ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(СПБГУ)

Образовательная программа магистратуры
«Прикладные математика и физика»

Лабораторная работа
ЗАДАНИЕ В-2

Выполнил студент
1 курса магистратуры
(группа 22.М21-фз)
Козлов Александр

Санкт-Петербург
2022

СОДЕРЖАНИЕ

ФОРМУЛИРОВКА ЗАДАНИЯ

Дана СЛАУ:

$$\mathbf{Ax} = \mathbf{b}^{(i)},\ i = 1, 2, \ldots, 10. \tag{1}$$

Матрица $\mathbf{A}$ симметричная и положительно определенная. Оптимизировать программную реализацию решателя этой системы, основанного на разложении Холецкого, используя MPI. Исследовать зависимость масштабируемости параллельной версии программы от ее вычислительной трудоемкости. Сравнить результаты с результатами первой лабораторной работы.

Проверка закона Амдала. Построить зависимость *ускорение:число процессов* для заданного примера.

1 ОПИСАНИЕ ПРОГРАММНО-АППАРАТНОЙ КОНФИГУРАЦИИ ТЕСТОВОГО СТЕНДА

В качестве тестового стенда выступала вычислительная машина, доступ к которой был предоставлен преподавателем. Краткое описание программно-аппаратной конфигурации тестового стенда приведено в Таблице 1.

ОС	Ubuntu 20.04.4 LTS
Число ядер	6
Число потоков	12
Модель процессора	Intel(R) Xeon(R) E-2136 CPU @ 3.30GHz
ОЗУ	62GB
Реализация MPI	mpich
Библиотека MPI	mpi_f08
MPI-компилятор	mpifort

Таблица 1: Программно-аппаратная конфигурация тестового стенда.

2 ОПИСАНИЕ ПОСЛЕДОВАТЕЛЬНОГО АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ

Решение задачи разбивается на два этапа:

1. Применение алгоритма Холецкого к симметричной и положительно определенной матрице $\mathbf{A}$, то есть ее представление в виде $\mathbf{A} = \mathbf{LL}^T$, где матрица $\mathbf{L}$ — нижняя треугольная матрица со строго положительными элементами на диагонали;

2. Последовательное решение двух СЛАУ с треугольными матрицами $\mathbf{LY} = \mathbf{B}$, откуда нужно найти неизвестную матрицу $\mathbf{Y}$, и $\mathbf{L}^T\mathbf{X} = \mathbf{Y}$, где на основе найденной из предыдущей СЛАУ матрицы $\mathbf{Y}$ нужно найти матрицу $\mathbf{X}$.

Рассмотрим подробно каждый этап.

2.1 РАЗЛОЖЕНИЕ ХОЛЕЦКОГО

В данной работе используется блочная версия разложения Холецкого, так как такая версия алгоритма лучше параллелится (см. работу). Ширина блока задается числом n_b, а ширина матрицы $\mathbf{A}$ — числом n. Используемый вариант блочного алгоритма Холецкого может быть описан следующим образом:

1. Матрица $\mathbf{A}$ представляется в виде $\begin{pmatrix} \mathbf{A}_{11} & \star \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}$, где $\mathbf{A}_{11}$ — матрица $b \times b$ (для удобства рассматриваем случай, когда b укладывается целое число раз в n);

2. Вычисляется матрица $\mathbf{L}_{11}$ такая, что $\mathbf{L}_{11}\mathbf{L}_{11}^T = \mathbf{A}_{11}$, то есть матрица $\mathbf{L}_{11}$ — фактор Холецкого матрицы $\mathbf{A}_{11}$, и матрица $\mathbf{A}_{11}$ замещается $\mathbf{L}_{11}$, то есть $\mathbf{A}_{11} := \mathbf{L}_{11}$;

3. Вычисляется матрица $\mathbf{L}_{21} = \mathbf{A}_{21}(\mathbf{L}_{11})^{-1}$, которая замещает матрицу $\mathbf{A}_{21}$, то есть $\mathbf{A}_{21} := \mathbf{L}_{21}$;

4. Обновляется матрица $\mathbf{A}_{22}$ так, что $\mathbf{A}_{22} := \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T$;

5. Алгоритм снова начинается с шага 1 с матрицей $\mathbf{A} = \mathbf{A}_{22}$ (если матрица $\mathbf{A}_{22}$ не пустая).

В Листинге 6 представлена реализация на современном Фортран последовательного блочного алгоритма разложения Холецкого, который был описан выше.

2.2 ОБРАТНАЯ ПОДСТАНОВКА

Вторым этапом решения задачи является обратная подстановка. То есть последовательно решаются 2 СЛАУ с треугольными матрицами

$$\begin{cases} \mathbf{L}\mathbf{y} = \mathbf{b}^{(i)}, \\ \mathbf{L}^T\mathbf{x} = \mathbf{y} \end{cases}$$

для $i = 1, 2, \ldots, 10$. Вектор $\mathbf{y}$ находится по формулам

$$y_1 = \frac{b_1^{(i)}}{l_{11}};\ y_j = \frac{b_j^{(i)} - \sum_{p=1}^{j-1} l_{jp}y_p}{l_{jj}},\ j = 2, 3, \ldots, n,$$

где n — число столбцов матрицы $\mathbf{A}$. Вектор $\mathbf{x}$ находится аналогично

$$x_n = \frac{y_n}{{l^T}_{nn}};\ x_j = \frac{y_j - \sum_{p=j+1}^{n} {l^T}_{jp}x_p}{{l^T}_{jj}},\ j = n-1, n-2, \ldots, 1.$$

Последовательный вариант реализации второго этапа решения задачи представлен в Листинге 7.

3 ОПТИМИЗАЦИЯ АЛГОРИТМА С ИСПОЛЬЗОВАНИЕМ MPI

Исходный код оптимизированного с помощью MPI алгоритма решения задачи (1) с помощью блочного разложения Холецкого представлен в Листинге 8. Рассмотрим каким образом оптимизируются этапы решения задачи (разложение Холецкого и обратная подстановка).

3.1 ОПТИМИЗАЦИЯ РАЗЛОЖЕНИЯ ХОЛЕЦКОГО С ИСПОЛЬЗОВАНИЕМ MPI

При оптимизации блочной версии разложения Холецкого с использованием MPI блоки столбцов матрицы $\mathbf{A}$ $A(i:n,\ i:\min(i+n_b-1,n))$, где $i=1,\ n_b,\ 2n_b,\ \ldots,\ n$, распределяются по процессам так, что блок $A(i:n,\ i:\min(i+n_b-1,n))$ принадлежит процессу под номером $\mathrm{mod}((i-1)/b,\ n_{proc})$, где n_{proc} — число процессов, n_b — ширина блока. Такое распределение происходит в строчках кода, приведенных в Листинге 1, при этом вместо матрицы $\mathbf{A}$ рассматривается матрица $\mathbf{L}$, ненулевые элементы которой задаются как элементы нижней треугольной части матрицы $\mathbf{A}$.

Листинг 1: Инициализация и распределение блоков столбцов матрицы $\mathbf{L}$ по процессам. Ненулевые блоки матрицы $\mathbf{L}$ задаются как блоки нижней треугольной части матрицы $\mathbf{A}$.

```
1  ! Initialization of the matrix L and distribution it between processes
2  allocate(l(n,n))
3  l(:,:) = 0.0
4  do i = rank * nb + 1, n, nb * nproc
5     j = min(i + nb - 1, n)
6     l(i:n, i:j) = a(i:n, i:j)
7  end do
```

Далее в цикле по i последовательно рассматривается блок столбцов $A(i:n,\ i:\min(i+n_b-1,n))$. Сначала вычисляется разложение Холецкого обычным способом для диагонального блока $A(i:\min(i+n_b-1,n),\ i:\min(i+n_b-1,n))$, для чего применяется подпрограмма `cholesky_serial`. Потом происходит преобразование, описанное в Разделе 2.1 в пункте 3. Для этого используется подпрограмма `updating_l_21`, в которой решается СЛАУ на $\mathbf{X}$

$$\mathbf{X}\mathbf{L}_{11}^T = \mathbf{A}_{21}$$

с нижней треугольной матрицей $\mathbf{L}_{11}$. После этого найденная матрица замещает блок `l(j+1:n, i:j)`, где `j = min(i+nb-1, n)`. Блок `l(j+1:n, i:j)` преобразуется в одномерный массив `l_massage` и рассылается широковещательной рассылкой (с помощью `MPI_BCAST`) от процесса, которому принадлежит блок `l(j+1:n, i:j)`, на все остальные процессы. После рассылки одномерный массив `l_massage` преобразуется в блок `l_21` и выполняется преобразование, описанное в Разделе 2.1 в пункте 4. Так как каждый процесс владеет своей частью обновляемого в пункте 4 массива, это позволяет провести вычисления в пункте 4 параллельно. Данным вычислениям соответствует код, представленный в Листинге 2.

Листинг 2: Параллельная реализация преобразования, описанного в Разделе 2.1 в пункте 4.

```
! Updating the other blocks of the L matrix
do k = i + nb, n, nb
   p = min(k + nb - 1, n)
   ! Updating the (k:n, k:p) block
   if (mod( (k - 1) / nb, nproc ) .eq. rank) then
      l(k:n, k:p) = l(k:n, k:p) - matmul(l_21(k:n, i:j), &
         transpose(l_21(k:p, i:j)))
   end if
end do
```

3.2 ОПТИМИЗАЦИЯ ОБРАТНОЙ ПОДСТАНОВКИ С ИСПОЛЬЗОВАНИЕМ MPI

При обратной подстановке оказывается неудобным тот факт, что матрица **L** не хранится в каждом процессе целиком, поэтому сперва матрица **L** распространяется на все процессы. Это делается в части кода, представленной в Листинге 3, где блок столбцов, принадлежащий какому-то одному процессу, сохраняется в виде одномерного массива `l_massage`, потом делается широковещательная рассылка этого массива и обратное преобразование из `l_massage` в блок столбцов, который теперь хранится в каждом процессе.

Листинг 3: Распространение фактора Холецкого **L** на все процессы.

```
   ! Broadcasting the matrix L
   do i = 1, n, nb
      j = min(i + nb - 1, n)
      allocate(l_massage(n * (j - i + 1)))
      l_massage(:) = 0.0
      if (mod( (i-1)/nb, nproc ) .eq. rank) then
         ! Filling l_massage with the (1:n, i:j) block
         do s = 1, n
            do t = 1, j-i+1
               l_massage((s-1)*(j-i+1) + t) = l(s, i-1+t)
            end do
         end do
      end if
      call MPI_BARRIER(MPI_COMM_WORLD, ierr)
      ! Broadcasting l_massage to other processes
      call MPI_BCAST(l_massage, n * (j-i+2), MPI_REAL, &
         mod( (i-1)/nb, nproc ), MPI_COMM_WORLD, ierr)
      if (rank .ne. mod( (i-1)/nb, nproc )) then
         ! Filling the block (1:n, i:j) with l_massage
         do s = 1, n
```

```
            do t = 1, j-i+1
               l(s, i-1+t) = l_massage((s-1)*(j-i+1) + t)
            end do
         end do
      end if
      deallocate(l_massage)
   end do
```

После распространения матрицы **L** на все процессы происходит сама обратная подстановка. Она реализуется параллельно, при этом процесс с рангом `rank` находит столбцы `y(:,i)` и `x(:,i)`, где `i=rank+1`, `rank+1+nproc`, `rank+1+2*nproc` и так далее до `m=10`. Исходный код этапа обратной подстановки представлен в Листинге 4.

Листинг 4: Исходный код этапа обратной подстановки в оптимизированной с помощью MPI версии алгоритма.

```
   ! Backward substitution
   do i = rank + 1, m, nproc
      ! Finding y(:,i)
      y(1,i) = b(1,i) / l(1,1)
      do j = 2, n
         y(j,i) = b(j,i)
         do k = 1, j - 1
            y(j,i) = y(j,i) - l(j,k) * y(k,i)
         end do
         y(j,i) = y(j,i) / l(j,j)
      end do
      ! Finding x(:,i)
      x(n,i) = y(n,i) / l(n,n)
      do j = n-1, 1, -1
         x(j,i) = y(j,i)
         do k = j+1, n
            x(j,i) = x(j,i) - l(k,j) * x(k,i)
         end do
         x(j,i) = x(j,i) / l(j,j)
      end do
   end do
```

3.3 ИНИЦИАЛИЗАЦИЯ МАТРИЦЫ A

Важно отметить, что матрица **A** всегда инициализируется следующим образом

$$\mathbf{A} = \mathbf{B} + n\mathbf{I}, \tag{2}$$

где $\mathbf{B}$ — $n \times n$ матрица, каждый элемент которой равен 1, $\mathbf{I}$ — единичная $n \times n$ матрица.

4 ВЕРИФИКАЦИЯ ОПТИМИЗИРОВАННОЙ ВЕРСИИ АЛГОРИТМА

Рассмотрим СЛАУ с $\mathbf{b} = (1, 1, 1, 1, 1)^{\mathrm{T}}$ и матрицей

$$\mathbf{A} = \begin{pmatrix} 6 & 1 & 1 & 1 & 1 \\ 1 & 6 & 1 & 1 & 1 \\ 1 & 1 & 6 & 1 & 1 \\ 1 & 1 & 1 & 6 & 1 \\ 1 & 1 & 1 & 1 & 6 \end{pmatrix}.$$

На Рис. 1 показан результат работы программы при числе процессов, равном 1 и 4. Программа выводит на экран транспонированный вектор решения $\mathbf{x}^T$. Видно, что как

```
[mrk@192 src]$ mpifort -O2 cholesky_mpi_block_test.f90 -o cholesky_mpi_block_test
[mrk@192 src]$ mpirun -np 1 ./cholesky_mpi_block_test
5 1
           1  0.10000000000000005        9.9999999999999978E-002  0.10000000000000001        9.9999999999999978E-002   9.9999999999999978E-002
           5  9.6005000045806810E-005
[mrk@192 src]$ mpirun -np 4 ./cholesky_mpi_block_test
5 1
           1  0.10000000000000005        9.9999999999999978E-002  0.10000000000000001        9.9999999999999978E-002   9.9999999999999978E-002
           5  2.6547000004484289E-004
```

Рис. 1: Результат работы алгоритма при одном потоке и при четырех потоках, на экран в первой строке выводится номер строки (в данном случае всегда 1) и сам транспонированный вектор решения $\mathbf{x}^T$, а на второй — размер матрицы n и время работы.

при одном процессе, так и при двух процессах программа выдает правильный ответ $\mathbf{x}^T = (0.1, 0.1, 0.1, 0.1, 0.1)^T$ с машинной точностью.

5 ИЗМЕРЕНИЕ УСКОРЕНИЯ

5.1 ИЗМЕРЕНИЕ ВРЕМЕНИ РАБОТЫ АЛГОРИТМА

Для проверки закона Амдала необходимо определить ускорение алгоритма при различном числе процессов. Ускорение определяется на основе времени работы алгоритма. Время работы алгоритма оказывается зависящим от ширины блока n_b, выбор наиболее эффективного размера блока n_b зависит от характеристик тестового стенда и составляет для рассматриваемой машины $n_b = 10$.

Время работы оптимизированной программы при различном числе потоков и различной вычислительной сложности (для $n = 100, 1100, 2500, 5000, 10000$) мерилось с помощью bash-сценария, который представлен в Листинге 5. Для каждого n измерения проводились по три раза. Стоит отметить, что внутри самой программы время мерилось с помощью функции `MPI_WTIME`.

Листинг 5: Сценарий запуска численных экспериментов на языке bash.

```
1 #!/bin/bash
2 # compiling the program
3 mpifort -O2 cholesky_mpi_block_test.f90 -o cholesky_mpi_block_test
4 # Entering the maximum number of processes, the amount of experiments and the
```

```
echo "Enter the maximum number of processes"
read max_nproc
echo "Enter the amount of experiments"
read nexp
echo "Enter the block width"
read nb
# Running the program in the following loop
for (( exp=1; exp<=$nexp; exp++ ))
do
   echo "Experiment: $exp"
   for (( nproc=1; nproc<=$max_nproc; nproc++ ))
   do
      echo "Number of processes: $nproc"
      filename=../data/$exp.$nproc.txt
      echo "# n; Execution time, sec" > $filename
      ns=(100 1100 2500 5000 10000)
      for n in ${ns[@]}
      do
         echo "n: $n"
         # keeping time data
         echo "$n $nb" | mpirun -np $nproc ./cholesky_mpi_block_test
         >> $filename
      done
   done
done
```

Результаты измерений приведены в Таблице 2. На Рис. 2 показана зависимость среднего времени выполнения оптимизированной версии программы от числа потоков при различной вычислительной сложности, вертикальными линиями отмечены плюс и минус одно стандартное отклонение (корень из дисперсии).

5.2 ИЗМЕРЕНИЕ УСКОРЕНИЯ АЛГОРИТМА

Чтобы определить ускорение, для каждого $n = 100, 1100, 2500, 5000, 10000$ вычисляется среднее время работы алгоритма в последовательном режиме $t_1^{(n)}$ с абсолютной погрешностью $\Delta t(n)_1$, которая равна стандартному отклонению времени работы алгоритма в последовательном режиме. Далее считается само ускорение алгоритма с матрицей размера $n \times n$ для T потоков по формуле

$$k_T^{(n)} = \frac{t_1^{(n)}}{t_T^{(n)}}, \tag{3}$$

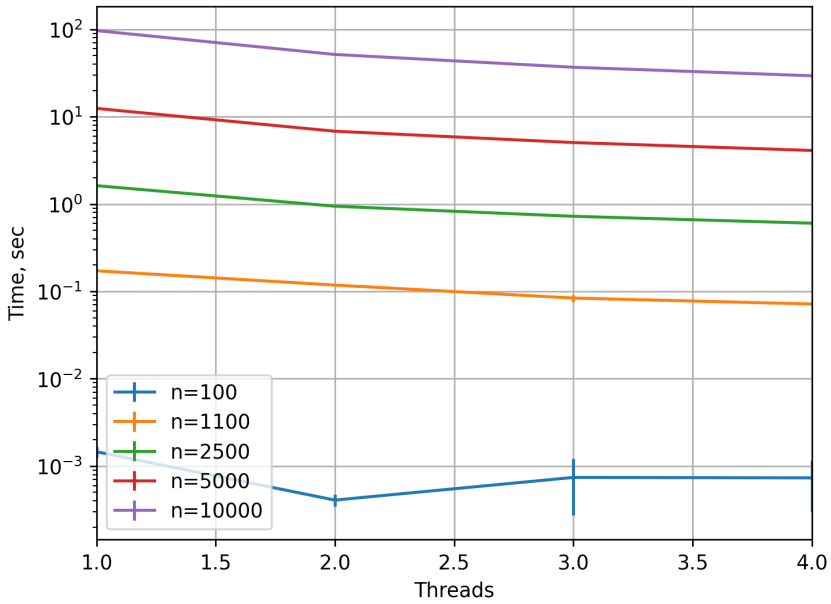

Рис. 2: Время выполнения оптимизированной программы в секундах при различном числе потоков. Показано среднее за 3 эксперимента время. Вертикальными линиями отмечены плюс и минус одно стандартное отклонение (корень из дисперсии; они не всегда различимы из-за своей малости).

где $t_T^{(n)}$ — среднее время работы алгоритма с матрицей размера $n \times n$ на T потоках. Погрешность ускорения рассчитывается следующим образом:

$$\Delta k_T^{(n)} = \frac{\Delta t_1^{(n)} \cdot t_T^{(n)} + \Delta t_T^{(n)} \cdot t_1^{(n)}}{(t_T^{(n)})^2}. \tag{4}$$

На Рис. 3 показана зависимость ускорения от числа потоков при различной вычислительной сложности, закрашенные области соответствуют плюс и минус одному стандартному отклонению.

6 ИНТЕРПРЕТАЦИЯ РЕЗУЛЬТАТОВ, ВЫВОДЫ

При большой вычислительной сложности задачи (при $n = 1100, 2500, 5000, 10000$) ускорение растет при увеличении числа процессов, при этом, чем больше вычислительная сложность, тем быстрее растет ускорение, что видно из Рис. 3. Закон Амдала при большой вычислительной сложности выполняется, при этом можно предположить, что доля последовательного кода уменьшается с увеличением вычислительной сложности задачи.

При малой вычислительной сложности (при $n = 100$) ускорение сильно увеличивается в случае двух потоков, при этом преодолевается теоретический предел ускорения (ускорение равно числу потоков), что видно из Рис. 3. Таким образом, при малой вычислительной сложности закон Амдала нарушается. При дальнейшем увеличении числа потоков ускорение уменьшается, что можно связать с тем, что при малой сложности задачи на обмен сообщениями между процессами тратится число операций, сравнимое с числом арифметических операций в задаче.

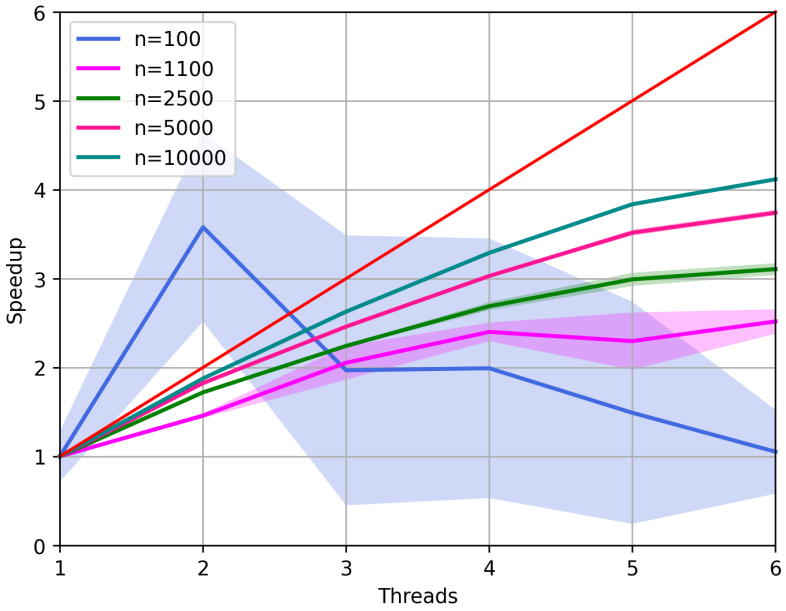

Рис. 3: Ускорение оптимизированной программы при различном числе потоков, закрашенные области соответствуют плюс и минус одному стандартному отклонению (для некоторых линий закрашенных областей может быть не видно, это свидетельствует лишь о малости ошибки). Красная линия обозначает идеальный случай, когда ускорение равно числу потоков.

В сравнении с оптимизацией с использованием OpenMP оптимизация с использованием MPI хуже ускоряет алгоритм при большой вычислительной сложности задачи (при $n = 1100, 2500$; в прошлой лабораторной работе рассматривались лишь $n = 100, 1100, 2500$, поэтому здесь рассматриваются только такие n). Так, если на 6 потоках при $n = 2500$ оптимизация с помощью OpenMP позволяет достичь ускорения, примерно равного 5, то оптимизация с использованием MPI позволяет достичь ускорения, примерного равного 3. При малой вычислительной сложности проводить сравнение двух оптимизаций некорректно, так как в прошлой лабораторной работе при $n = 100$ ускорение было измерено с большой погрешностью, что не позволяет сделать какие-либо выводы о том, насколько хорошо оптимизирован алгоритм.

ПРИЛОЖЕНИЕ

Листинг 6: Реализация последовательного блочного алгоритма разложения Холецкого на современном Фортране.

```
! Block version of the Cholesky decomposition
do i = 1, n, nb
   j = min(i + nb - 1, n)
   ! Updating the (i:n, i:j) block of the L matrix
   if (mod( (i-1)/nb, nproc ) .eq. rank) then
      ! Updating the (i:j, i:j) block of the L matrix
      call cholesky_serial(j - i + 1, l(i:j, i:j))
   end if
   if (j .lt. n) then
      if (mod( (i-1)/nb, nproc ) .eq. rank) then
         ! Updating the (j+1:n, i:j) block of the matrix L
         call updating_l_21(n - j, j - i + 1, l(j + 1:n, i:j), &
            l(i:j, i:j))
      end if
      ! Allocation the useful array
      allocate(l_21(j+1:n, i:j))
      l_21 = l(j + 1:n, i:j)
      ! Updating the other blocks of the L matrix
      do k = i + nb, n, nb
         p = min(k + nb - 1, n)
         ! Updating the (k:n, k:p) block
         l(k:n, k:p) = l(k:n, k:p) - matmul(l_21(k:n, i:j), &
            transpose(l_21(k:p, i:j)))
      end do
      deallocate(l_21)
   end if
end do

subroutine cholesky_serial(n, l)
   implicit none
   integer, intent(in) :: n
   real*8 :: l(n,n)
   real*8 :: s
   integer :: i, ip, j
   ! Serial Cholesky decomposition
   do  i = 1, n
      s = l(i,i)
      do  ip = 1, i - 1
         s = s - l(i,ip) * l(i,ip)
```

```
      end do
      l(i,i) = sqrt(s)
      do  j = i + 1, n
         s = l(j,i)
         do  ip = 1, i-1
            s = s - l(i,ip) * l(j,ip)
         end do
         l(j,i) = s / l(i,i)
      end do
   end do
   do i = 1, n
      l(i,i+1:n) = 0.0
   end do
end subroutine cholesky_serial

subroutine updating_l_21(nrows, ncols, l_21_, l_11_)
   integer, intent(in) :: nrows, ncols
   integer :: i_, j_, k_
   real*8 :: l_21_(nrows, ncols), x(nrows, ncols)
   real*8, intent(in) :: l_11_(ncols, ncols)
   do i_ = 1, nrows
      x(i_, 1) = l_21_(i_, 1) / l_11_(1, 1)
      do j_ = 2, ncols
         x(i_, j_) = l_21_(i_, j_)
         do k_ = 1, j_ - 1
            x(i_, j_) = x(i_, j_) - x(i_, k_) * l_11_(j_, k_)
         end do
         x(i_, j_) = x(i_, j_) / l_11_(j_, j_)
      end do
   end do
   l_21_ = x
end subroutine
```

Листинг 7: Последовательная реализация обратной подстановки на современном Фортране.

```
! Backward substitution
   do i = 1, m
      ! Finding y(:,i)
      y(1,i) = b(1,i) / l(1,1)
      do j = 2, n
         y(j,i) = b(j,i)
         do k = 1, j - 1
            y(j,i) = y(j,i) - l(j,k) * y(k,i)
         end do
         y(j,i) = y(j,i) / l(j,j)
      end do
      ! Finding x(:,i)
      x(n,i) = y(n,i) / l(n,n)
      do j = n-1, 1, -1
         x(j,i) = y(j,i)
         do k = j+1, n
            x(j,i) = x(j,i) - l(k,j) * x(k,i)
         end do
         x(j,i) = x(j,i) / l(j,j)
      end do
   end do
```

Листинг 8: Исходный код оптимизированного с помощью MPI алгоритма решения задачи (1) с помощью блочного разложения Холецкого на современном Фортране.

```
program cholesky_mpi_block
   use mpi_f08
   implicit none
   integer, parameter :: m = 10
   integer :: i, j, k, n, nb, t, s, p
   real*8, allocatable :: id(:,:), a(:,:), l(:,:), l_21(:,:), &
      l_massage(:)
   real*8, allocatable :: b(:,:), x(:,:), y(:,:)
   real*8 :: time  ! time counter
   integer :: nproc, rank, ierr
   type(MPI_Status) :: status
   ! Interface for subroutines
   interface
      subroutine cholesky_serial(n, l)
         implicit none
         integer, intent(in) :: n
         real*8 :: l(n,n)
         real*8 :: s
         integer :: i, ip, j
      end subroutine cholesky_serial

      subroutine updating_l_21(nrows, ncols, l_21_, l_11_)
         integer, intent(in) :: nrows, ncols
         integer :: i_, j_, k_
         real*8 :: l_21_(nrows, ncols), x(nrows, ncols)
         real*8, intent(in) :: l_11_(ncols, ncols)
      end subroutine
   end interface
   ! Initialization of the MPI environment
   call MPI_INIT(ierr)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   ! Initialization of the time counter
   ! Reading n and calculation nb
   if (rank .eq. 0) then
      read(*, *) n, nb
      ! nb = max(n / nproc, 1)
   end if
   if (rank .eq. 0) then
      time = MPI_WTIME()
   end if
```

```
42    ! Broadcasting n and nb to other processes
43    call MPI_BARRIER(MPI_COMM_WORLD, ierr)
44    call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
45    call MPI_BCAST(nb, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
46    ! Id matrix
47    allocate(id(n, n))
48    ForAll(i = 1:n, j = 1:n) id(i,j) = (i/j)*(j/i)
49    ! Initialization of the matrix A
50    allocate(a(n, n))
51    a(:, :) = 1.0
52    a = (a + transpose(a)) / 2 + n * id
53    ! Initialization of the matrix L and distribution it between processes
54    allocate(l(n,n))
55    l(:,:) = 0.0
56    do i = rank * nb + 1, n, nb * nproc
57       j = min(i + nb - 1, n)
58       l(i:n, i:j) = a(i:n, i:j)
59    end do
60    ! Block version of the Cholesky decomposition
61    do i = 1, n, nb
62       j = min(i + nb - 1, n)
63       ! Updating the (i:n, i:j) block of the L matrix
64       if (mod( (i-1)/nb, nproc ) .eq. rank) then
65          ! Updating the (i:j, i:j) block of the L matrix
66          call cholesky_serial(j - i + 1, l(i:j, i:j))
67       end if
68       if (j .lt. n) then
69          allocate(l_massage((n-j)*(j-i+1)))
70          l_massage(:) = 0.0
71          if (mod( (i-1)/nb, nproc ) .eq. rank) then
72             ! Updating the (j+1:n, i:j) block of the matrix L
73             call updating_l_21(n - j, j - i + 1, l(j + 1:n, i:j), &
74                l(i:j, i:j))
75             ! Filling l_massage
76             do s = 1, n-j
77                do t = 1, j-i+1
78                   l_massage((s-1)*(j-i+1) + t) = l(j+s, i-1+t)
79                end do
80             end do
81          end if
82          call MPI_BARRIER(MPI_COMM_WORLD, ierr)
83          ! Broadcasting the block l_massage to other processes
84          call MPI_BCAST(l_massage, (n-j)*(j-i+2), MPI_REAL, &
```

```
85              mod( (i-1)/nb, nproc ), MPI_COMM_WORLD, ierr)
86          ! Allocation the useful array
87          allocate(l_21(j+1:n, i:j))
88          do s = 1, n-j
89              do t = 1, j-i+1
90                  l_21(j+s, i-1+t) = l_massage((s-1)*(j-i+1) + t)
91              end do
92          end do
93          ! Updating the other blocks of the L matrix
94          do k = i + nb, n, nb
95              p = min(k + nb - 1, n)
96              ! Updating the (k:n, k:p) block
97              if (mod( (k - 1) / nb, nproc ) .eq. rank) then
98                  l(k:n, k:p) = l(k:n, k:p) - matmul(l_21(k:n, i:j), &
99                      transpose(l_21(k:p, i:j)))
100             end if
101         end do
102         deallocate(l_21)
103         deallocate(l_massage)
104     end if
105 end do
106 ! Showing the transposed matrix L
107 ! do i = rank * nb + 1, n, nb * nproc
108 !     j = min(i + nb - 1, n)
109 !     do k = i, j
110 !         print *, k, l(:, k)
111 !     end do
112 ! end do
113 ! Broadcasting the matrix L
114 do i = 1, n, nb
115     j = min(i + nb - 1, n)
116     allocate(l_massage(n * (j - i + 1)))
117     l_massage(:) = 0.0
118     if (mod( (i-1)/nb, nproc ) .eq. rank) then
119         ! Filling l_massage with the (1:n, i:j) block
120         do s = 1, n
121             do t = 1, j-i+1
122                 l_massage((s-1)*(j-i+1) + t) = l(s, i-1+t)
123             end do
124         end do
125     end if
126     call MPI_BARRIER(MPI_COMM_WORLD, ierr)
127     ! Broadcasting l_massage to other processes
```

```
128       call MPI_BCAST(l_massage, n * (j-i+2), MPI_REAL, &
129          mod( (i-1)/nb, nproc ), MPI_COMM_WORLD, ierr)
130       if (rank .ne. mod( (i-1)/nb, nproc )) then
131          ! Filling the block (1:n, i:j) with l_massage
132          do s = 1, n
133             do t = 1, j-i+1
134                l(s, i-1+t) = l_massage((s-1)*(j-i+1) + t)
135             end do
136          end do
137       end if
138       deallocate(l_massage)
139    end do
140    ! Initialization of matrices B, X and Y
141    allocate(b(n,m), x(n,m), y(n,m))
142    do i = 1, m
143       b(:,i) = i
144    end do
145    ! Backward substitution
146    do i = rank + 1, m, nproc
147       ! Finding y(:,i)
148       y(1,i) = b(1,i) / l(1,1)
149       do j = 2, n
150          y(j,i) = b(j,i)
151          do k = 1, j - 1
152             y(j,i) = y(j,i) - l(j,k) * y(k,i)
153          end do
154          y(j,i) = y(j,i) / l(j,j)
155       end do
156       ! Finding x(:,i)
157       x(n,i) = y(n,i) / l(n,n)
158       do j = n-1, 1, -1
159          x(j,i) = y(j,i)
160          do k = j+1, n
161             x(j,i) = x(j,i) - l(k,j) * x(k,i)
162          end do
163          x(j,i) = x(j,i) / l(j,j)
164       end do
165    end do
166    ! Showing the transposed matrix X
167    ! do i = 1, m
168    !    if (mod(i-1, nproc) .eq. rank) then
169    !       print *, i, x(:,i)
170    !    end if
```

```
171     ! end do
172     ! Time measurement
173     call MPI_BARRIER(MPI_COMM_WORLD, ierr)
174     if (rank .eq. 0) then
175        time = MPI_WTIME() - time
176        print *, n, time
177     end if
178     deallocate(id, a, l)
179     deallocate(b, x, y)
180     call MPI_FINALIZE(ierr)
181 end program cholesky_mpi_block
182 
183 subroutine cholesky_serial(n, l)
184     implicit none
185     integer, intent(in) :: n
186     real*8 :: l(n,n)
187     real*8 :: s
188     integer :: i, ip, j
189     ! Serial Cholesky decomposition
190     do  i = 1, n
191        s = l(i,i)
192        do  ip = 1, i - 1
193           s = s - l(i,ip) * l(i,ip)
194        end do
195        l(i,i) = sqrt(s)
196        do  j = i + 1, n
197           s = l(j,i)
198           do  ip = 1, i-1
199              s = s - l(i,ip) * l(j,ip)
200           end do
201           l(j,i) = s / l(i,i)
202        end do
203     end do
204     do i = 1, n
205        l(i,i+1:n) = 0.0
206     end do
207 end subroutine cholesky_serial
208 
209 subroutine updating_l_21(nrows, ncols, l_21_, l_11_)
210     integer, intent(in) :: nrows, ncols
211     integer :: i_, j_, k_
212     real*8 :: l_21_(nrows, ncols), x(nrows, ncols)
213     real*8, intent(in) :: l_11_(ncols, ncols)
```

```
214    do i_ = 1, nrows
215       x(i_, 1) = l_21_(i_, 1) / l_11_(1, 1)
216       do j_ = 2, ncols
217          x(i_, j_) = l_21_(i_, j_)
218          do k_ = 1, j_ - 1
219             x(i_, j_) = x(i_, j_) - x(i_, k_) * l_11_(j_, k_)
220          end do
221          x(i_, j_) = x(i_, j_) / l_11_(j_, j_)
222       end do
223    end do
224    l_21_ = x
225 end subroutine
```

Threads \n	100			1100			2500		
	Exp#1	Exp#2	Exp#3	Exp#1	Exp#2	Exp#3	Exp#1	Exp#2	Exp#3
1	1.75e-03	1.32e-03	1.31e-03	1.71e-01	1.73e-01	1.72e-01	1.62e+00	1.62e+00	1.64e+00
2	3.66e-04	3.61e-04	4.98e-04	1.20e-01	1.17e-01	1.17e-01	9.49e-01	9.37e-01	9.46e-01
3	4.21e-04	1.40e-03	4.03e-04	7.83e-02	9.42e-02	7.92e-02	7.23e-01	7.24e-01	7.26e-01
4	1.34e-03	3.75e-04	4.82e-04	6.94e-02	7.56e-02	7.03e-02	6.13e-01	6.01e-01	5.97e-01
5	4.19e-04	1.94e-03	5.81e-04	6.64e-02	8.92e-02	6.95e-02	5.58e-01	5.36e-01	5.35e-01
6	1.04e-03	1.99e-03	1.14e-03	7.32e-02	6.60e-02	6.60e-02	5.25e-01	5.31e-01	5.11e-01

Threads \n	5000			10000		
	Exp#1	Exp#2	Exp#3	Exp#1	Exp#2	Exp#3
1	1.24e+01	1.24e+01	1.24e+01	9.67e+01	9.69e+01	9.66e+01
2	6.82e+00	6.81e+00	6.82e+00	5.15e+01	5.15e+01	5.17e+01
3	5.05e+00	5.04e+00	5.08e+00	3.68e+01	3.68e+01	3.68e+01
4	4.10e+00	4.10e+00	4.12e+00	2.94e+01	2.94e+01	2.94e+01
5	3.50e+00	3.58e+00	3.53e+00	2.52e+01	2.53e+01	2.53e+01
6	3.36e+00	3.32e+00	3.30e+00	2.35e+01	2.35e+01	2.35e+01

Таблица 2: Время работы программы в секундах при различном числе потоков (Threads) и числе столбцов матрицы $\mathbf{A}$ n. Приведены результаты трёх численных экспериментов для каждого n.