Санкт-Петербургский государственный университет
Физический факультет

ЗАДАНИЕ В

Работу выполнил студент
Козлов Александр

Санкт-Петербург, 19 мая 2023 г.

Содержание

Формулировка задания

СЛАУ:

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

Матрица $\mathbf{A}$ симметричная и положительно определенная. Оптимизировать программную реализацию решателя этой системы, основанного на разложении Холецкого, используя OpenMP. Исследовать зависимость масштабируемости параллельной версии программы от ее вычислительной трудоемкости.

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

1 Описание программно-аппаратной конфигурации тестового стенда

В качестве тестового стенда выступала вычислительная машина, доступ к которому 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
Компилятор	gfortran

Таблица 1: Программно-аппаратная конфигурация тестового стенда.

2 Описание метода решения задачи

2.1 Описание последовательного алгоритма решения задачи

Первым этапом решения задачи является применение разложения Холецкого к симметричной и положительно определенной матрице $\mathbf{A}$, то есть ее представление в виде $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, где матрица $\mathbf{L}$ — нижняя треугольная матрица со строго положительными элементами на диагонали. Реализация последовательного алгоритма разложения Холецкого на Фортране в простейшем (без перестановок суммирования) варианте была взята с сайта проекта AlgoWiki. Данная реализация представлена в Листинге 1. Переменная s имеет двойную точность, в то время как остальные — одинарную.

Листинг 1: Реализация последовательного алгоритма разложения Холецкого на Фортране в простейшем (без перестановок суммирования) варианте.

```
do  i = 1, n
   s = l(i,i)
```

```
   do  ip = 1, i - 1
      s = s - dprod(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 - dprod(l(i,ip), l(j,ip))
      end do
      l(j,i) = s / l(i,i)
   end do
end do
```

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 2.

Листинг 2: Последовательная реализация обратной подстановки на Фортране.

```
do i = 1, m
    y(1) = b(1,i) / l(1,1)
    do j = 2, n
        s = b(j,i)
        do ip = 1, j - 1
            s = s - dprod(l(j,ip), y(ip))
        end do
        y(j) = s / l(j,j)
    end do
    x(n,i) = y(n) / l_t(n,n)
    do j = n - 1, 1, -1
    s = y(j)
        do ip = j + 1, n
            s = s - dprod(l_t(j,ip), x(ip,i))
        end do
```

```
        x(j,i) = s / l_t(j,j)
    end do
end do
```

2.2 Описание оптимизации с использованием OpenMP

Исходный код, оптимизированного с помощью OpenMP решателя системы (1), основанного на разложении Холецкого, на Фортране приведен в Листинге 6. Сперва был распараллелен код, отвечающий за декомпозицию Холецкого. Из Листинга 1 видно, что цикл по j не содержит цикловых зависимостей, поэтому он был распараллелен с помощью директивы следующей директивы OpenMP:

Листинг 3: Директива OpenMP, которая была использована для распараллеливания цикла `do` при реализации разложения Холецкого.

```
!$omp parallel do private(j,ip,s) shared(l,i) schedule(static)
```

Директива `!$omp parallel do` помечает цикл `do`, следующий сразу после данной директивы, чтобы сделать его распараллеливаемым. Число итераций цикла зависит от j и составляет $n-(j+1)$, что означает, что данный цикл может одновременно выполняться не более чем $n-(j+1)$ потоками. Оператор `private(j,ip,s)` объявляет переменные j, ip и s локальными, то есть на каждом потоке создается локальная копия данных переменных. Оператор `shared(l,i)` объявляет переменные l и i общими для потоков. Оператор `schedule(static)` задает такой способ распределения итераций цикла между потоками, что количество итераций цикла, передаваемых для выполнения каждому потоку, фиксировано (в данном случае равно 1) и распределяется между нитями по принципу кругового планирования.

Далее была распараллелена часть решателя, отвечающая обратной подстановке. Цикл `do` по i, как можно заметить из Листинга 2, не содержит цикловых зависимостей, поэтому он и был выбран для распаралелливания. Для этого перед началом данного цикла была помещена директива:

Листинг 4: Директива OpenMP, которая была использована для распараллеливания цикла `do` при реализации обратной подстановки.

```
!$omp parallel do private(i,j,y,s) shared(x,l,l_t) schedule(static)
```

Директива `!$omp parallel do` помечает цикл `do`, следующий сразу после данной директивы, чтобы сделать его распараллеливаемым. Число итераций цикла равно 10. Оператор `private(i,j,y,s)` объявляет переменные i, j, y и s локальными, то есть на каждом потоке создается локальная копия данных переменных. Оператор `shared(x,l,l_t)` объявляет переменные x, l и l_t общими для потоков. Оператор `schedule(static)` задает такой способ распределения итераций цикла между потоками, что количество итераций цикла, передаваемых для выполнения каждому потоку, фиксировано (в данном случае равно 1) и распределяется между нитями по принципу кругового планирования.

2.3 Инициализация матрицы

Каждый раз происходит инициализация матрицы $\mathbf{A}$ размера $n \times n$ случайными значениями, при этом достигается положительная определенность и симметричность матрицы $\mathbf{A}$. Для этого сперва происходит инициализация матрицы $\mathbf{B}$ размера $n \times n$ случайными числами от 0 до 1, далее считается сама матрица $\mathbf{A}$:

$$\mathbf{A} = (\mathbf{B} + \mathbf{B}^{\mathrm{T}})/2 + n\mathbf{I}. \tag{2}$$

Таким образом, кроме всех нужных условий на симметричность и положительную определенность, матрица $\mathbf{A}$ оказывается еще и матрицей с диагональным преобладанием.

3 Проверка корректности работы параллельной версии программы

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

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

Для случая с 1 потоком получим

$$\mathbf{x} = \begin{pmatrix} 9.99999940 \cdot 10^{-02} \\ 9.99999940 \cdot 10^{-02} \\ 0.100000009 \\ 0.100000009 \\ 0.100000001 \end{pmatrix},$$

а для случая с 4 потоками ответ будет такой же

$$\mathbf{x} = \begin{pmatrix} 9.99999940 \cdot 10^{-02} \\ 9.99999940 \cdot 10^{-02} \\ 0.100000009 \\ 0.100000009 \\ 0.100000001 \end{pmatrix}.$$

Видно, что ответы совпадают с верным $\mathbf{x} = (0.1, 0.1, 0.1, 0.1, 0.1)^{\mathrm{T}}$ с машинной точностью (используется тип чисел с плавающей точкой `real*4`).

4 Методика организации численных экспериментов

Время работы оптимизированной программы при различном числе потоков и различной вычислительной сложности (для $n = 100,\ 1100,\ 2500$) мерилось с помощью bash-сценария, который представлен в Листинге 5.

Листинг 5: Сценарий запуска численных экспериментов на языке bash.

```
#!/bin/bash
# compiling the program
gfortran linear_algebra_cholesky.f90 -o linear_algebra_cholesky -fopenmp
# Entering the maximum number of threads and the amount of experiments
echo "Enter␣maximum␣number␣of␣threads"
read max_num_threads
echo "Enter␣amount␣of␣experiments"
read experiments
# Running the program in the following loop
for (( experiment=1; experiment<=$experiments; experiment++ ))
do
   echo "Experiment␣#$experiment"
   for (( num_threads=1; num_threads<=$max_num_threads; num_threads++ ))
   do
      export OMP_NUM_THREADS=$num_threads
      echo "num_threads=$num_threads"
      filename=../data/1/$experiment.$num_threads.txt
      echo "#␣n;␣Execution␣time,␣sec" > $filename
      for n in `seq 100 1100 2500`
      do
         echo "n=$n"
         # keeping time data
         echo "$n" | ./linear_algebra_cholesky >> $filename
      done
   done
done
```

5 Результаты измерений

5.1 Время работы алгоритма

Было измерено время работы оптимизированной программы при различном числе потоков и различной вычислительной сложности (для $n = 100,\ 1100,\ 2500$). Для каждого n измерения проводились по три раза. Время определялось с помощью функции `omp_get_time()`. Результаты измерений приведены в Таблице 2. На Рис. 1 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 Ускорение алгоритма

Чтобы определить ускорение, для каждого $n = 100,\ 1100,\ 2500$ 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

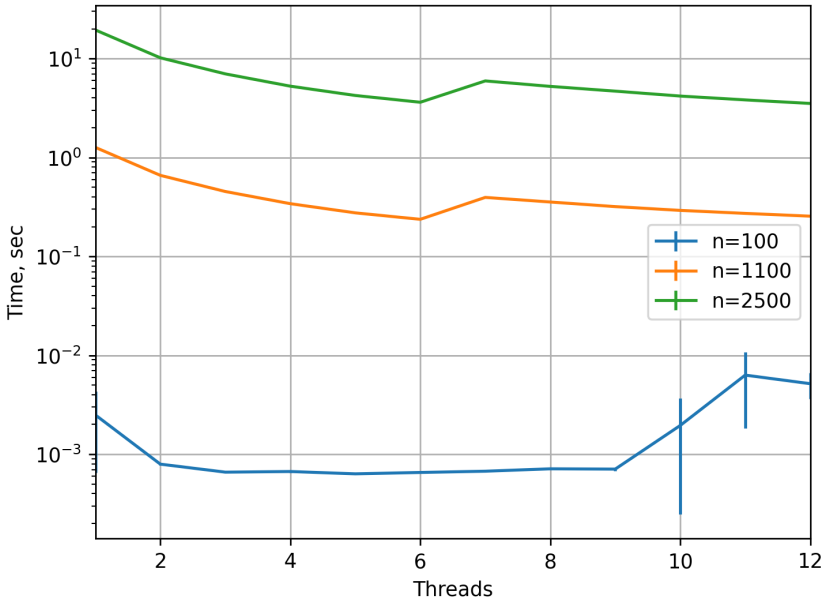

Рис. 1: Время выполнения оптимизированной программы в секундах при различном числе потоков. Показано среднее за 3 эксперимента время. Вертикальными линиями отмечены плюс и минус одно стандартное отклонение (корень из дисперсии)

режиме. Далее считается само ускорение алгоритма с матрицей размера $n \times n$ для T потоков по формуле

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

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. 2 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, как видно на Рис. 2 для n=1100 и 2500, закон Амдала выполняется при числе потоков от 1 до 6, при этом удается достичь ускорения, близкого к числу потоков. Затем, при переходе от числа потоков, равного 6, к числу потоков, равному 7, происходит резкое падение ускорения. При дальнейшем увеличении числа потоков снова виден рост ускорения. Вероятно, падение ускорения при увеличении числа потоков от 6 до 7 связано с особенностями устройства вычислительной машины.

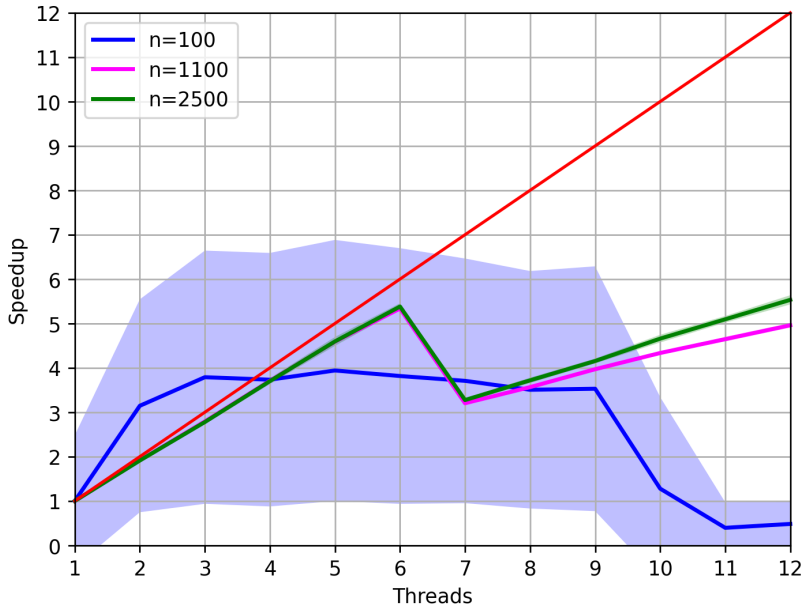

Рис. 2: Ускорение оптимизированной программы при различном числе потоков, закрашенные области соответствуют плюс и минус одному стандартному отклонению (для некоторых линий закрашенных областей может быть не видно, это свидетельствует лишь о малости ошибки). Красная линия обозначает идеальный случай, когда ускорение равно числу потоков.

При малой вычислительной сложности задачи, как видно на Рис. 2 для n=100, трудно сделать выводы относительно выполнимости закона Амдала из-за большой погрешности определения ускорения. Однако, можно заметить, что при большом числе потоков видно уменьшение ускорения. Это можно связать с тем, что при малой сложности задачи на обмен информации между потоками тратится число операций, сравнимое с числом арифметических операций в задаче.

Таким образом, наиболее эффективно поддаются распараллеливанию задачи более трудоемкие (n больше либо порядка 1000).

Приложение

Листинг 6: Исходный код, оптимизированного с помощью OpenMP решателя системы (1), основанного на разложении Холецкого, на Фортране.

```
program linear_algebra_cholesky
   use omp_lib
   implicit none
   integer, parameter :: m = 10
   integer :: n
   real(4), allocatable :: a(:,:), l(:,:), l_t(:,:), id(:,:)
   real(4), allocatable :: b(:,:), x(:,:), y(:)
   integer :: i, j, ip
   real(8) :: time, s
   ! Reading n
   read(*, *) n
   ! Id matrix
   allocate(id(n,n))
   ForAll(i = 1:n, j = 1:n) id(i,j) = (i/j)*(j/i)
   ! Initialization of the matrix A
   allocate(a(n,n))
   call random_number(a)
   a = (a + transpose(a)) / 2 + n * id
   ! Matrix b initialization
   allocate(b(n,m))
   b(:,:) = 1.0
   ! Initialization of a time counter
   time = omp_get_wtime()
   ! Cholesky decomposition
   allocate(l(n,n))
   l = a
   do  i = 1, n
      s = l(i,i)
      do  ip = 1, i - 1
         s = s - dprod(l(i,ip), l(i,ip))
      end do
      l(i,i) = sqrt(s)
      !$omp parallel do private(j,ip,s) shared(l,i) schedule(static)
      do  j = i + 1, n
         s = l(j,i)
         do  ip = 1, i-1
            s = s - dprod(l(i,ip), l(j,ip))
         end do
         l(j,i) = s / l(i,i)
      end do
      !$omp end parallel do
   end do
   ! back substitution
```

```
   allocate(l_t(n,n))
   l_t = transpose(l)
   allocate(y(n))
   allocate(x(n,m))
   !$omp parallel do private(i,j,y,s) shared(x,l,l_t) schedule(static)
   do i = 1, m
      y(1) = b(1,i) / l(1,1)
      do j = 2, n
         s = b(j,i)
         do ip = 1, j - 1
            s = s - dprod(l(j,ip), y(ip))
         end do
         y(j) = s / l(j,j)
      end do
      x(n,i) = y(n) / l_t(n,n)
      do j = n - 1, 1, -1
         s = y(j)
         do ip = j + 1, n
            s = s - dprod(l_t(j,ip), x(ip,i))
         end do
         x(j,i) = s / l_t(j,j)
      end do
   end do
   !$omp end parallel do
   ! Time
   time = omp_get_wtime() - time
   write(*,*) n, time
   deallocate(id)
   deallocate(a)
   deallocate(b)
   deallocate(l)
   deallocate(l_t)
   deallocate(y)
   deallocate(x)
end program linear_algebra_cholesky
```

Threads \n	100			1100			2500		
	Exp#1	Exp#2	Exp#2	Exp#1	Exp#2	Exp#3	Exp#1	Exp#2	Exp#3
1	5.11e-03	1.17e-03	1.21e-03	1.27e+00	1.26e+00	1.26e+00	1.97e+01	1.92e+01	1.95e+01
2	7.93e-04	8.16e-04	7.73e-04	6.59e-01	6.63e-01	6.59e-01	1.02e+01	1.02e+01	1.01e+01
3	6.58e-04	6.70e-04	6.49e-04	4.49e-01	4.58e-01	4.53e-01	6.95e+00	7.08e+00	6.99e+00
4	6.73e-04	6.87e-04	6.47e-04	3.39e-01	3.43e-01	3.43e-01	5.29e+00	5.21e+00	5.29e+00
5	6.32e-04	6.39e-04	6.31e-04	2.78e-01	2.75e-01	2.75e-01	4.19e+00	4.22e+00	4.31e+00
6	6.49e-04	6.68e-04	6.46e-04	2.38e-01	2.36e-01	2.38e-01	3.63e+00	3.60e+00	3.62e+00
7	6.74e-04	6.76e-04	6.71e-04	3.97e-01	3.91e-01	3.97e-01	5.97e+00	5.93e+00	5.94e+00
8	7.33e-04	7.12e-04	6.93e-04	3.53e-01	3.58e-01	3.54e-01	5.28e+00	5.22e+00	5.21e+00
9	7.07e-04	7.44e-04	6.72e-04	3.19e-01	3.18e-01	3.20e-01	4.71e+00	4.68e+00	4.68e+00
10	4.37e-03	7.55e-04	7.40e-04	2.92e-01	2.91e-01	2.93e-01	4.23e+00	4.14e+00	4.17e+00
11	6.11e-03	9.19e-04	1.19e-02	2.75e-01	2.70e-01	2.72e-01	3.80e+00	3.85e+00	3.82e+00
12	5.31e-03	3.24e-03	6.96e-03	2.53e-01	2.56e-01	2.57e-01	3.50e+00	3.56e+00	3.50e+00

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.