8 Практические задания

8.1 Задание 1: многопоточная реализация операций с сеточными данными на неструктурированной смешанной сетке, решение СЛАУ

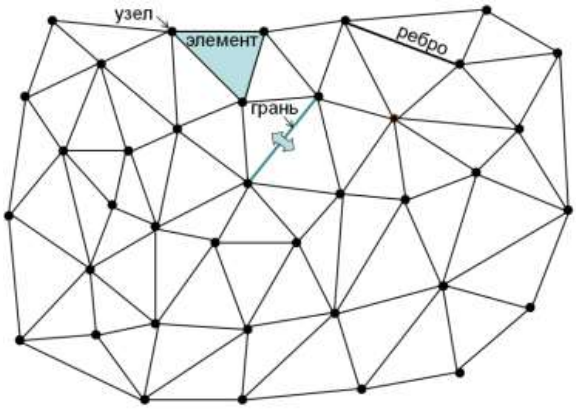
Цели задания:

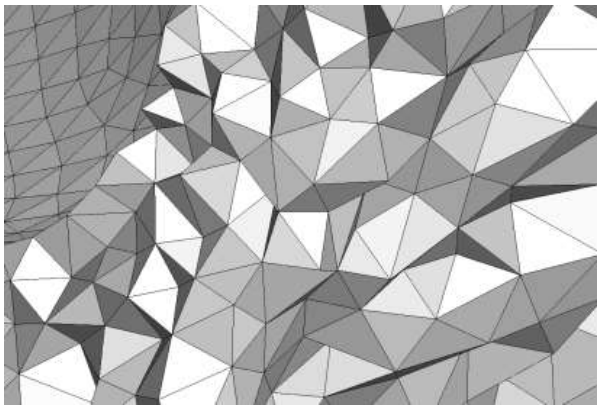
- освоение базовых структур данных для представления неструктурированной сетки, графа связей расчетных ячеек, портрета разреженной матрицы;
- постижение взаимосвязи между сеткой, графом и разреженной матрицей;
- освоение многопоточного распараллеливания простейших операций.

В качестве наиболее простой и репрезентативной операции выбрано матрично-векторное произведение с разреженной матрицей, портрет которой соответствует дискретизации на неструктурированной сетке. Задание будет разделено на этапы.

8.1.1 Введение

Для дискретизации по пространству используется неструктурированная сетка. Сетка – это разбиение некоторой пространственной расчетной области на сеточные элементы (в двухмерном случае – многоугольники, в трехмерном – многогранники). Сетка задана в виде набора узлов и набора элементов. Объединение сеточных элементов полностью заполняет нашу расчетную область, а сами элементы не наползают друг на друга, а стыкуются через общие грани и узлы. Ниже на картинке слева показан пример двухмерной треугольной сетки. Двухмерной – для простоты рисования картинок. В случае трехмерной сетки (справа) всё, по сути, то же самое, для наших целей никакой разницы.



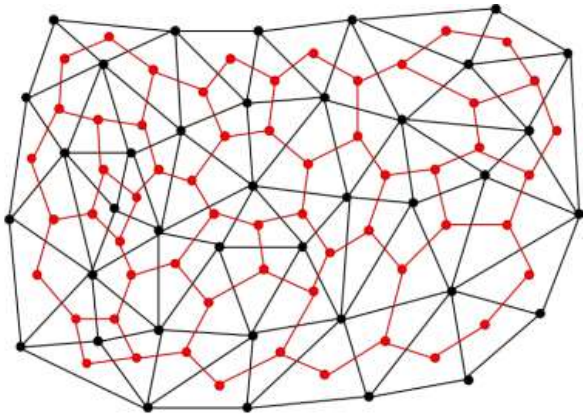


У нас есть несколько наборов сеточных объектов – узлов, элементов, ребер и граней этих элементов. На картинке (слева) элементы – это треугольнички, но могут быть и всякие другие многоугольники. А ребра и грани в двухмерном случае – это одно и то же. В трехмерном случае (как справа) грани станут многоугольниками – гранями многогранных элементов (их видно на картинке справа), а ребра останутся ребрами.

Теперь надо понять, где на сетке заданы сеточные функции, то есть, какие-то переменные, хранящиеся в сеточных объектах. Это будет дискретизация. В частности, можно задавать переменные 1) **в узлах**, а можно 2) **в элементах**. В первом случае контрольные объемы, то есть расчетные ячейки или просто ячейки, будут как-то строиться вокруг узлов, во втором – ячейками будут сами сеточные элементы.

Для дискретизации на сетке есть граф, описывающий связи между ячейками, то есть смежность ячеек (элементов или узлов). В таком графе связей (или смежности) ячеек вершины графа соответствуют ячейкам – элементам или узлам сетки. Ребрам графа соответствуют общие грани между ячейками. Если между ячейками i и j есть общая грань, то в графе вершины i и j соединены ребром. Когда ячейки – это элементы, то все понятно, есть общая грань, значит элементы смежные. Когда ячейки строятся вокруг узлов, получится некая двойственная (дуальная) сетка, грани которой будут проткнуты сеточными ребрами. Поэтому грани между ячейками можно взаимно однозначно сопоставить сеточным ребрам. Тогда ячейки вокруг узлов будут смежными, если узлы имеют соединение ребром – тоже все просто.

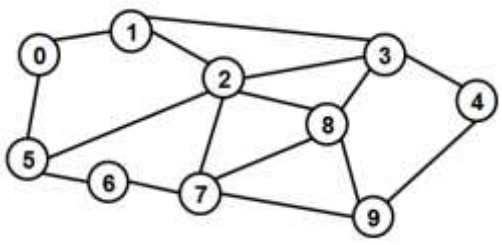
Наша сеточка на картинке, это, по сути, и есть граф. Но это узловой граф. Его вершины соответствуют узлам. Это как раз и будет граф для случая 1 с определением переменных в узлах. Для случая 2 с определением переменных в элементах нужен двойственный (dual) граф. Вот тут он красным нарисован.



Теперь у нас есть вершины графа, с которыми ассоциированы ячейки и заданные в них наборы переменных, и его ребра, которым соответствуют связи между ячейками. Если ячейки построены вокруг узлов, то это черный граф на картинке выше, если ячейками являются элементы, то это красный граф.

У графа есть матрица смежности. Это такая матрица N×N, где N – число вершин в графе, в которой ненулевые позиции соответствуют ребрам между вершинами графа. Коэффициент матрицы a_{ij} в j-м столбце i-й строки равен 1, если между вершинами i и j графа есть ребро, иначе там ноль. Если такое ребро есть, то и a_{ji} – неноль. Значит, число ненулей в матрице равно удвоенному числу ребер в графе.

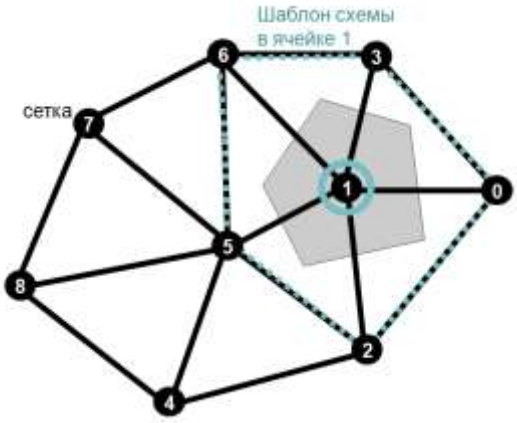
Возьмем какой-нибудь граф. Построим для него матрицу смежности и запишем ее портрет. Вот такой граф и его матрица для примера:



	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0
2	0	1	0	1	0	1	0	1	1	0
3	0	1	1	0	1	0	0	0	1	0
4	0	0	0	1	0	0	0	0	0	1
5	1	0	1	0	0	0	1	0	0	0
6	0	0	0	0	0	1	0	1	0	0
7	0	0	1	0	0	0	1	0	1	1
8	0	0	1	1	0	0	0	1	0	1
9	0	0	0	0	1	0	0	1	1	0

Часто в вычислительных алгоритмах приходится иметь дело с разреженными матрицами, портрет которых имеет ненули на главной диагонали, а во внедиагональных позициях совпадает с портретом матрицы смежности. С такими матрицами мы и будем работать: **матрица смежности + главная диагональ**.

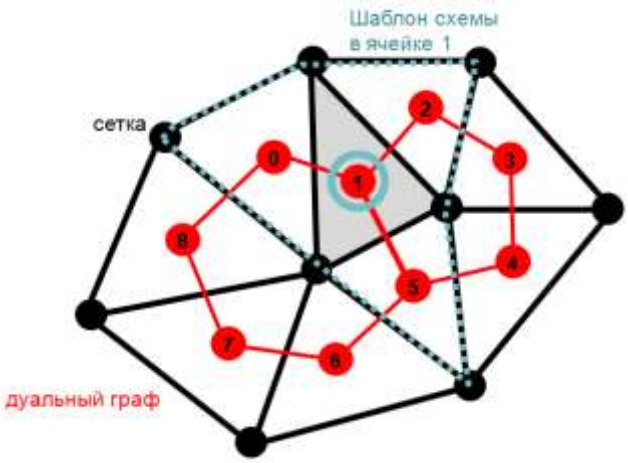
Вот пример треугольной сетки с определением сеточных функций в узлах сетки:



	0	1	2	3	4	5	6	7	8
0	#	#	#	#					
1	#	#	#	#		#	#		
2	#	#	#		#	#			
3	#	#		#			#		
4			#		#	#			#
5		#	#		#	#	#	#	#
6		#		#		#	#	#	
7						#	#	#	#
8					#	#		#	#

В этом примере шаблон схемы в ячейке состоит из этой ячейки и соседних (смежных) с ней ячеек, т.е. ячеек, имеющих с данной ячейкой соединение ребром сетки. Номера в узлах обозначают номера неизвестных в векторе. Справа показан портрет матрицы. Номер узла в сетке соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам соседних с ним узлов. Также присутствует диагональный элемент – каждый узел сам себе сосед.

На следующем рисунке показан пример для случая определения сеточных функций в элементах сетки. Пока нас интересует только портрет (то есть расположение ненулей). Значения коэффициентов матрицы определяются используемым численным методом. Способ их расчета в рамках данного задания нас не интересует, мы их нагенерим потом как попало.



	0	1	2	3	4	5	6	7	8
0	#	#							#
1	#	#	#			#			
2		#	#	#					
3			#	#	#				
4				#	#	#			
5		#			#	#	#		
6						#	#	#	
7							#	#	#
8	#							#	#

В этом примере шаблон схемы в ячейке состоит из этой ячейки и ее соседних ячеек, т.е. ячеек, имеющих с данной ячейкой общую грань. Справа показан портрет матрицы. Номера в узлах обозначают номера неизвестных в векторе. Номер элемента в сетке (красные кружки) соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам

соседних с ним элементов. Также присутствует диагональный элемент – каждый элемент сам себе сосед.

Формат хранения графа и разреженной матрицы

Граф можно описать в виде портрета его матрицы смежности. Запишем портрет матрицы, то есть описание, где в матрице ненулевые позиции, в построчено-разреженном формате CSR (Compressed Sparse Row). Пусть в графе N вершин и E ребер. Для хранения этого добра нам нужны 2 массива:

IA – целочисленный, размера N+1 (также называют rowptr, xadj, ...). В нем для каждой строки храним позицию начала списка столбцов данной строки. Последний элемент в массиве, IA[N], хранит общее число ненулей в матрице (для матрицы смежности графа – равное удвоенному числу ребер графа – 2E).

JA – целочисленный, размера IA[N] (также называют colptr, adjncy, ...). В нем подряд для всех строк матрицы (упорядоченно по строкам) хранятся номера столбцов с ненулевыми значениями. И пусть для определенности номера столбцов каждой строки упорядочены по возрастанию.

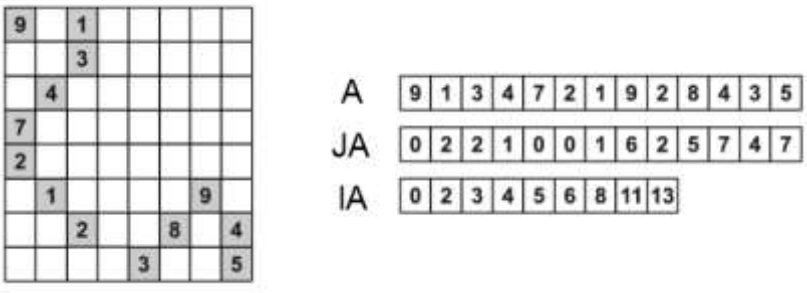
Как нам, например, перебрать все ненулевые столбцы i-й строки? Что, по сути, то же самое, что перебрать всех соседей i-й вершины. Вот так, например:

```
for(int col=IA[i]; col<IA[i+1]; ++col){
    j = JA[col]; // номер соседней вершины, т.е. соединенной с вершиной i ребром.
    ...
}
```

Это был портрет матрицы. Для того, чтобы это стало матрицей, нам нужны коэффициенты. Они хранятся в третьем массиве:

A – вещественный, размера IA[N]. В нем подряд по всем строкам хранятся коэффициенты матрицы.

Пример:



Доступ к ненулевым коэффициентам i-й строки:

```
for(int col=IA[i]; col<IA[i+1]; ++col){
    int j = JA[col]; // номер столбца
    double a_ij = A[col]; //значение коэффициента aij
    . . .
}
```

В случае произвольной неструктурированной сетки максимальное число смежных узлов никак не ограничено. Обычно на практике для тетраэдральной сетки в среднем у узлов где-то

по 15 соседей, а максимальное число может быть и больше 30 легко. Для случая 1 с определением переменных в узлах хорошо подходит формат CSR.

Для случая 2 число соседей ограничено максимальным числом граней элементов. Для тетраэдральной сетки – это всего 4. Для смешанной сетки с элементами до 6 граней число соседей тоже будет варьироваться не сильно, от 4 до 6, не считая граничных ячеек, которых мало, на них можно не обращать внимание. Тут может лучше подойти формат ELLPACK. Этот формат предполагает, что у всех строк равное число ненулевых коэффициентов. Раз оно равное, то мы просто берем CSR, выкидываем из него массив IA, который больше не нужен, а позицию строки в векторах JA и A находим, просто умножая номер строки i на константу, пусть m, равную числу ненулей в строке. Получим то же самое, но без IA:

```
for(int col=i*m; col<(i+1)*m; ++col){
    int j = JA[col]; // номер столбца
    double a_ij = A[col]; //значение коэффициента aij
    . . .
}
```

Но что делать со строками, в которых ненулей меньше, чем m? С граничными ячейками, например, у которых есть внешние грани, по которым нет соседей. Если нет ненулей, то доберем нулями. Вместо каких-то ненулевых коэффициентов окажутся нули, ну ничего страшного. Ну в A мы легко положим 0, а что подставить в JA? В JA подсунем, например, номер столбца последнего ненулевого коэффициента в строке, чтобы не надо было читать никаких лишних позиций из входного вектора.

Что такое сетка разобрались, как хранить матрицы, связанные с сеткой, тоже поняли. Можно перейти к постановке задачи.

8.1.2 Постановка задачи

Работа программы, которую будем делать, состоит из 4 этапов:

1. **Generate** – генерация графа/портрета по тестовой сетке;
2. **Fill** – заполнение матрицы по заданному портрету;
3. **Solve** – решение СЛАУ с полученной матрицей;
4. **Report** – проверка корректности программы и выдача измерений.

Сделать задание можно на языках C, C++. С новыми стандартами C++11 и выше (на момент написания этого текста) могут быть проблемы на кластерах ВМК, особенно на Блюджине.

Этап 1: Generate – генерация портрета на основе тестовой сетки

Для простоты будем иметь дело с двухмерной неструктурированной смешанной сеткой, состоящей из треугольников и четырехугольников.

Кто хочет поиграться с 3D – берем за основу двухмерную сетку и вытягиваем сколько угодно призматических слоев. Будет вполне себе трехмерная смешанная сетка их гексаэдров и треугольных призм.

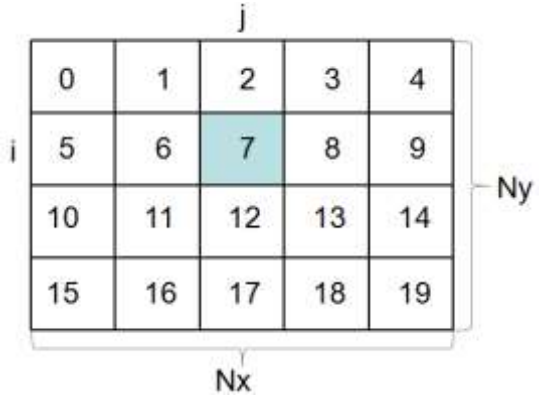
Есть два варианта определения сеточных функций

A) в узлах;

B) в элементах.

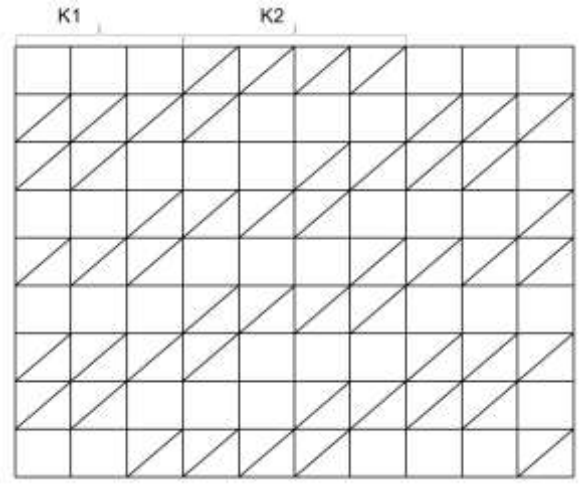
Для упрощения генерации тестовой сетки будем использовать самое просто, что только может быть – двухмерную решетку.

Решетка состоит из N = Nx×Ny клеточек. У каждой клеточки есть позиция (i, j), где i – номер строки в решетке, j – номер столбца. Нумерация клеточек в решетке построчно слева направо, сверху вниз. Номер клеточки в единой общей нумерации: I = i*Nx + j.



Итак, мы задали два параметра Nx, Ny, получили решетку. Теперь получим из решетки сетку. Чтобы сетка была смешанной, то есть чтобы в ней были разные типы элементов, часть квадратных клеток поделим на треугольники. Для этого введем еще два параметра K1, K2, чтобы регулировать соотношение числа треугольников и четырехугольников. Будем делить клетки следующим образом. K1 клеток не делим, K2 следующих клеток делим, K1 следующих клеток снова не делим, K2 следующих клеток снова делим, и так далее. Легко догадаться, что клетка поделена на треугольники, если I%(... впрочем, вы сами легко напишете формулу.

Пример для Nx=10, Ny=9, K1=3, K2=4:



Да, поделить четырехугольник можно двумя способами:

Способ 1 Способ 2

Получились следующие варианты:

А) ячейки, то есть вершины графа = **узлы**

Б) ячейки, то есть вершины графа = **элементы**

1) делим клетки решетки **способом 1**

2) делим клетки решетки **способом 2**.

Итого 4 варианта: **А1, А2, Б1, Б2**.

Ну это чтобы всем не делать одно и то же. Жизнь же должна быть сложной? И вам будет чуть сложнее передирать, и проверяющим сложнее проверять.

Вариант определяем, например, по списку группы по алфавиту. Берем порядковый номер в списке, находим остаток от деления на 4, по нему берем один из этих вариантов в указанном выше порядке.

На этапе 1 нужно сгенерировать "топологию" связей ячеек, то есть построить графа, и по нему сгенерировать портрет матрицы смежности. И дополнить этот портрет главной диагональю! Обязательно. **Не забываем главную диагональ!**

Входные данные:

Nx, Ny – число клеток в решетке по вертикали и горизонтали;

K1, K2 – параметры для количества однопалубных и двухпалубных кораблей треугольных и четырехугольных элементов.

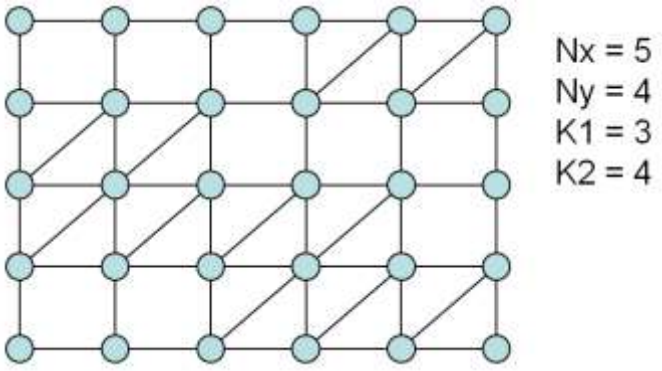
Выходные данные:

N – размер матрицы = число вершин в графе;

IA, JA – портрет разреженной матрицы смежности графа, дополненный главной диагональю (в формате CSR, при желании для случая Б можно делать ELLPACK).

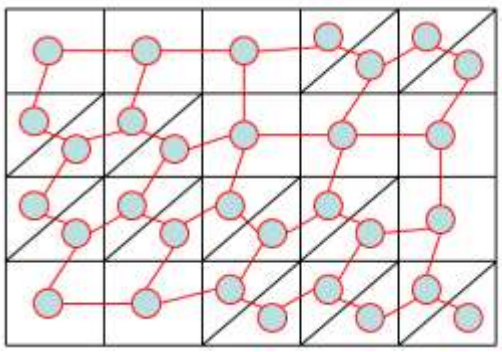
Чтобы было понятнее, разберемся еще подробнее с разными вариантами, что там будет графом, и как что нумеруется.

Вариант А – вершины графа = узлы сетки. Узлы сетки – это точки пересечения сеточных линий. Вот они кружочками отмечены:



Граф связей узлов совпадает с сеткой, он тут и нарисован на этой картинке. Вершин в графе N = (Nx+1)×(Ny+1). Нумерация узлов точно такая же – слева направо, сверху вниз. Удобство этого варианта – сразу понятно число вершин, оно не меняется от разбиения на треугольники. Неудобство – чтобы посчитать число соседей у вершины, надо смотреть, что делается в соседних клетках.

Вариант В – вершины графа = элементы сетки, то есть четырехугольники и треугольники. Граф связей – двойственный граф сетки. Элементы считаем смежными, если у них есть общая грань. Граф будет такой:



Нумерация элементов такая же, слева направо, сверху вниз. Что там из них левее/правее/выше/ниже – определяем по центру масс элемента. Удобство этого варианта – сразу понятно, сколько соседей у ячеек. Неудобство – число ячеек зависит от параметров разбиения на треугольники.

Во всех вариантах не забываем, что к портрету надо добавлять главную диагональ.

Генерацию надо сделать в виде функции с входными и выходными данными в качестве аргументов.

В main только считываются параметры пользовательского ввода (из командной строки или файла) и вызывается эта функция генерации. Интерактивный ввод не надо использовать. Ну то есть не надо никаких cin >> и прочих scanf. Обычно расчетные коды запускаются через систему очередей кластера не в интерактивном режиме, поэтому интерактивный ввод неуместен.

Программа должна брать хотя бы один аргумент командной строки. Если ввод параметров через файл, то это имя входного файла. Если программа запущена без аргументов, она должна напечатать хелп, как ей пользоваться.

Программа должна иметь проверку корректности пользовательского ввода. В случае ошибки она должна выйти с информативным сообщением, а не по какому-нибудь сегфолту.

Для проверки результатов программа должна уметь печатать выходные данные в текстовый файл или в stdout. Для этого должен быть параметр, включающий отладочную печать. Но по умолчанию печать должна быть выключена. Иначе на больших размерах будет печально.

Для проверки задания можно:

1) напечатать выходные данные для маленькой сетки, которую можно просто нарисовать на бумаге и проверить;
2) сравнить выходные данные с результатами других авторов с тем же вариантом задания;
3) проверить работоспособность с N порядка 10^6, 10^7, взять сеточку 2000x2000 хотя бы;
4) проверить, что расход памяти и время работы программы растут линейно с ростом числа ячеек;
5) сравнить время работы с другими авторами.

Этап 2: Fill – построение СЛАУ по заданному портрету матрицы

Входные данные:

N – размер (квадратной матрицы NxN);

IA, JA – портрет матрицы.

Выходные данные:

A – массив ненулевых коэффициентов матрицы (размера IA[N]);

b – вектор правой части (размера N).

На этом этапе мы уже ничего не знаем о том, что сетка когда-то была решеткой. **Забыли про решетку!** Теперь мы работаем только с топологией, то есть с описанием связей между ячейками/объектами/вершинами графа в общем виде.

Для заполнения матрицы и правой части лучше использовать одинаковые формулы, чтобы можно было сравнивать результаты программы друг с другом для проверки (для одинаковых вариантов, естественно).

Например, для внедиагональных элементов строки можно взять какой-нибудь косинус от индексов строки и столбца, $a_{ij} = cos(i * j + i + j)$, $i \neq j$, $\boldsymbol{j} \in \boldsymbol{Col(i)}$, где Col(i) – множество номеров столбцов для строки i, которые входят в портрет матрицы (это то, что лежит в JA[IA[i],...,IA[i+1]–1]).

Диагональный элемент в каждой строке будем вычислять как сумму модулей внедиагональных элементов строки, умноженную на больший единицы коэффициент:

$$a_{ii} = 1.234 \sum_{j, j \neq i} |a_{ij}|.$$

Все остальные коэффициенты матрицы равны нулю (К.О.). Домножаем диагональ на коэффициентик, чтобы у матрицы было диагональное преобладание. Это нужно, чтобы решатель (солвер – от *англ.* solver), который мы дальше будем делать, устойчиво сходился к решению, а не болтался в проруби.

Заполняем матрицу так: идем по строкам, перебираем номера столбцов, если внедиагональный коэффициент – заполняем по формуле и прибавляем значение в сумму по строке, если диагональный – запоминаем позицию и пропускаем. Дошли до конца строки, домножили полученную сумму по строке на коэффициент и записали в диагональный элемент. Перешли к следующей строке, занулив сумму.

Вектор правой части заполняем по формуле $b_i = sin(i)$.

Когда функции для этапов 1 и 2 сделаны и корректно работают, нужно внедрить многопоточное распараллеливание средствами OpenMP. В связи с этим у программы появляется еще один **входной параметр:**

T – число нитей (threads).

Функции этапов 1 и 2 нужно распараллелить. Построение портрета, заполнение коэффициентов – всё должно выполняться параллельно T нитями.

Чтобы оценить эффект от распараллеливания, нужно добавить замеры времени. Для этого можно использовать функцию omp_get_wtime.

Замечания по выдаче. Программа может сообщать в лог о прохождении этапов, выдавать контрольные величины (например, число вершин графа, число ребер, число ненулей в портрете, ...), выдавать табличку замеров времени. По замерам будем проверять параллельное ускорение.

По умолчанию, без запроса пользователя, программа не должна никуда печатать никаких массивов, ни портрет, ни коэффициенты, ни в stdout, ни в файл. Выдача отладочной

информации – только в режиме отладки. Размеры сетки, которые надо будет тестировать, это 1 ~ 10 млн ячеек (К.О. сообщает: на таких размерах печатать портрет– плохая идея).

Результаты работы программы в параллельном и последовательном режиме должны быть полностью идентичны. Для проверки можно, например, печатать в файл массивчики и сравнивать файлы. Можно вычислять по массивам контрольные суммы и сравнивать просто числа, что удобнее, особенно на больших размерах. Можно и так, и эдак, но массивы – только в режиме отладки, а контрольные величины можно и всегда выдавать.

Этап 3: Solve – решение СЛАУ итерационным методом

Входные данные:

N – размер (квадратной матрицы N×N);

IA, JA – портрет матрицы;

A – массив ненулевых коэффициентов матрицы (размера IA[N]);

b – вектор правой части (размера N);

eps – критерий остановки (ε), которым определяется точность решения;

maxit – максимальное число итераций.

Выходные данные:

x – вектор решения (размера N);

n – количество выполненных итераций;

res – L2 норма невязки (невязка – это вектор $\boldsymbol{r} = \boldsymbol{Ax} - \boldsymbol{b}$).

Поскольку матрица симметричная, будем использовать простейший метод сопряженных градиентов CG (Conjugate Gradient) с диагональным предобуславливателем.

Решаем СЛАУ $\boldsymbol{Ax} = \boldsymbol{b}$

Будем использовать такой алгоритм предобусловленного метода CG:

$\boldsymbol{x}_0 = 0$

$\boldsymbol{r}_0 = \boldsymbol{b}$

$k = 0$

do

$k = k + 1$

$\boldsymbol{z}_k = \boldsymbol{M}^{-1}\boldsymbol{r}_{k-1}$ // SpMV

$\rho_k = (\boldsymbol{r}_{k-1}, \boldsymbol{z}_k)$ // dot

if $k = 1$ **then**

$\boldsymbol{p}_k = \boldsymbol{z}_k$

else

$\beta_k = \rho_k / \rho_{k-1}$

$\boldsymbol{p}_k = \boldsymbol{z}_k + \beta_k \boldsymbol{p}_{k-1}$ // axpy

end if

$\boldsymbol{q}_k = \boldsymbol{A}\boldsymbol{p}_k$ // SpMV

$\alpha_k = \rho_k / (\boldsymbol{p}_k, \boldsymbol{q}_k)$ // dot

$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$ // axpy

$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{q}_k$ // axpy

while $\rho_k > \varepsilon$ **and** $k <$ *maxit*

В формулах вектора и матрицы обозначены жирным шрифтом, из них матрицы обозначены прописными (заглавными) буквами.

В случае диагонального предобуславливателя матрица $\boldsymbol{M}$ – просто диагональная матрица, с диагональю из матрицы $\boldsymbol{A}$. Начальное приближение – нулевое.

Для отладки надо на каждой итерации вычислять L2 норму невязки и печатать в стандартный поток вывода с новой строки номер итерации и текущую норму невязки системы на данной итерации. Этот лог итерационного процесса можно сравнивать друг с другом для проверки (у кого одинаковые варианты, естественно). На выходе из солвера надо проверить финальную невязку и вернуть ее в переменную *res*.

Для решателя понадобится сделать несколько кернелов, то есть вычислительных функций для базовых операций в солвере. Основные из них:
SpMV – матрично-векторное произведение с разреженной матрицей (sparse matrix-vector);
dot – скалярное произведение;
axpy – поэлементное сложение двух векторов с умножением одного из них на скаляр, $\alpha\boldsymbol{x} + \mathbf{y}$.
Помимо этих трех базовых операций, там еще будут кернелы копирования векторов, заполнения константой.

Рекомендуется сделать проверочные тесты для каждой из базовых операций. Входные вектора можно заполнять по какой-то формуле от номера элемента. Проверочный тест для каждой операции должен выдать контрольные значения: для dot – просто результат, для axpy и SpMV контрольные значения по выходному вектору, например: сумма всех элементов, L2 норма – sqrt(dot(x, x)), и т.д. Последующие параллельные реализации можно будет сравнивать с исходными последовательными реализациями по этим контрольным значениям для подтверждения корректности.

Для решателя и составляющих его кернелов нужно сделать замеры времени, для чего можно использовать функцию omp_get_wtime().

Для каждой из трех базовых операций тоже нужно сделать замеры.

Нужно попробовать выполнить оптимизацию последовательных версий базовых операций, сохранив также исходные версии для сравнения. Можно применить развертку циклов, SIMD инструкции и т.д. Сделать таймирование по всем базовым операциям – кернелам, оценить улучшения.

Затем нужно сделать многопоточные параллельные реализации всех операций с векторами и матрицами с помощью OpenMP. Вообще все этапы 1 – 3 должны быть полностью распараллелены. В итоге надо получить многопоточную корректно работающую версию решателя, дающую адекватное ускорение при сохранении корректности результата.

Этап 4: Report – проверка корректности программы и выдача измерений

На этом этапе надо просто проверить, что невязка системы после работы решателя удовлетворяет заданной точности, выдать значение фактической невязки, и распечатать табличку таймирования, в которой указано, сколько времени в секундах затрачено на:

1. этап генерации;
2. этап заполнения;
3. этап решения СЛАУ;
4. каждую из трех кернел-функций, базовых алгебраических операций решателя СЛАУ.

Чтобы эти данные получить, по вашей программе надо расставить соответствующие замеры времени.

8.1.3 Отчет

По результатам нужно будет подготовить отчет, в котором привести данные по исследованию производительности решателя. Все вычислительные этапы должны адекватно ускоряться.

Для каждой из базовых операций и для всего алгоритма солвера (этап 3) исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N. Измерить OpenMP ускорение для различных N. Оценить максимально достижимую производительность (TBP) с учетом пропускной способности памяти и сравнить с фактической производительностью. Оценить достигаемую скорость передачи данных между процессором и памятью, получаемый процент от пропускной способности.

Сдавать код нужно вместе с инструкцией по компиляции и запуску, желательно сделать make-файл или какой-то скрипт сборки. Также надо, чтобы исполнимый файл печатал хэлп при запуске без аргументов. Отчет в формате PDF нужно загружать в систему отдельным файлом, а не в общем архиве с кодом.

Требования к отчету:

Титульный лист, содержащий
1.1 Название курса
1.2 Название задания
1.3 Фамилия, Имя, Отчество (при наличии)
1.4 Номер группы
1.5 Дата подачи

Содержание отчета:

2 Описание задания и программной реализации.
2.1 Краткое описание задания.
2.2 Краткое описание программной реализации – как организованы данные, какие функции реализованы (название, аргументы, назначение). Просьба указывать, как программа запускается – с какими параметрами, с описанием этих параметров.
2.3 Описание опробованных способов оптимизации последовательных вычислений (по желанию).

3 Исследование производительности
3.1 Характеристики вычислительной системы: описание одной или нескольких систем, на которых выполнено исследование (подойдет любой многоядерный процессор), тип процессора, количество ядер, пиковая производительность, пиковая пропускная способность памяти.
По желанию – промерять и на своем десктопе/ноуте, и на кластере.
Описание параметров компиляции под конкретную систему
3.2 Результаты измерений производительности
3.2.1 Последовательная производительность
Для всех этапов, включая генерацию и заполнение (то есть этапы инициализации), продемонстрировать линейный рост стоимости, линейный рост потребления памяти. Solve-часть (этап 3) надо исследовать подробнее. Для каждой из трех базовых операций и

для всего алгоритма солвера исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N.

Нескольких N достаточно: N = 10000, 100000, 1000000, 10000000.

Для повышения точности измерений, замеры времени лучше производить, выполняя операции многократно в цикле, чтобы осреднить время измерений. Суммарное время измерений лучше, чтобы получалось хотя бы порядка десятых долей секунды.

Оценить выигрыш от примененной оптимизации (по желанию).

3.2.2. Параллельное ускорение

Для этапов инициализации продемонстрировать наличие параллельного ускорения. Достигаемую производительность для этапов инициализации измерять не надо, считать ТВР не надо.

Для solve-части измерить OpenMP ускорение для различных N для каждой из 3-х базовых операций и для всего алгоритма солвера. Измерить достигаемую производительность. Можно сделать таблички или графики, показывающие GFLOPS от T, где T – число нитей. Можно представить данные по ускорению и производительности единой таблицей, в которой для разных N и T привести ускорение и достигаемую производительность.

4 Анализ полученных результатов

Оценить ТВР и получаемый процент от достижимой производительности для каждой из трех базовых операций, какой процент от пиковой производительности устройства составляет максимальная достигаемая в тесте производительность. Для этапов инициализации ничего оценивать и считать не требуется.

Приложение1: исходный текст программы в файлах C/C++

Требования к программе:

1) Программа должна использовать OpenMP для многопоточного распараллеливания и адекватно ускоряться.
2) Солвер должен корректно работать, т.е. показывать быструю сходимость.
3) Программа должна печатать хелп при запуске без аргументов.
4) При ошибках пользовательского ввода программа должна контролируемо завершаться с диагностическим сообщением.
5) Программа не должна валиться по сигналу или зависать (ни при каких параметрах пользовательского ввода).

8.2 Задание 2: параллельный решатель СЛАУ для кластерных систем с распределенной памятью

Цели задания:

- освоение организации распределенных вычислений и обмена данными в MPI;
- освоение работы с кластерной вычислительной системой.

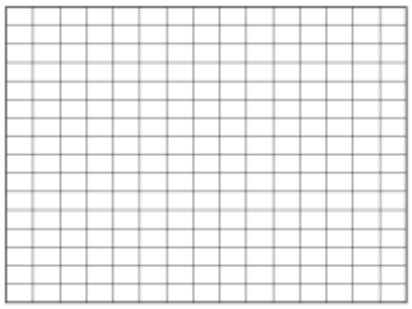
В этом задании необходимо расширить реализацию, сделанную в Задании 1, на вычисления в рамках параллельной модели с распределенной памятью. Это потребует существенной доработки. Для организации работы параллельных процессов и обмена данными используется MPI.

Распределенная реализация подразумевает, что расчетная область задачи разделяется на части – подобласти. Эти подобласти распределяются между параллельными процессами. Каждый процесс работает со своей частью расчетной области и обменивается с соседними процессами только информацией по интерфейсным ячейкам, граничащим с ячейками других подобластей.

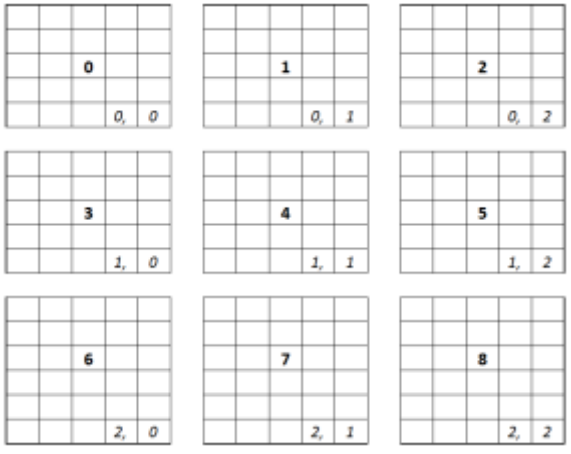
8.2.1 Введение

Рассмотрим на очень простом примере – на примере декартовой решетки. Возьмем и разделим нашу модельную расчетную область, представленную решеткой из N = Nx×Ny ячеек, на P = Px×Py подобластей путем декомпозиции по двум направлениям (осям координат).

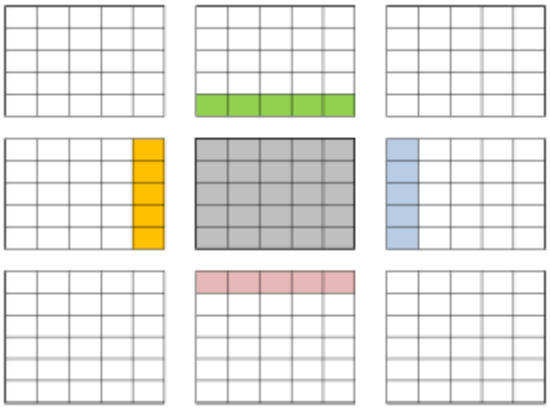
Каждый процесс работает только **со своей частью** расчетной области и с ее гало (т.е. приграничными ячейками из соседних подобластей). Рассмотрим простейший пример – делим на части двухмерную решетку, распределяем ее клеточки-ячейки. Вот есть какая-то решетка:



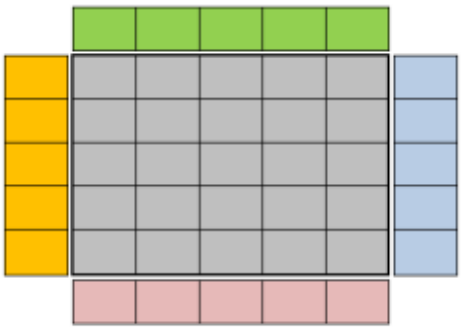
Поделим ее, например, на 9 частей 3×3, части нумеруем по тому же принципу, что и клетки, например, сверху вниз, слева направо (в каждой части по центру – номер процесса и курсивом в углу его позиция в решетке процессов px, py):



Рассмотрим один из процессов, например, который по середине – №4. Ячейки его подобласти отмечены серым. Другими цветами отмечены гало ячейки для этой локальной подобласти, принадлежащие чужим подобластям.



Так выглядит локальная часть, с которой имеет дело 4-й процесс:



О существовании других ячеек этот процесс не знает. В общем это был просто пример, чтобы понять принцип.

8.2.2 Постановка задачи

Делаем Задание 2 на основе Задания 1 (раздел 8.1). Работа программы, которую будем делать, теперь состоит из 5 этапов. Добавился еще этап построения обмена данными:

1. **Generate** – генерация графа/портрета по тестовой сетке;
2. **Fill** – заполнение матрицы по заданному портрету;
3. **Com** – построение схемы обменов;
4. **Solve** – решение СЛАУ с полученной матрицей;
5. **Report** – проверка корректности программы и выдача измерений.

Этап 1: Generate – генерация портрета на основе тестовой сетки

Имеется решетка состоит из **Ng** = Nx×Ny клеточек (g – global). На вход поступают два числа Px и Py, которые указывают, на сколько частей мы делим решетку по соответствующим направлениям. Общее число MPI процессов должно быть P=Px×Py.

Для начала по заданным Nx, Ny и Px, Py, зная ID процесса, определим подобласть данного процесса. Найдем диапазоны индексов подобласти по двум осям, которые мы делим: ib ≤ i < ie, jb ≤ j < je (b – begin, e – end). Диапазоны следуют из позиции данного процесса в решетке процессов, т.е. его порядковых номеров по двум осям. Поделить надо так, чтобы

дисбаланс по каждому направлению был не более 1 шт. (т.е. если Nx не делится нацело на Px, то остаток от деления надо раскидать по одной ячейке по процессам).

Для формирования подобласти процесса еще прибавим к своим клеткам один уровень гало – прилегающие клетки из соседних подобластей.

Модифицируем этап генерации, чтобы сетка строилась только для локальной подобласти (свои + 1 уровень гало). Там можно заменить диапазоны у двух циклов по двум осям на соответствующие границы, которые $ib \le i < ie$, $jb \le j < je$.

В зависимости от варианта мы распределяем узлы или элементы. В варианте по узлам – просто распределяем узлы исходной решетки. В варианте с элементами распределяем клетки исходной решетки, а элементам ставим владельца клетки (если клетку поделили на треугольники, то треугольники принадлежат тому же, кому и клетка).

Далее, чтобы не путаться в этих вариантах, вершинах и элементах, будем говорить в терминах графа. То, что распределяем – соответствует вершинам этого графа (которые сопоставлены узлам или элементам сетки, в зависимости от варианта).

Для всех вершин, попавших в подобласть (свои + гало) надо знать их номера в глобальном графе, то есть во всей сетке.

Отображение из номера вершины в локальной подобласти в номер вершины во всей глобальной области обозначим L2G (Local To Global). Обратное отображение – G2L.

Упорядочим вершины так, чтобы сначала были собственные, потом шли гало. Пусть число вершин в локальной области данного процесса будет **N**, из них собственных вершин – **No** (o – own), во всей глобальной области вершин **Ng** (g – global). Собственные вершины будут с номерами от 0 до No-1, гало вершины с номерами от No до N-1.

На выходе с генерации мы должны получить портрет, который хранит связи только для собственных вершин. IA у нас размера No+1, то есть мы будем распределять матрицу построчно и хранить No наших строк. В эти связи, естественно, входят гало вершины, то есть JA содержит все те же столбцы, что в исходной версии (только в локальной нумерации). Если на генерации JA делался в глобальных номерах, их надо перегнать в локальные, используя G2L.

Заполняем массив Part размера N, определяющий для каждой вершины, кому она принадлежит, т.е. вписываем каждой вершине номер владельца.

Как все это заполнять, все эти нумерации, описано в разделах 4.3.2, 4.3.3. Итак…

Входные данные:

Nx, Ny – число клеток во всей глобальной решетке по вертикали и горизонтали;

K1, K2 – параметры для количества треугольных и четырехугольных элементов;

Px, Py – параметры декомпозиция по двум осям;

Программа запускается через MPI с числом процессов P = Px×Py.

Выходные данные:

N – число вершин в локальной подобласти (собственные и гало);

No – число собственных вершин в локальной подобласти;

IA, JA – локальный портрет разреженной матрицы смежности графа, дополненный главной диагональю (в формате CSR), то есть только собственные строки для данной подобласти (но эти строки, естественно, включают столбцы для гало от соседей!);

Part – массив с номерами владельцев по каждой вершине в локальном графе;

L2G – массив с номерами вершин локального графа в глобальном графе, то есть отображение из локальной нумерации в глобальную.

Этап 2: Fill – построение СЛАУ по заданному портрету матрицы

Входные данные:

N – число вершин в локальной подобласти (собственные и гало);
No – число собственных вершин в локальной подобласти;
IA, JA – локальный портрет матрицы;
L2G – отображение из локальной нумерации в глобальную.

Выходные данные:

A – массив ненулевых коэффициентов матрицы (размера IA[No]), только свои строки;
b – вектор правой части (размера N).

Заполняем матрицу и вектора как в Задании 1, но индексы берем **глобальные**, иначе не будет совпадать с последовательной версией. Не cos(i*j…), например, а cos(L2G[i]*L2G[j]…).

Этап 3: Com – построение схемы обмена данными

Входные данные:

N – число вершин в локальной подобласти (собственные и гало);
No – число собственных вершин в локальной подобласти;
IA, JA – локальный портрет матрицы;
Part – массив с номерами владельцев по каждой вершине в локальном графе;
L2G – отображение из локальной нумерации в глобальную;

Выходные данные:

Com – схема обменов, то есть списки вершин на отправку всем соседям и на прием от всех соседей. Как её делать и какие контейнеры использовать – на ваше усмотрение.

Построение схемы обмена и организация обмена данными подробно описаны в разделе 4.4 (см. 4.4.1, 4.4.2). Просто делаем, как там написано.

Этап 4: Solve – решение СЛАУ итерационным методом

Входные данные:

N – число вершин в локальной подобласти (собственные и гало);
No – число собственных вершин в локальной подобласти;
IA, JA – локальный портрет матрицы;
Com – схема обменов;
A – массив ненулевых коэффициентов матрицы (размера IA[No]);
b – вектор правой части (размера N).
eps – критерий остановки (ε), которым определяется точность решения;
maxit – максимальное число итераций.

Выходные данные:

x – вектор решения (размера N);
n – количество выполненных итераций;
res – L2 норма невязки (невязка – это вектор $\boldsymbol{r = Ax - b}$).

Делаем на основе Задания 1 (раздел 8.1). Добавляем обмены по схеме Com для обновления гало для SpMV (которое делается только для своих позиций)

Добавляем Allreduce обмен для сбора глобальной суммы в скалярном произведении dot (которое делается только для своих).

Axpy остается без изменений.

Проверяем, что солвер работает идентично последовательно версии, что все контрольные величины совпадают.

Этап 4: Report – проверка корректности программы и выдача измерений

Делаем полностью аналогично Заданию 1 (раздел 8.1).

8.2.3 Отчет

Отчет делаем по аналогии с Заданием 1 (раздел 8.1). Только исследование параллельной эффективности делаем на кластере для разного числа процессов, а не потоков.

Делаем тестовые запуски на разном числе процессов, для разных размеров сетки, проверяем, что все работает, что асимптотика по времени и потреблению памяти у всего кода линейная. Тестируем на системе с общей памятью, тестируем комбинированный режим распараллеливания – MPI+OpenMP.