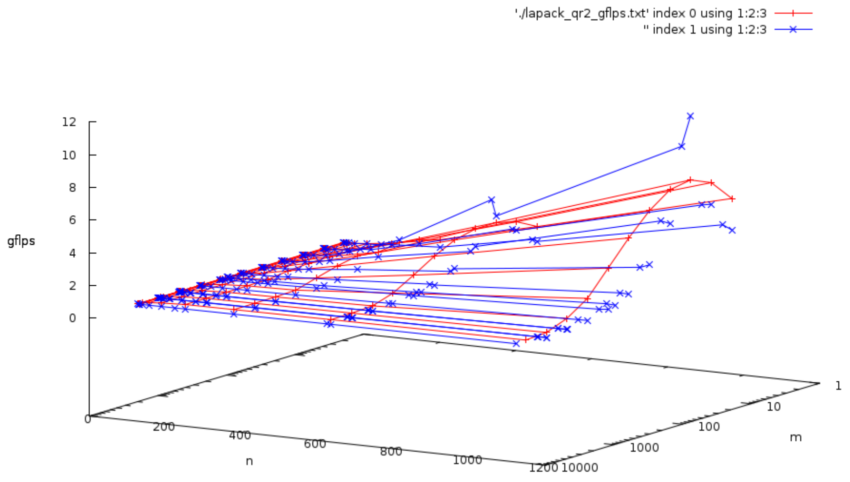
The addition of more threads aids the performance of our code but not by much, at least for very large 2048x2048 matrices. Perhaps we would have seen different numbers for smaller matrices or other combinations of m and n.

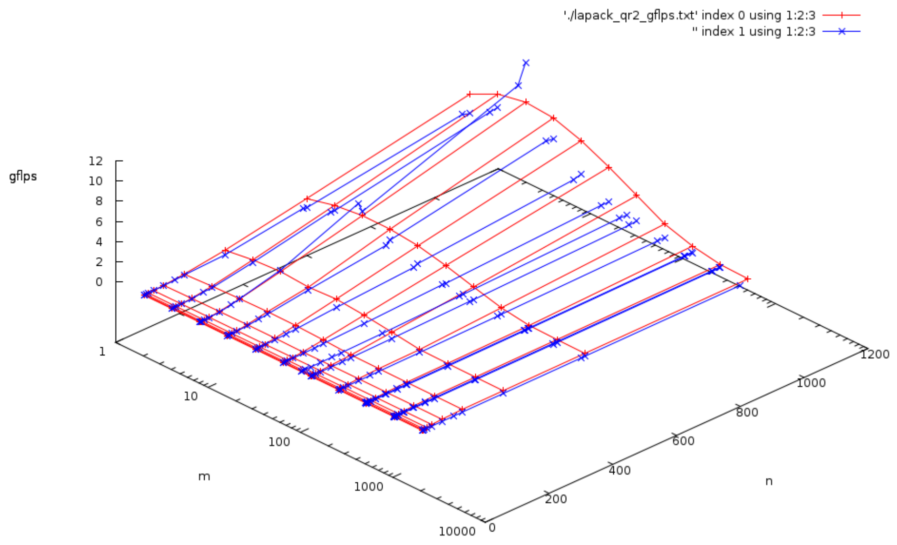
Finally, we consider and compare against LAPACK time and rate measurements. First we present two plots of LAPACK alone under various combination of m and n in terms of time and rate followed by 3 plots of the second QR algorithm compared against LAPACK.






'./lapack_qr2_gflps.txt' index 0 using 1:2:3
'' index 1 using 1:2:3
gflps
12
10
8
6
4
2
0
10000
1000
100
10
1
m
1200
1000
800
600
400
200
0
n


'./lapack_qr2_gflps.txt' index 0 using 1:2:3
'' index 1 using 1:2:3
gflps
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
n
10000
1000
100
10
1
m



These plots show that LAPACK across the board outperforms our implementation of the QR algorithm, except with regards to one case, where $m = 8$. This anomaly is most likely a result of some error in methodology or measurement as it is highly doubtful that we could have outperformed such a well-honed linear algebra library.

7 Conculsion

Further work could explore implementing an alternative algorithm that is optimized for tall skinny matrices (m far greater than n) and employed selectively depending upon the input. On the hand, the implementation does work relatively well for fat short matrices (m far less than n), and this is because when the input matrices are short and fat, updating Q is always a lot less work. The first version of our QR algorithm had poor performance for tall, thin matrices, which inspired the second QR algorithm, whose performance shows improvement for large values of m and small values of n.

Within the current implementation alone, a bottleneck exists in which the workload is not evenly distributed amongst threads. In retrospect it would not be too difficult to modify the binary tree algorithm so that the first tree is at least not monstrously larger than all the subsequent trees assigned to other threads. Ideally, it would be nice if all the trees could be of about the same size.

An additional use of blocking in a second layer between the input matrices and the WY algorithm could prove beneficial if the individual tiles were of a size targeted for L2 cache. At the moment there is no targeting of the L2 cache at all, but only L1 cache through the optimization of the size of submatrices fed to the WY algorithm.

8 Andreas's Questions = ANSWERED

Note: Most of these (if not all) are answered in the main report. We just didn't want you to miss the answers.....

1. What are you doing? Revelvant liturature?

   - We are preforming a QR factorization on a random matrix A. Check References.

2. What is the scale of the problem you are aiming for? Liminatations?

   - Our code performs for matrices smaller than 5000 by 5000. However for matrices smaller than 130 by 130 perf reports that most of the work in BlockedQR2 is being done by the OMP library. If the matrix is bigger than 5000 it might crash. On the virtual machine the 5000 by 5000 case ran in 2451.923687 s or approximately 40 minutes.

3. Describe existing work/software on your topic. What codes if any did you look at?

   - Since QR decompositions are useful for computing harder problems such as least square, alot of work has already been done on this problem. There are QR implementations in Lapak, Plasma, and Flame to name a few linear algebra libraries. We studied a paper describing the implementation of Tiled QR factorization in Plasma. We also looked at the code in Lapak but gave up because we couldn't figure out what they were doing.

4. Performance expectations: What was the most time consuming step?

   - In the performance study on WY. We saw that WY was optimized by block size $b = 8$. In hw 6 we saw that matrix multiplication was optimized by block size 16. We thought that the matrix multiplication part of our code was the most time consuming step. However in the BlockedQR Performance section we saw that the code preformed better with block size 8. Hence the WY section of our code must be the most time consuming.

5. Performance expectations: Which is the hardest to parallelize?

    - When we began to work on a tiled version of QR factorization. Applying the row updates in parallel was obvious. However performing the updates along the column was much harder. - See section Tiled QR : Version 2 for more information.

6. Performance expectations: Relate this to your scaling discussion?

    - By figuring out how to run both the row and column updates in parallel we were able to compute QR factorizations for larger matrices. For instance if we compute the QR factorization for a 1000 by 1000 matrix in version one on the virtual machine it takes around 45 seconds. However when we compute the QR factorization for the same size matrix in version 2 it only take around 25 seconds. Hence we have cut the time in two by adding another layer of parallelization.

7. Describe Preformance and scaling measurements.

    - We discussed the performance of each version, WY, Version One and Version Two in their respective performance section in the paper. Scaling was discussed for both version one and version two.

8. Where and how have you made your code available?

- We have made it available in the forge respository and it will be on github shortly after submission at git://github.com/pat227/hpc12-fp-jbb383-pat227.git.

9. Instructions on how to build and run your code.

   Provided you have all the sublibraries, makefile, and headers (there are quite a few) our code compiles on any platform (we think). It definitely runs on Bowery and in linux... Calling make will build our code. Now you have four options for running our code. They all have the same arguments m = height of matrix, n = width of matrix, iterations = how many times you want to repeat code (normally 1), test = 0 if you don't want to test code, test= 1 if you want to test code. Note: If you run the code with test on, the test run time is included in the timing of the code.

   ./householder m n iterations test, will run the navie householder case by itself ./wy m n iterations test, will run the wy case by itself ./BlockedQR m n iterations test, will run Tiled QR version 1 ./BlockedQR2 m n iterations test, will run Tiled QR version 2

   You can enter any m or n.