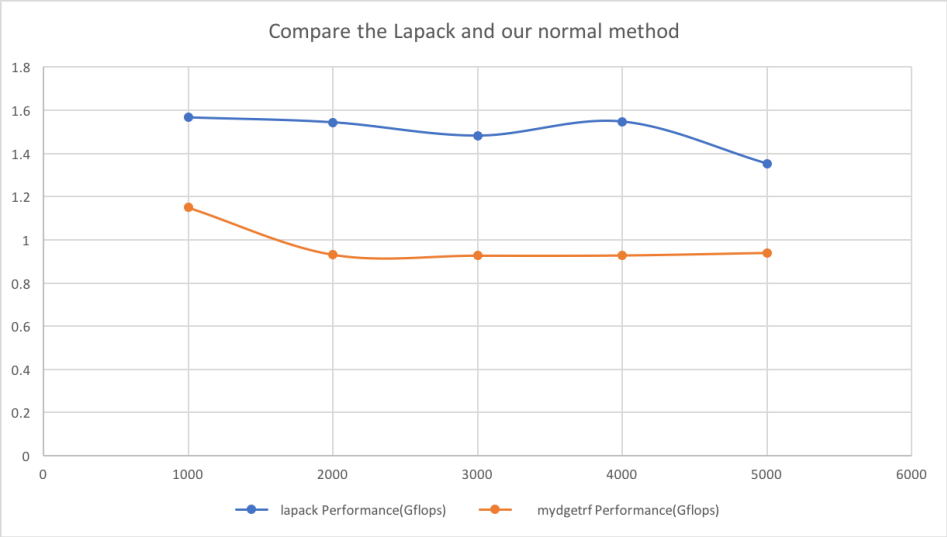
Project 2

Part 1

First, I want to compare the two versions of LU factorization. They are using cmake 2.8 and g++ 4.7.2, both are using default compile flags and the lapack is compiled from the source with the integrated BLAS library.

Here is the result:

Size	lapack Performance (Gflops)	mydgetrf Performance (Gflops)	lapack Time (s)	mydgetrf Time (s)
1000	1.566502	1.148916	0.425257	0.579822
2000	1.543587	0.931408	3.453859	5.723948
3000	1.482189	0.927059	12.141161	19.411387
4000	1.54727	0.9282	27.570282	45.958471
5000	1.353224	0.939804	61.572101	88.657706



I should notice that the performance(Gflops) calculation:

For i = 1 to n -1
    For j = i+1 to n
        Divide - 1 double calculation
        For k = i+1 to n
            Multiply and Add – 2 double calculations

Float point operators for Divide:

$$(n-1)+(n-2)+\cdots+1=\frac{(n-1)n}{2}$$

Float point operators for Multiply and Add:

$$[(\mathrm{n}-1)^2+(n-2)^2+\cdots+1^2]\times 2=\frac{n(n-1)(2n-1)}{3}$$

So, the performance should be:

$$\mathrm{P}=\left[\frac{(\mathrm{n}-1)\mathrm{n}}{2}+\frac{n(n-1)(2n-1)}{3}\right]/time$$

Second, I will introduce my method to implement the code. That has three steps:

1. Pivoting
2. LU factorization
3. Calculate the Ly = b and Ux = y

First two part is listed below. There isn't too much different between this version and the Matlab version.

```
#define A(x, y) a[(x)*n + (y)]
/*
 * This is the normal version for LU factorization
 * n - matrix size
 * a - input matrix A (output will replace that matrix)
 * pvt - used for order transform
 */
int mydgetrf(int n, double* a, int* pvt) {
    for (int j = 0; j < n; ++j) pvt[j] = j;
    for (int i = 0; i < n-1; ++i) {
        int maxind = i;
        double maxa = fabs(A(i, i));
        for (int t = i + 1; t < n; ++t) {
            if (fabs(A(t, i)) > maxa) {
                 maxa = fabs(A(t, i));
                 maxind = t;
            }
        }
        if (maxa == 0.0) {
            return -1;
        } else {
            if (maxind != i) {
                int temps = pvt[i];
                pvt[i] = pvt[maxind];
                pvt[maxind] = temps;

                double tempv;
                for (int j = 0; j < n; ++j) {
                    tempv = A(i, j);
                    A(i, j) = A(maxind, j);
                    A(maxind, j) = tempv;
                }
            }
        }
    }
    for (int i = 0; i < n-1; ++i) {
        for (int j = i+1; j < n; ++j) {
            A(j, i) /= A(i, i);
            for (int k = i+1; k < n; ++k)
                A(j, k) -= A(j, i) * A(i, k);
        }
    }
    return 0;
}
```

The second part is the function `mydtrsm`, used to solve the equations:

```
enum dtrsm_type {
    Lower = 0,
    Upper
};
/*
 * normal dtrsm for calculate Ax = B
 * t - the triangle type (Lower or Upper)
 * n - the matrix size
 * a - the input A matrix
 * b - the input B vector
 * pvt is used for order transform
 */
double* mydtrsm(dtrsm_type t, int n, double* a, double* b, int* pvt) {
    if (t == Lower) {
        double* y = new double[n];
        y[0] = b[pvt[0]];
        for (int i = 1; i < n; ++i) {
            double sum = b[pvt[i]];
            for (int j = 0; j < i; ++j) {
                sum -= y[j] * A(i, j);
            }
            y[i] = sum;
        }
        return y;
    } else {
        double* x = new double[n];
        x[n-1] = b[n-1] / A(n-1, n-1);
        for (int i = n-2; i >= 0; --i) {
            double sum = b[i];
            for (int j = i + 1; j < n; ++j) {
                sum -= x[j] * A(i, j);
            }
            x[i] = sum / A(i, i);
        }
        return x;
    }
}
```

Part2

Firstly, I used my local computer to find the best block size for the program:

Because my matrix multiply function use the best block size 30, obviously, the several times of 30 will be better choice. Finally, 120 become the best block size. Those tests are under the data size N = 2000.

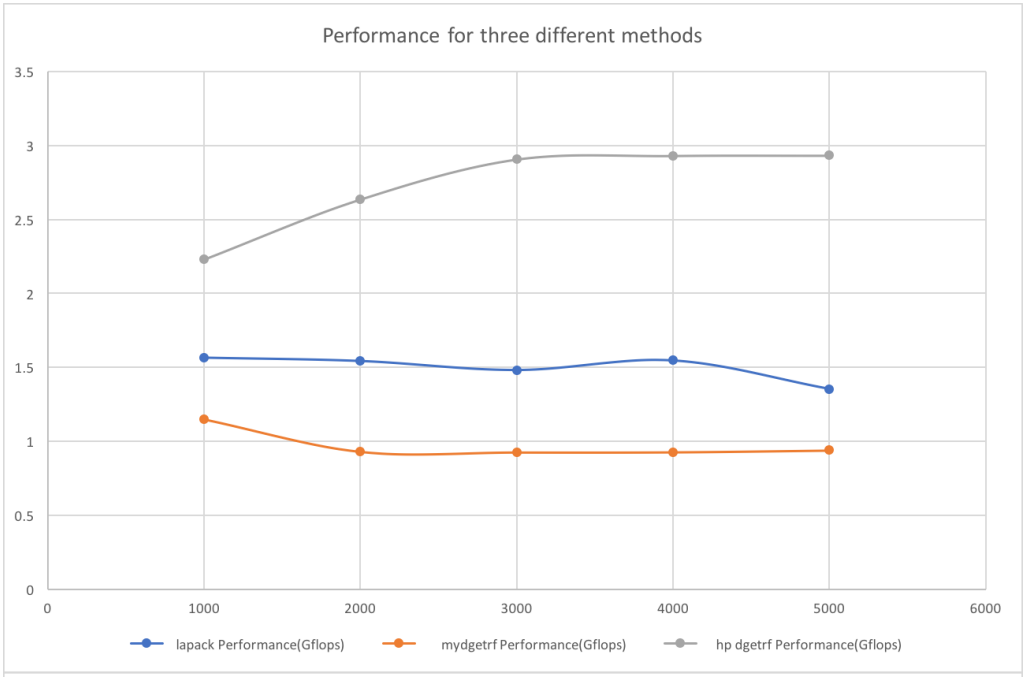
Compare with others, this method is also much better, the benchmarks listed below are all tested on our cluster with one node.

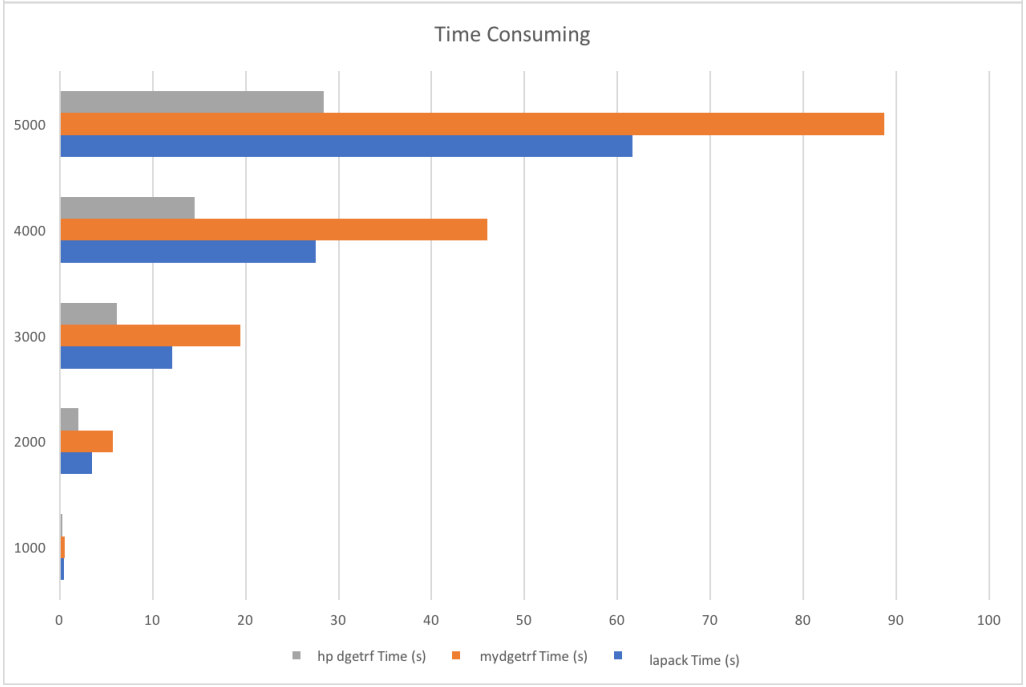
Here is the performance:

Size	lapack Performance(Gflops)	mydgetrf Performance(Gflops)	hp_dgetrf Performance(Gflops)
1000	1.566502	1.148916	2.229051
2000	1.543587	0.931408	2.635332
3000	1.482189	0.927059	2.906329
4000	1.54727	0.9282	2.930502
5000	1.353224	0.939804	2.932302

And here is the time:

Size	lapack Time (s)	mydgetrf Time (s)	hp dgetrf Time (s)
1000	0.425257	0.579822	0.298857
2000	3.453859	5.723948	2.023021
3000	12.141161	19.411387	6.191831
4000	27.570282	45.958471	14.55678
5000	61.572101	88.657706	28.414824





In the end, I will introduce the code structure for this high-performance method.
The first part is still pivoting. Then we jump each B steps, and for each step, we will calculate the answer with normal method first for the narrow rectangle. Then after using inv_triangle to calculate the inverse of matrix LL, multiply it with the right matrix `A(ib:end, end+1:n)`.
Finally, we update the big matrix `A(end+1:n , end+1:n)`.

```
/**
 * This is the basic matrix multiply function in project 1
 * a & b is the input matrix A and B
 * c is the target matrix C (must be all zero before calling)
 * n, n1, n2 is the the matrix size (A: n x n1  B: n1 x n2  C: n x n2)
 * la, lb, lc is the matrix width, because when you want to calculate a sub-matrix,
 *   that will be useful to indicate the real size of the matrix.
 */
void
dgemm_mixed(double *a, double *b, double *c,
            unsigned int n, unsigned int n1, unsigned int n2,
            unsigned int la, unsigned int lb, unsigned int lc);


inline void
copy_matrix(double *a, double *b, unsigned int n, unsigned int m, unsigned int la,
unsigned int lb) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            a[i*la+j] = b[i*lb+j];
}


inline void
minus_matrix(double *a, double *b, unsigned int n, unsigned int m, unsigned int la,
unsigned int lb) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            a[i*la+j] -= b[i*lb+j];
}


void inv_triangle(int n, double* a) {
    for (int i = 1; i < n; ++i)
        A(i,0) = -A(i,0);
    for (int k=1; k<n-1; ++k)
        for(int i = k+1; i < n; ++i) {
            for(int j = 0; j < k; j++)
               A(i,j) -= A(k,j) * A(i,k);
            A(i,k) = -A(i,k);
        }
}
#define LL(x, y) ll[(x)*B + (y)]
```

```
/**
 * the highest performance version (Blocked GEPP)
 * n - matrix size
 * a - input/output matrix
 * pvt - used for order transform
 */
int hp_dgetrf(int n, double* a, int* pvt, const int B) {
    for (int j = 0; j<n; ++j) pvt[j] = j;

    for (int i = 0; i < n-1; ++i) {
        int maxind = i;
        double maxa = fabs(A(i, i));
        for (int t = i + 1; t < n; ++t) {
            if (fabs(A(t, i)) > maxa) {
                 maxa = fabs(A(t, i));
                 maxind = t;
            }
        }
        if (maxa == 0.0) {
            return -1;
        } else {
            if (maxind != i) {
                int temps = pvt[i];
                pvt[i] = pvt[maxind];
                pvt[maxind] = temps;

                double tempv;
                for (int j = 0; j < n; ++j) {
                    tempv = A(i, j);
                    A(i, j) = A(maxind, j);
                    A(maxind, j) = tempv;
                }
            }
        }
    }

    int m = n/B*B; int m1 = (n-1)/B*B;
    double* ll = createMatrix(B, B);
    double* temp = createMatrix(n, n);

    int i;
    for (i = 0; i < m1; i += B) {
        int end = i+B;
        // apply BLAS2 version to get A(i:n, i:i+B)
        for (int t = i; t < end; ++t)
            for (int j = t+1; j < n; ++j) {
                A(j, t) /= A(t, t);
                for (int k = t+1; k < end; ++k)
                    A(j, k) -= A(j, t) * A(t, k);
            }
```

```
        // get LL
        for (int p = 0; p < B; ++p)
            for (int q = 0; q < B; ++q)
                if (p == q) LL(p, q) = 1;
                else if (p < q) LL(p, q) = 0;
                else LL(p, q) = A(i+p, i+q);
        inv_triangle(B, ll); // LL^-1
        memset(temp, 0, (n-end)*B*sizeof(double));
        // LL^-1 * A(ib:end , end+1:n)
        dgemm_mixed(ll, &A(i, end), temp, B, B, n-end, B, n, n-end);
        // update A(ib:end , end+1:n)
        copy_matrix(&A(i, end), temp, B, n-end, n, n-end);
        memset(temp, 0, (n-end)*(n-end)*sizeof(double));
        // update A(end+1:n , end+1:n)
        dgemm_mixed(&A(end, i), &A(i, end), temp, n-end, B, n-end, n, n, n-end);
        minus_matrix(&A(end, end), temp, n-end, n-end, n, n-end);
    }
    for (; i < n-1; ++i) { // continue to do the unfinished part
        for (int j = i+1; j < n; ++j) {
            A(j, i) /= A(i, i);
            for (int k = i+1; k < n; ++k)
                A(j, k) -= A(j, i) * A(i, k);
        }
    }
    free(ll); free(temp);
    return 0;
}
```