МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»
Профиль подготовки: «Вычислительная математика и суперкомпьютерные технологии»

Отчет по лабораторной работе №1

«Блочное LU разложение»

Выполнил: студент группы 381903-3м
_____ Панов А.А.
Подпись

Нижний Новгород
2020

Содержание

Введение

LU разложение – представление матрицы A в виде A=LU. L – нижняя треугольная матрица с диагональными элементами, равными единице, а U – верхняя треугольная матрица с ненулевыми диагональными элементами.

$$L = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ l_{1,n} & \cdots & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{1,1} & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_{n,n} \end{pmatrix}$$

LU разложение используется для решения СЛАУ вида Ax = b. LU разложение особенно полезно если имеется «несколько правых частей». Алгоритм решения в таком случае следующий:

1. Находится LU разложение матрицы A:
2. Для каждой «правой части» b_i из B:
   2.1. $LUx_i = b_i$ // вид системы после подстановки
   2.2. $Ly_i = b_i$ // L – нижняя треугольная матрица, y_i ищется обратным ходом Гаусса
   2.3. $Ux_i = y_i$ // U – верхняя треугольная матрица, x_i ищется обратным ходом Гаусса

LU разложение достаточно вычислительно сложная задача (алгоритмическая сложность пропорциональна n^3) и для её эффективного решения полезно использовать параллельные вычисления.

1. Постановка задачи

Реализовать блочное LU-разложение для квадратной матрицы, используя технологию OpenMP, то есть представить матрицу A в виде произведения двух матриц: A=LU, где L — нижняя треугольная матрица, а U — верхняя треугольная матрица.

Программа на языке C++ должна реализовывать функцию со следующим заголовком:

```
void LU_Decomposition(double * A, double * L, double * U, int n);
```

Функция получает в аргументах следующие переменные:
A – указатель на массив, в котором по строкам хранится матрица A размера n×n
n – размерность матрицы

L – указатель на массив, в котором по строкам необходимо записать матрицу L размера n×n,

U – указатель на массив, в котором по строкам необходимо записать матрицу U размера n×n.

Ответ считается корректным, если:

$$\frac{\|LU - A\|_2}{\|A\|_2} < 0.01$$

Размерность матрицы n ≤ 3000.

2. Алгоритм LU разложения и его свойства

2.1 Связь метода Гаусса и LU разложения

Рассмотрим СЛАУ вида Ax = b, с невырожденной матрицей A. С помощью метода Гаусса систему можно представить в виде: Ux = y = L^{-1}b

Ux = L^{-1}b, умножим обе части на L

LUx = b

Таким образом коэффициенты U можно получить в результате применения к матрице A метода Гаусса. Матрица L будет состоять из коэффициентов метода Гаусса. Существует теорема (доказательство можно посмотреть в «Численных методах» В. Б. Андреева), согласно которой LU-разложение существует и единственно, если главные миноры матрицы A отличны от нуля. Также для любой невырожденной матрицы A существует матрица перестановок P такая, что матрица PA имеет ненулевые главные миноры. Следовательно систему Ax = b можно представить в виде PAx = LUx = Pb.

2.2 Алгоритмическая сложность

Вычисление коэффициентов L и U выполняется по следующим формулам:

$$u_{ij} = \left[a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\right] \quad i = 1, \ldots, n; \quad j = i, \ldots, n;$$

$$l_{ij} = \frac{1}{u_{jj}}\left[a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right] \quad j = 1, \ldots, n; \quad i = j+1, \ldots, n$$

Формула 1. Коэффициенты матриц LU

Количество умножений и делений можно подсчитать по следующей формуле:

$$Q = \sum_{i=1}^{n}\sum_{j=i}^{n}(i-1) + \sum_{j=1}^{n}\sum_{i=j+1}^{n} j = \sum_{i=1}^{n}[(i-1)(n-i+1) + i(n-i)] =$$

$$= 2\sum_{i=1}^{n}[(n+1)i - i^2] - n(n+1) = n(n+1)^2 - \frac{n(n+1)(2n+1)}{3} - n(n+1) =$$

$$= \frac{n(n^2-1)}{3} = \frac{n^3}{3} + O(n) \approx \frac{n^3}{3}$$

Для получения LU разложения потребуется $O(n^3)$ умножений и делений.

3. Наивная реализация

Реализация LU разложения по формуле 1 достаточно простая. При этом код легко параллелится.

```
#define indx(i, j, n) ((i)*(n)+(j))
for (int ved = 0; ved < n; ved++)
{
    double el = U[indx(ved, ved, n)];
    #pragma omp parallel for
    for (int i = ved + 1; i < n; i++)
    {
        double coeff = U[indx(i, ved, n)] / el;
        L[indx(i, ved, n)] = coeff;
        #pragma ivdep
        for (int j = ved + 1; j < n; j++)
            U[indx(i, j, n)] -= coeff * U[indx(ved, j, n)];

    }
}
```

Недостаток данного метода – плохое использование кэш памяти процессора.

3.1 Погрешность решения

Пусть на компьютере выполнено LU разложение с ошибкой вычислений ε_m: $A + \delta A = LU$, где δA эквивалентное возмущение. Тогда выполняется неравенство: $\|\delta A\| \leq n\varepsilon_m \|L\|\|U\| + O({\varepsilon_m}^2)$

4. Блочная реализация

Чтобы улучшить работу с кэшем процессора представим исходную матрицу A и искомые матрицы L и U в следующем виде (на рисунке ниже r это размер блока):

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \qquad U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix}$$

$$\begin{matrix} r & n-r \end{matrix} \qquad \begin{matrix} r & n-r \end{matrix} \qquad \begin{matrix} r & n-r \end{matrix}$$

Подставив в формулу A = LU получим:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

Алгоритм решения в таком случай стовится таким:

1. Находим L_{11}, U_{11} используя стандартное LU разложение
2. Находим U_{12} из СЛАУ с несколькими правыми частями $A_{12} = L_{11}U_{12}$
3. Находим L_{21} из СЛАУ с несколькими правыми частями $A_{21} = L_{21}U_{11}$
4. Находим матрицу $\tilde{A}_{22} = A_{22} - L_{21}U_{12}$, $\tilde{A}_{22} = L_{22}U_{22}$, далее алгоритм применяется рекурсивно.

Так как матрицы L_{11} и U_{11} треугольные, то для поиска U_{12} и L_{21} можно применить обратный ход Гаусса. Алгоритмическая сложность первого этапа алгоритма $O(r^2)$, второго и третьего $O((n-kr)r^2)$ на k шаге, третьего $O(r(n-(k+1)r)^2)$. Всего шагов n/k, так что можно оценить сложность алгоритма как $O(n^3)$. Есть более точная оценка $\frac{2n^3}{3} + O(n^2)$. Можно заметить, что если $r << n$, то умножение матриц будет наиболее «трудоемкой» операцией. Общую долю матричных умножений можно оценить величиной $1 - (r/n)^2$.

4.1 Структуры данных

Матрицы A_{11}, A_{12}, … назовем «подматрицами» матрицы A. Для удобной работы с такими «подматрицами» была добавлена структура `BMatrix`. В данной структуре хранится размер «подматрицы», индекс первой строки и столбца внутри матрицы, размер «подматрицы» и размер всей матрицы. Также хранится индекс первого элемента матрицы. В данной структуре для удобного доступа к i,j элементу перегружен оператор (). Код данной структуры:

```
class BMatrix
{
    const int firstIndex;
    double *A;
    const int realSize;
    const int n, m;
    const int si, sj;
public:
    BMatrix(double *A, int rs, int n, int m, int si, int sj) : A(A), realSize(rs), n(n),
    m(m), si(si), sj(sj), firstIndex(indx(si, sj, rs)) {};
    __forceinline double &operator() (int i, int j)
    {
        return A[firstIndex + i * realSize + j];
    }
    __forceinline const double &operator() (int i, int j) const
    {
        return A[firstIndex + i * realSize + j];
    }
    __forceinline int row() const
    {
        return n;
    }
    __forceinline int col() const
    {
        return m;
    }
    int end_row() const
    {
        return row() + si;
    }
    int end_col() const
    {
        return col() + sj;
    }
};
```

4.2 Параллельная версия блочного алгоритма

Так как матричное умножение занимает большую часть вычислений, важно распараллелить его как можно эффективней. Для эффективной работы с кэшами матричное умножение выполняется «поблочно», с размером блока bs.

```
void blockMultMatrix(const BMatrix &mA, const BMatrix &mB, const int bs)
{
    #pragma omp parallel for
    for (int bi = 0; bi < mA.row(); bi += bs)
        for (int bj = 0; bj < mB.col(); bj += bs)
            for (int bk = 0; bk < mA.col(); bk += bs)
                for (int i = bi; i < MIN(bi + bs, mA.row()); i++)
                {
                    const double *A = mA.A + mA.firstIndex + i * realSize;
                    double *mThis = this->A + firstIndex + i * realSize;
                    const int bkmin = MIN(bk + bs, mA.col());
                    for (int k = bk; k < bkmin; k++)
                    {
                        const double *B = mB.A + mB.firstIndex + k * realSize;
                        const int bjmin = MIN(bj + bs, mB.col());
                        #pragma ivdep
                        for (int j = bj; j < bjmin; j++)
                            mThis[j] -= A[k] * B[j];
                    }
                }
}
```

Для повышения производительности от удобной индексации через круглые скобки пришлось отказаться.

Матричное умножение занимает более 90% времени работы LU разложения, поэтому этапы 1,2,3 можно реализовать довольно «наивным» способом. Код первого этапа приведен выше.

Код второго этапа:

```
void gaussU12(const BMatrix &L11, BMatrix &U12, const BMatrix &A12)
{
    #pragma omp parallel for
    for (int r = 0; r < A12.col(); r++)
    {
        #pragma ivdep
        for (int i = 0; i < U12.row(); i++)
            U12(i, r) = A12(i, r);

        for (int i = 1; i < U12.row(); i++)
        {
            #pragma novector
            for (int j = 0; j < i; j++)
                U12(i, r) -= L11(i, j) * U12(j, r);
        }
    }
   }
```

Код третьего этапа:

```
void gaussL21(BMatrix &L21, const BMatrix& U11, const BMatrix& A21)
{
#pragma omp parallel for
    for (int rr = 0; rr < A21.row(); rr++)
    {
        #pragma ivdep
        for (int j = 0; j < L21.col(); j++)
            L21(rr, j) = A21(rr, j);

        for (int j = 0; j < L21.col(); j++)
        {
            #pragma novector
            for (int k = 0; k < j; k++)
                L21(rr, j) -= L21(rr, k) * U11(k, j);
            L21(rr, j) /= U11(j, j);
        }
    }
   }
```

Код всего алгоритма:

```
void LU_Decomposition(double *A, double *L, double *U, int n) //block version
{
    const int bs = 32;
    for (int bi = 0; bi < n; bi += bs)
    {
        //этап 0, подготовка L11 и U11
        BMatrix A11(A, n, MIN(n - bi, bs), MIN(n - bi, bs), bi, bi),
                L11(L, n, MIN(n - bi, bs), MIN(n - bi, bs), bi, bi),
                U11(U, n, MIN(n - bi, bs), MIN(n - bi, bs), bi, bi);
        //этап 1, поиск L11 и U11
        LU(L11, U11, A11);

        //этап 2 поиск U12
        BMatrix U12(U, n, MIN(n - bi, bs), n - U11.end_col(), bi, bi + bs),
                A12(A, n, MIN(n - bi, bs), n - A11.end_col(), bi, bi + bs);

        gaussU12(L11, U12, A12);

        //этап 3 поиск L21
        BMatrix L21(L, n, n - L11.end_row(), MIN(n - bi, bs), bi + bs, bi),
                A21(A, n, n - A11.end_row(), MIN(n - bi, bs), bi + bs, bi);
        gaussL21(L21, U11, A21);

        //этап 4 A -= L21 U12
        BMatrix A22(A, n, n - A11.end_row(), n - A11.end_col(), bi + bs, bi + bs);
        A22.blockMultMatrix(L21, U12, 32);
    }
```

5. Вычислительные эксперименты

LU разложение выполнялось для матрицы вида AA^T (матрица Грама), данная матрица положительна определена если $|A| \neq 0$ => если $|A| \neq 0$ для AA^T существует LU разложение. Для проверки корректности произведение LU сравнивается с AA^T по манхэттенской норме ($||A||_1$).

$$\|A\|_1 = \max_j \left\{ \sum_{i=1}^{n} | a_{ij} | \right\}$$

Запуск производился на восьми ядерном процессоре Intel core i7 9700K, 16 gb ОЗУ. Размер матрицы n x n. Использовался Intel® C++ Compiler 19.0 for Windows* с оптимизацией O2,

5.1 Наивный метод

Время работы наивного метода:

Время, сек.	1 core	2 core	4 core	6 core	8 core
n = 2000	1,2	0,9	0,8	0,8	0,8
n = 2500	2,7	2,3	2,2	2,1	2,1
n = 4000	12	11.5	11	11	11

Таблица 1. Время работы наивного метода.

Масштабируемость наивного метода:

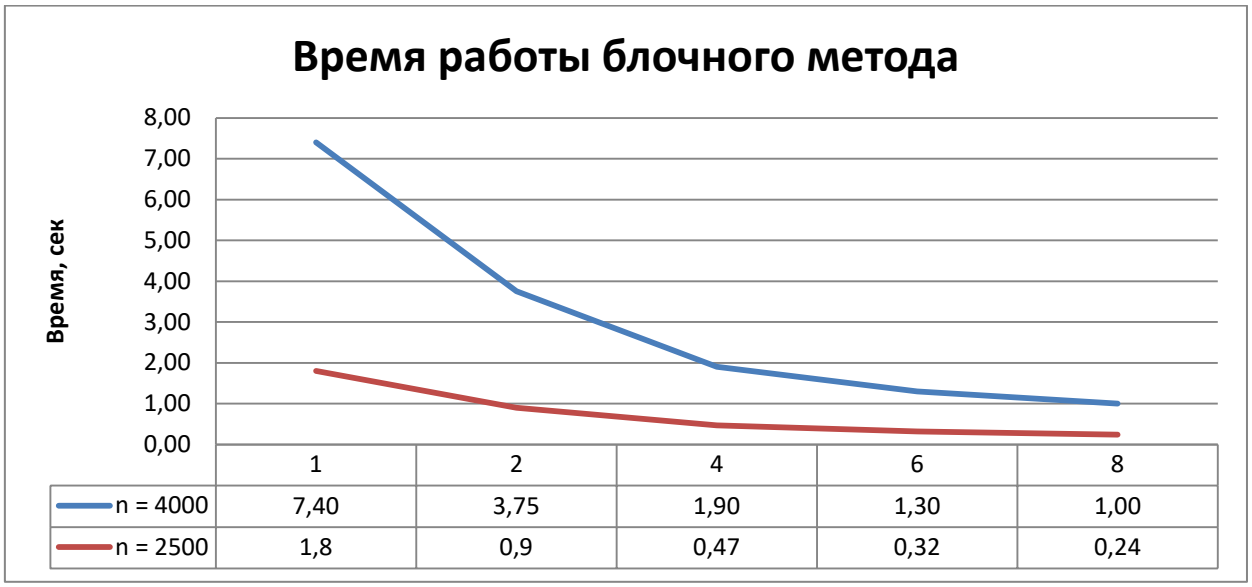
Время, сек.	1 core	2 core	4 core	6 core	8 core
n = 2000	1	0,67	0,38	0,25	0,19
n = 2500	1	0,59	0,31	0,21	0,16
n = 4000	1	0.67	0,38	0,25	0,19

Таблица 2. Масштабируемость наивного метода.

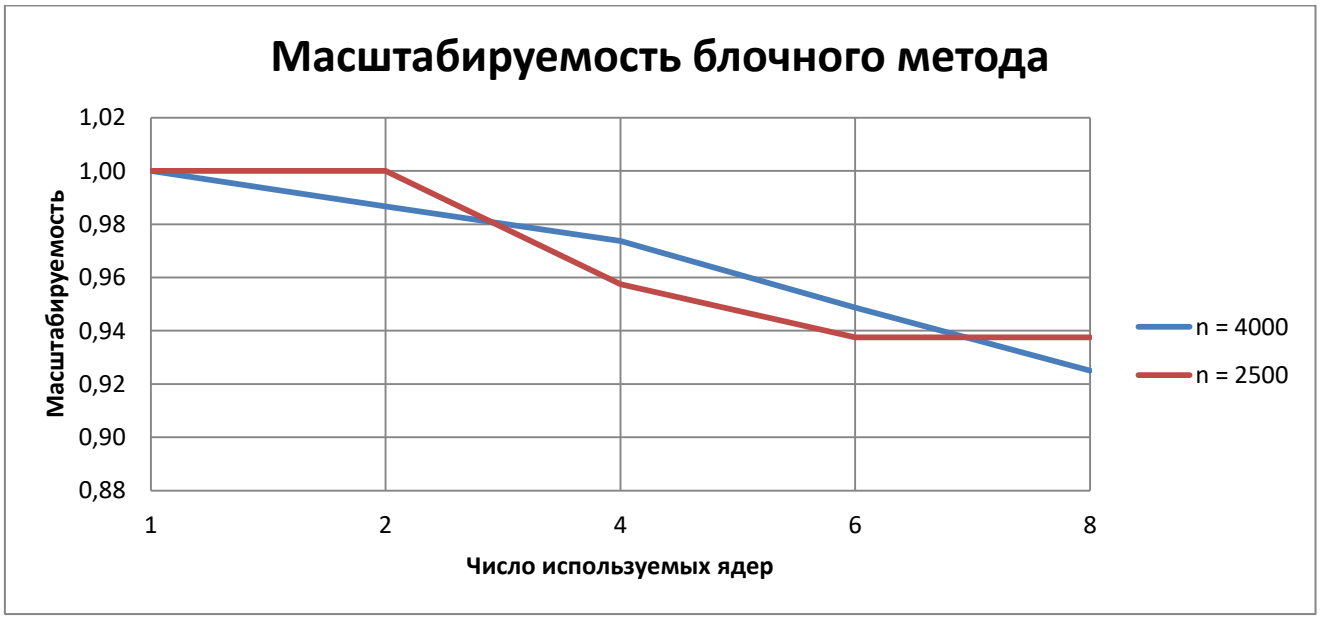
Наивный метод плохо масштабируется, с увеличением n масштабируемость продолжает уменьшаться. Скорее всего это связано с плохой работой с памятью.

5.2 Блочный метод

Рассмотрим работу блочного алгоритма, с размером блока алгоритма (bs) 32 и с размером блока при матричном умножении (mbs) 32.

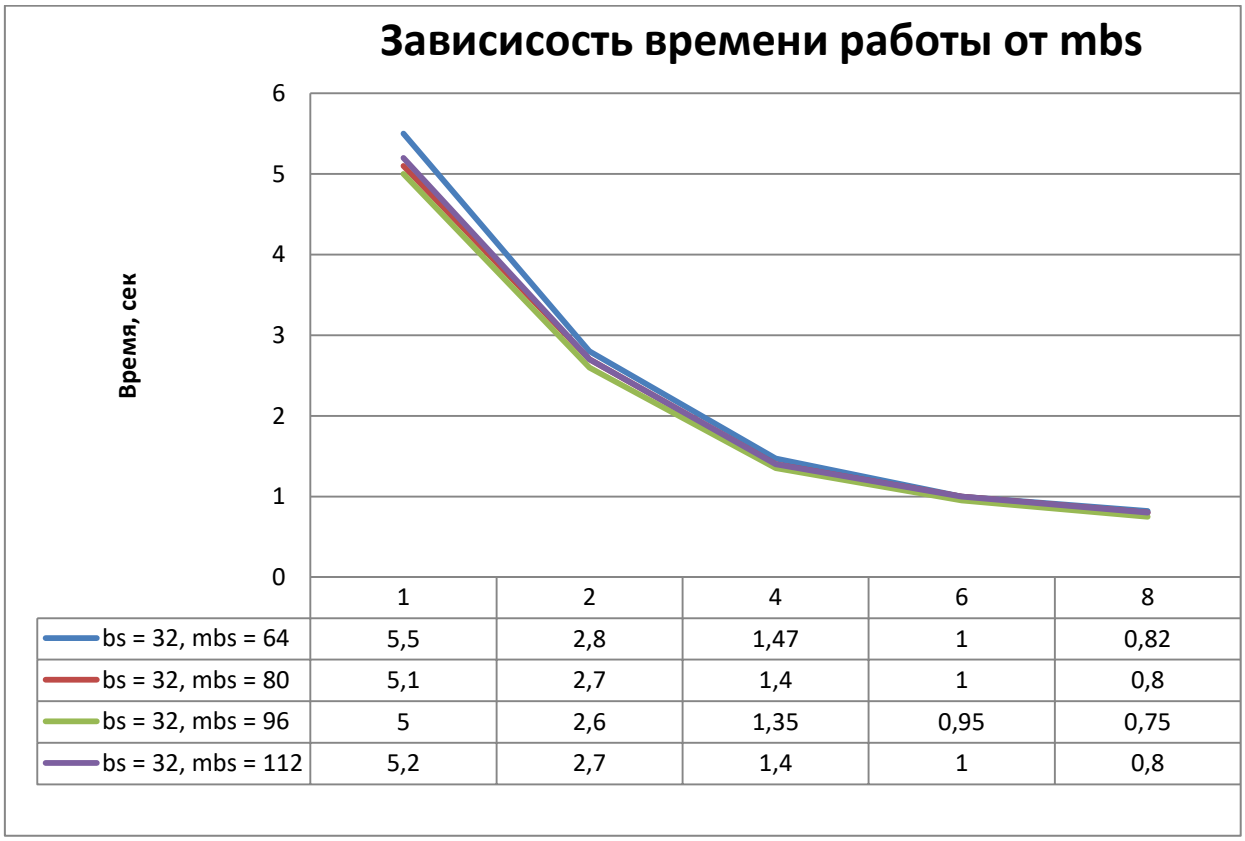


	1	2	4	6	8
n = 4000	7,40	3,75	1,90	1,30	1,00
n = 2500	1,8	0,9	0,47	0,32	0,24



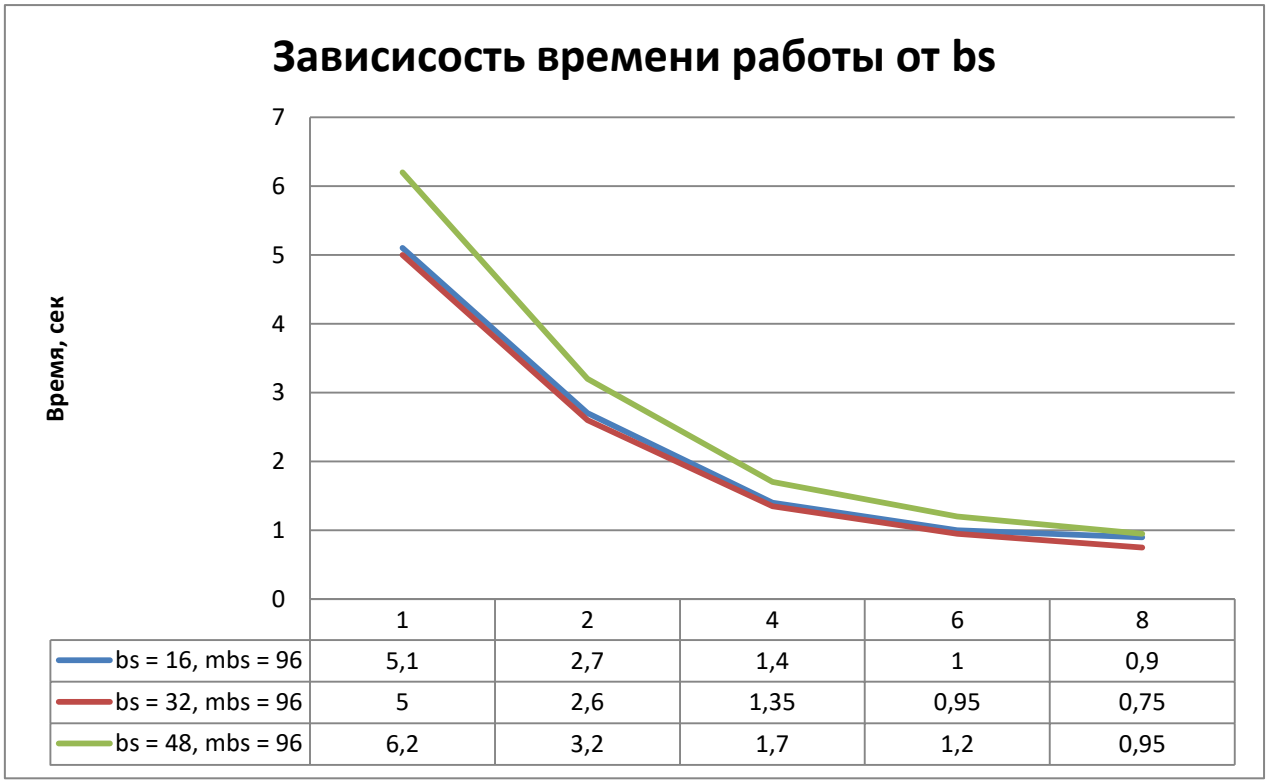
Даже не при самых оптимальных размерах блока масштабируемость на 8 ядрах более 90% (0.9).

На матрице с n = 4000, bs = 32 исследуем зависимость времени работы LU разложения от размера блока в умножении матриц (переменная mbs):



	1	2	4	6	8
bs = 32, mbs = 64	5,5	2,8	1,47	1	0,82
bs = 32, mbs = 80	5,1	2,7	1,4	1	0,8
bs = 32, mbs = 96	5	2,6	1,35	0,95	0,75
bs = 32, mbs = 112	5,2	2,7	1,4	1	0,8

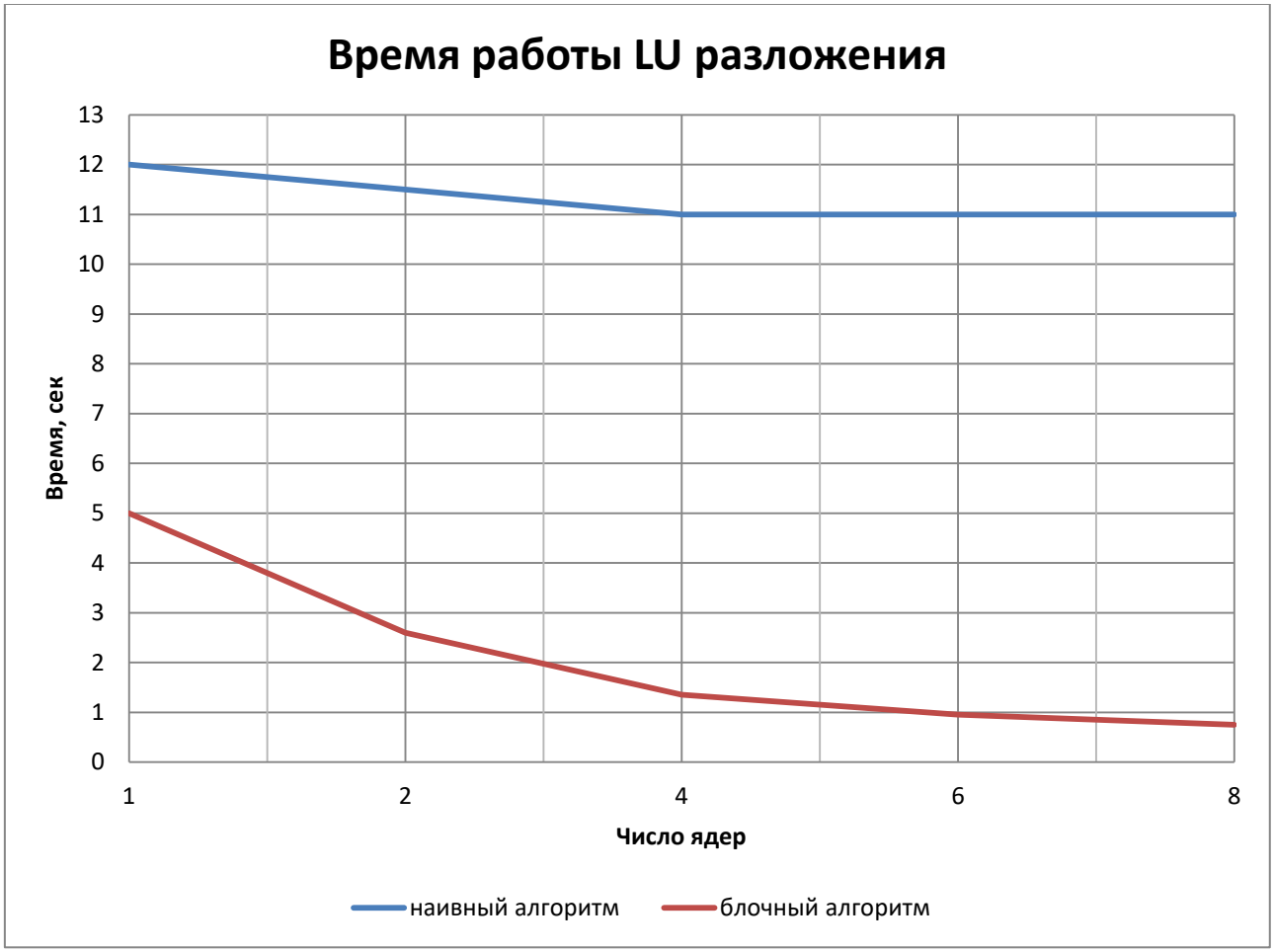
На матрице с n = 4000, bs = 96 исследуем зависимость времени работы LU разложения от размера блока (bs):



	1	2	4	6	8
bs = 16, mbs = 96	5,1	2,7	1,4	1	0,9
bs = 32, mbs = 96	5	2,6	1,35	0,95	0,75
bs = 48, mbs = 96	6,2	3,2	1,7	1,2	0,95

6. Заключение

Численный эксперимент показал, что на 8 ядрах блочная версия имеет гораздо большую масштабируемость (около 0.9) в сравнении с наивной (около 0.2). При этом блочная версия в 11 раз быстрее. Блочная версия с оптимальными размерами блоков быстрее еще на 25% быстрее. Итого ускорение в 14.6 раз!



Сравнение блочного алгоритма с оптимальными размерами блока с наивной версией, на матрице размера 4000x4000.