Министерство науки и высшего образования Российской Федерации

Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского

РЕШЕНИЕ СЛАУ С РАЗРЕЖЕННОЙ МАТРИЦЕЙ ПРЯМЫМИ МЕТОДАМИ НА СУПЕРКОМПЬЮТЕРЕ

Учебное пособие

Рекомендовано методической комиссией института информационных технологий, математики и механики для студентов ННГУ, обучающихся по направлениям подготовки 01.04.02. «Прикладная математика и информатика» (магистратура), 02.04.02. «Фундаментальная информатика и информационные технологии» (магистратура)

Нижний Новгород
2023

УДК 519.61
ББК 22.192.31
Б-26

Рецензенты:
доцент, д.т.н., **Н.В. Старостин**,
доцент, д.ф.-м.н, **М.Л. Цымблер**

Бартенев, Ю.Г.

Б-26 **Решение СЛАУ с разреженной матрицей прямыми методами на суперкомпьютере**: Учебное пособие / Ю.Г. Бартенев, Е.А. Козинов, И.Б. Мееров, А.Ю. Пирова. – Нижний Новгород: Нижегородский госуниверситет, 2023. – 50 с.

ISBN 978-5-91326-825-9

В учебном пособии рассматривается задача численного решения системы линейных уравнений (СЛАУ) с разреженной матрицей прямыми методами с помощью специализированных программ (решателей). Описана процедура численного решения СЛАУ, обсуждаются вопросы оценки и улучшения вычислительной точности решения. Приводятся примеры использования двух распространенных решателей: коммерческого пакета Intel oneMKL PARDISO, предназначенного для систем с общей и распределенной памятью, академического пакета MUMPS, предназначенного для систем с распределенной памятью. Приводятся сведения о новом российском решателе USPARS, ориентированном на системы с общей памятью.

Учебное пособие предназначено для студентов института ИТММ ННГУ в качестве дополнительных материалов к курсам «Параллельные численные методы», «Анализ производительности и оптимизация программ».

Ответственный за выпуск:
зам. председателя методической комиссии института
информационных технологий, математики и механики ННГУ,
к.х.н., доцент **Г.В. Кузенкова**

УДК 519.61
ББК 22.192.31

ISBN 978-5-91326-825-9

Оглавление

Введение

При решении многих задач в физике, математике, экономике, финансах требуется решать системы линейных алгебраических уравнений (СЛАУ) с большим числом неизвестных. Так, например, такие СЛАУ возникают при дискретизации систем дифференциальных уравнений в частных производных. Нередко подавляющее число элементов матрицы системы равны нулю «по построению», исходя из особенностей применяемых численных методов. Очевидно, что классическое представление матрицы в виде обычного двумерного массива в этом случае является чрезвычайно неэффективным как в смысле затрат памяти, так и учитывая последующее время решения системы. Работа с *разреженными матрицами*, преимущественно состоящими из нулей, требует разработки и применения соответствующих подходов.

Прежде чем говорить о способах хранения разреженных матриц необходимо определиться, что мы под этим понимаем. Известно несколько определений, из которых наиболее удачным, по нашему мнению, является следующее. Будем считать матрицу *разреженной*, если учет данного фактора позволяет выбрать структуры данных для представления матриц и алгоритмы их обработки так, чтобы сэкономить память и уменьшить время работы для конкретного программно-аппаратного окружения. Отметим, что данное определение хотя и не является строгим в математическом смысле, хорошо отражает суть: у нас есть свобода выбора. Мы можем либо рассматривать матрицу как «плотную» и хранить ее традиционным образом, либо считать ее разреженной и конструировать для работы с ней специальные, более сложные, но существенно более экономичные структуры данных. Разумеется, экономия памяти, в большинстве случаев очень существенная, не дается «бесплатно». Разработка и реализация ключевых матричных алгоритмов для разреженных структур также усложняются, однако вычисления требуют намного меньшего времени в сравнении с использованием плотного представления. Итак, представление матриц, преимущественно состоящих из нулей, в специальных разреженных структурах данных, приводит к значительной экономии памяти и времени счета, но усложняют разработку программ.

Структуры данных и алгоритмы для работы с разреженными матрицами изучаются в рамках «алгебры разреженных матриц» или, менее формально, «разреженной алгебры» – раздела вычислительной математики, имеющего обширное практическое применение при решении многих задач. За

последние несколько десятилетий в данной области накоплен значительный практический опыт. Так, наиболее распространенными являются координатный формат хранения, представляющий ненулевые элементы и их координаты, а также разреженный строчный (или столбцовый) формат (Compressed Sparse Row/Column Storage, CRS/CCS), отличающийся от координатного одним из массивов, хранящим границы между строками (или столбцами). Эти форматы поддерживаются большинством программных библиотек. Учет особенностей этих форматов требует существенной модификации базовых алгоритмов линейной алгебры, таких как транспонирование матрицы, умножение матрицы на вектор, умножение матриц и т.д.

С точки зрения высокопроизводительных вычислений соответствующие алгоритмы характеризуются существенными проблемами производительности. Так, для алгоритмов обработки разреженных структур характерен нерегулярный доступ к памяти, создающий проблемы при кешировании данных, а также значительно снижающий эффективность векторизации вычислений. Различное число ненулевых элементов в строках и столбцах матриц приводит к дисбалансу вычислительной нагрузки при распараллеливании. В целом, низкая арифметическая интенсивность (отношение числа выполненных операций к числу байт задействованной памяти) наряду с перечисленными проблемами приводит к значительному падению производительности по сравнению с пиковыми возможностями оборудования. Нередко удается задействовать лишь 1% вычислительного ресурса. В связи с указанными факторами разрабатываются и другие структуры хранения разреженных матриц, требующие несколько большего объема памяти, но позволяющие улучшить ситуацию и уменьшить время решения задач при использовании параллельных вычислительных устройств.

В данном пособии мы рассматриваем один из классических алгоритмов алгебры разреженных матриц – решение больших СЛАУ с разреженной матрицей (далее – разреженных СЛАУ). Отметим, что речь идет об одном из ключевых алгоритмов, нередко являющимся «узким местом» сложного многоэтапного процесса численного моделирования, приводящим к наибольшим затратам памяти и времени счета. Решение таких СЛАУ с миллионами, десятками и сотнями миллионов элементов может выполняться методами разной природы. Так, например, решение СЛАУ $Ax = b$ можно представить как решение задачи минимизации невязки $||Ax - b||$, что

позволяет построить так называемые итерационные методы.[1] Суть этих методов заключается в выборе начального приближения с последующим итеративным вычислением невязки и выбором следующей точки до тех пор, пока ответ не окажется удовлетворительным или пока не будет исчерпано предварительно выбранное число итераций. Такие методы доминируют в настоящее время, имеют немало достоинств и активно используются, однако, имеют и некоторые недостатки. Так, число итераций, необходимых для достижения требуемой точности решения, существенно зависит от свойств матрицы. В «неудачных» случаях несмотря на все дополнительные модификации метод может сходиться очень медленно либо вовсе не достигать требуемой точности. Также итерационные методы применяются в комбинации с прямыми методами в качестве итерационного уточнения, что позволяет повысить точность прямого решателя СЛАУ.

Альтернативой итерационным методам являются *прямые методы*. Идея прямых методов заключается в том, чтобы представить разреженную матрицу системы в виде произведения двух или трех матриц, имеющих какую-либо простую структуру (например, треугольных или диагональных), с последующим решением простых систем[2]. Такой подход позволяет получить достаточно точное решение, однако в процессе разложения (факторизации) исходной разреженной матрицы могут получиться существенно более плотные матрицы. Это неизбежно не только на порядки увеличит затраты памяти, но и замедлит расчет. Для уменьшения затрат памяти и времени в прямых методах применяют предварительную процедуру – переупорядочение строк и столбцов матрицы, позволяющее уменьшить «заполнение» и хотя бы частично преодолеть указанную проблему. Весьма примечательно, что задача построения оптимальной перестановки является NP-трудной, тогда как исходный алгоритм факторизации матрицы решается за время $O(n^3)$. На первый взгляд это выглядит как парадокс: нам нужно решить крайне сложную задачу, чтобы ускорить решение более простой задачи. На самом деле для поиска перестановки строк и столбцов применяются эвристические алгоритмы (наиболее известные – метод минимальной степени и метод вложенных сечений), позволяющие получить приемлемый результат за небольшое время. Наряду с решением проблемы заполнения, в прямых методах работают над повышением точности

[1] Смотрите фундаментальный труд Ю.Саада (Y.Saad) [20], написанный по материалам лекций для студентов университета Миннесоты и перевод этой книги, выполненный Х.Д. Икрамовым [35, 36].

[2] Различаются два прямых метода: треугольно-ортогональное разложение или треугольное разложение исходной матрицы. Далее рассматривается треугольное разложение, которое существенно экономичнее треугольно-ортогонального разложения, хотя более чувствительно к ошибкам округления.

вычислений с плавающей запятой, используют вычисления в смешанной точности, когда трудоемкие расчеты выполняются в одинарной (или даже в половинной) точности, а более простые, но критически важные – в двойной, комбинируют прямые и итерационные методы, а также используют другие улучшения. Прямые методы также используются в итерационных методах, например, в алгебраическим многосеточным методе на самом грубом уровне самого малого размера, или в блочных методах Шварца и Якоби. В целом, прямые методы, наряду с итерационными являются признанным подходом к решению больших разреженных СЛАУ.

В данном пособии мы рассматриваем три распространенных программных пакета для прямого решения разреженных СЛАУ. Первый из них, oneMKL PARDISO, разработан компанией Intel и на протяжении многих лет входит в пакет инструментов Intel Parallel Studio (Intel oneAPI). Второй – MUMPS – разработан сотрудниками французских университетов и открыто распространяется в исходных кодах. Третий программный пакет – решатель USPARS [14, 34] – разработан в ООО НЦИТ Унипро (г. Новосибирск) и входит в состав пакета Логос Прочность [31] РФЯЦ-ВНИИЭФ. Все три пакета представляют широкий спектр методов и программных средств для прямого решения СЛАУ и оптимизированы для современных многопроцессорных многоядерных вычислительных систем (ПО Intel обычно лучше работает в системах на базе Intel-процессоров) на архитектуре x86-64. USPARS допускает применение на архитектурах ARM и отечественной архитектуре Эльбрус [27]. Основная цель пособия – научить использовать пакеты MUMPS и oneMKL PARDISO для решения нераспределённых и распределённых СЛАУ, дать информацию о пакете USPARS, рассказать о некоторых сложностях, которые могут встретиться начинающему разработчику, и также продемонстрировать методы оценки корректности результата и анализа производительности при работе на суперкомпьютере.

Отметим, что многопоточный oneMKL PARDISO и многопроцессорный, многопоточный MUMPS широко используются при решении стационарных задач прочности, модального (спектрального) и вибрационного анализа модулем Логос Прочность [24] и начинают использоваться в модуле Логос ЭМИ для моделирования воздействия электро-магнитного излучения на объекты. К настоящему моменту в библиотеку LParSol [23] и указанные модули пакета Логос, которые пока не распространяются, проходя стадию проверки, встроен пакет USPARS, который по возможностям и производительности не уступает Intel PARDISO [26]. В 2024 году ожидается создание кластерной версии USPARS, начатое в 2023 году. В настоящее

время USPARS оптимизирован для архитектуры x86-64, используя высокопроизводительные библиотеки OpenBLAS и Intel MKL BLAS, работоспособен на архитектуре ARM и Эльбрус – 6-го поколения, допускающей 16-ядерные процессоры в составе двухпроцессорного сервера.

1 Решение СЛАУ с разреженной матрицей

1.1 Математические основы

Рассмотрим систему линейных уравнений

$$Ax = b \quad (1)$$

в которой A – невырожденная, разреженная матрица размера $n \times n$, b – плотный вектор, x – вектор неизвестных. Если для решения системы (1) применяются прямые методы, то матрица системы раскладывается, или *факторизуется,* на произведение двух или трех матриц специального вида. В зависимости от типа матрицы применяются разные типы разложений. Так,

- если A – симметричная, положительно определенная, то используется *разложение Холецкого*:

$$A = LL^T \quad (2)$$

- если A – симметричная, неопределенная:

$$A = LDL^T \quad (3)$$

- если A – матрица общего вида:

$$A = LU \quad (4)$$

здесь L – нижняя треугольная матрица, U – верхняя треугольная матрица, D – диагональная матрица. Матрицы L и U называются *фактором* матрицы A. Отметим, что b и x – могут быть матрицами $n \times m$. В этом случае говорят о решении СЛАУ с многими правыми частями, где ряд дорогостоящих этапов решения СЛАУ выполняется один раз для всех правых частей. Существует ряд приложений, генерирующих такие СЛАУ, например, разрабатываемый модуль Логос ЭМИ.

Известно, что для невырожденной матрицы разложение (2), (3), или (4) существует и единственно. В ряде случаев для повышения численной устойчивости решения используют разложение (3) для симметричных положительно определенных матриц или разложение (4) для симметричных неопределенных матриц. Далее будем рассматривать матрицы общего вида и разложение (4).

После вычисления матриц L и U решение системы (1) сводится к последовательному решению двух систем с треугольными матрицами:

$$Ly = b, Ux = y. \quad (5)$$

Для этого применяется обратный ход метода Гаусса.

При численном решении СЛАУ с разреженной матрицей необходимо выполнять предобработку, которая заключается в масштабировании элементов матрицы и нахождении специальных перестановок. От успешности выполнения предобработки зависит эффективность дальнейшего

решения (затраты памяти, времени), а иногда и сама возможность получить решение.

Таким образом, решение СЛАУ (1) выполняется в несколько этапов:

1. Фаза анализа – предобработка матрицы, подготовка данных;
2. Численная фаза – численное разложение матрицы;
3. Обратный ход – решение треугольных систем (5);
4. Постобработка решения;
5. Анализ вычислительных ошибок.

При решении потока СЛАУ с одинаковым портретом матрицы этап 1, а для одинаковых значений элементов матрицы и этап 2 не обязательно выполнять для второй и далее СЛАУ. Этапы 3, 4, 5 выполняются для каждой СЛАУ.

Рассмотрим подробнее ключевые этапы решения.

1.1.1 Фаза анализа

Остановимся подробнее на предобработке матрицы.

В отличие от плотных матриц, разреженная матрица претерпевает *заполнение* в процессе факторизации. Это означает, что при выполнении операций исключения для строк матрицы A в факторе L возникают ненулевые элементы в позициях, где в нижней треугольной части матрицы A стояли нули. Как правило, заполнение фактора на порядки больше, чем у исходной матрицы. Например, при решении стационарных задач прочности модулем Логос Прочность фактор/факторы заполняются элементами, число которых превышает в 10, 20 и более раз исходное число (обычно около 80-ти) элементов матрицы. С ростом числа ненулевых элементов увеличиваются затраты памяти на их хранение, а также значительно увеличивается время, необходимое для факторизации матрицы. Для уменьшения заполнения фактора применяется симметричное переупорядочение строк и столбцов матрицы A. Нахождение точной матрицы перестановки[3], минимизирующей размер фактора при прямом решении СЛАУ, – NP-трудная задача. На практике используются эвристические методы, позволяющие получить перестановку, приемлемую по числу ненулевых элементов фактора. Как правило, для этого применяются модификации метода минимальной степени, или метода вложенных сечений, или сочетание методов вложенных сечений и минимальной степени. Подробнее о перестановках, минимизирующих заполнение, описано в [30, 7].

[3] Матрицей перестановки называется единичная матрица, в которой произвольно переставлены строки.

Другой особенностью численного решения СЛАУ является возможная потеря точности при выполнении операций с плавающей запятой. Например, если матрица плохо обусловлена и содержит элементы, различающиеся между собой на порядки, то при выполнении операций исключения могут быть потеряны значащие цифры решения. С увеличением числа операций ошибка будет только накапливаться. Для уменьшения этого эффекта в решателях применяется *масштабирование значений* элементов матрицы (*scaling*), а также нахождение перестановки строк, которая перемещает большие по модулю элементы к диагонали.

Таким образом, фаза анализа выполняется в несколько этапов:

1. Масштабирование, то есть нахождение матрицы

$$A_{scaled} = P_M D_r A D_c \tag{6}$$

где D_r – диагональная матрица масштабирования строк, D_c – диагональная матрица масштабирования столбцов, $P_M = (p_i)$ – несимметричная перестановка строк для максимизации модулей элементов на главной диагонали. Как правило [8], перестановка P_M максимизирует выражение (7) или (8):

$$\frac{|a_{jj}|}{\max\limits_{i \neq j} a_{ij}}, \text{ для всех } j,\ 1 \leq j \leq n \tag{7}$$

$$\left| \prod_{i=1}^{n} a_{i,\,p_i} \right| \tag{8}$$

2. Нахождение симметричной перестановки строк P, минимизирующей заполнение:

$$A_{pre} = P A_{scaled} P^T \tag{9}$$

Если матрица A была несимметричная, то перестановка ищется для симметризованной матрицы $A_{scaled} + {A_{scaled}}^T$.

Та же перестановка P применяется к вектору решения x и вектору правой части b:

$$x_{pre} = P D_c^{-1} x,\ b_{pre} = P P_M D_r b \tag{10}$$

3. Оценка объема памяти для хранения фактора L.
4. *Символьная факторизация* – определение расположения ненулевых элементов в матрицах L и U, построение вспомогательных структур данных, необходимых для выполнения численной фазы.

1.1.2 Численная факторизация. Выбор главного элемента

В результате работы фазы анализа на этапе численной факторизации переходят к решению системы (11):

$$A_{pre}x_{pre} = b_{pre} \quad (11)$$

Этот этап является наиболее трудоемким: для плотных матриц требует выполнения $O(2/3 \times n^3)$ операций, для разреженных матриц число операций зависит от структуры заполненности переупорядоченной матрицы. Как было сказано ранее, в зависимости от типа матрицы выполняется одно из разложений (2), (3) или (4). Для вычисления разреженных матриц L и U используются специальные алгоритмы, отличающиеся способом обхода исходной матрицы: левосторонний, правосторонний и мультифронтальный методы (*left-looking, right-looking, multifrontal method*). Алгоритмы разложения подробно описаны в [7, 33]. Подробный обзор литературы по алгоритмам факторизации приведен в [6].

Для улучшения численной стабильности гауссова исключения при решении системы с симметричной неопределенной или несимметричной матрицей A_{pre} применяют специальный прием для выбора главного элемента, который в англоязычной литературе называется *numerical pivoting*. При вычислении -го столбца матрицы L выполняется обмен строк матрицы A_{pre} так, чтобы наибольший по абсолютной величине элемент i-го столбца находился на главной диагонали (*partial pivoting*[4] [21]). Так, обмениваются строки i и k, если выполняется условие

$$|a_{k,i}| = \max_{l=1,\dots,n} |a_{l,i}| \quad (12)$$

Этот прием гарантирует, что элементы матрицы L будут ограничены по абсолютной величине числом 1.0 [29, стр. 50]. Использование этого приема для разреженных матриц влечет за собой некоторые трудности при реализации численной факторизации. Перестановка строк матрицы изменяет расположение ее ненулевых элементов, а значит, «обнуляет» результаты символьной фазы, может ухудшить заполнение матрицы и увеличить число вычислительных операций для дальнейшей факторизации, что приводит к необходимости использования динамических структур данных. Поэтому на практике для большого числа задач достаточно выбрать главный элемент $a_{i,i}$, не максимальный по абсолютной величине, а больший некоторого порогового значения *(partial threshold pivoting)* [9]:

$$|a_{i,i}| \geq u \times \max_{l=1,\dots,n} |a_{l,i}|, \quad (13)$$

где u – числовой параметр, $0.0 \leq u \leq 1.0$.

[4] Подразумевается, что full pivoting заключается в перестановке и строк, и столбцов при выборе главного элемента. Для большинства задач при выборе главного элемента достаточно применять только перестановку строк исходной матрицы, то есть выполнять partial pivoting.

Также применяется исключение со *статическим выбором главного элемента (static pivoting)* [12]: диагональные элементы, меньшие некоторого порога, заменяются на величину этого порога. Как правило, пороговое значение определяется формулой

$$\alpha = \varepsilon \times \|A_{pre}\|, \quad (14)$$

где $\|...\|$ – норма матрицы, ε – машинная точность.

Для симметричных неопределенных матриц в ходе разложения необходимо поддерживать симметрию фактора L. Поэтому при выборе главного элемента на диагональ перемещаются блоки размером 1×1 и 2×2, и в разложении $A = LDL^T$ матрица D состоит из диагональных блоков [2, 3].

1.1.3 Постобработка

На этапе постобработки решения выполняется обратная перестановка вектора x_{pre}. Также, при необходимости, выполняется *итерационное уточнение*, в котором найденное решение x корректируется для уменьшения численной ошибки решения. В частности, итерационное уточнение вызывается, если на этапе численной факторизации использовался статический выбор главного элемента [12]. Процедура итерационного уточнения заключается в следующем:

Пока не достигнута точность w или не выполнено максимальное число итераций K, повторить:

1. $r = b - Ax$;
2. Решить систему $A\Delta x = r$, используя приближенное разложение со статическим выбором главного элемента;
3. $x = x + \Delta x$.

На практике обычно достаточно выполнить 2–3 шага итерационного уточнения.

По умолчанию итерационное уточнение выполняется в той же точности, что и численная факторизация. Однако для прямого решателя, использующего числа с одинарной точностью, в качестве итерационного уточнения можно использовать итерационный решатель двойной точности, ограничив число итераций уточнения числом повторений или требуемой точностью. Это позволяет получить решение с точностью выше 10^{-7}, тратя в 2 раза меньше памяти и времени на факторизацию по сравнению с вычислением решения в двойной точности. В частности, такой прием используется в библиотеке LParSol для подключенных к ней прямых решателей USPARS, Intel PARDISO и MUMPS, наряду с встроенными в них итерационными уточнениями той же точности, что и прямое решение.

1.1.4 Анализ вычислительных ошибок

Поскольку вычисления выполняются в числах с плавающей запятой, то в ходе решения накапливаются ошибки округления и «усечения» полученных значений. Поэтому найденное численное решение $\hat{x}$ является точным решением не исходной, а возмущенной системы

$$(A + \delta A)\hat{x} = b + \delta b \tag{15}$$

Здесь δA – возмущение матрицы A, δb – возмущение вектора b, и существует такое малое число ε, что $|\delta a_{ij}| \le \varepsilon |a_{ij}|$, $|\delta b_i| \le \varepsilon |b_i|$, для всех $i, j = 1, \dots n$.

Для того, чтобы решение $\hat{x}$ можно было использовать в других расчетах, необходимо оценить его разницу с «правильным» решением x. Как правило, для этого оценивают норму вектора $\delta x = \hat{x} - x$ и норму *вектора невязки* $r = b - A\hat{x}$. Приведем некоторые оценки ошибки, согласно [29]. В оценках используется произвольная векторная норма и согласованная с ней матричная норма. Как правило, это нормы $\|\dots\|_1$, $\|\dots\|_2$, $\|\dots\|_\infty$.

Если известно или можно оценить число обусловленности матрицы A, $cond(A) = \|A^{-1}\| \cdot \|A\|$, то будут справедливы оценки (16)–(18):

$$\frac{\|\delta x\|}{\|\hat{x}\|} \le cond(A) \cdot \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\|\|\hat{x}\|} \right) \tag{16}$$

$$\frac{\|\delta x\|}{\|\hat{x}\|} \le \frac{cond(A)}{1 - cond(A)\dfrac{\|\delta A\|}{\|A\|}} \cdot \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right) \tag{17}$$

$$\|\delta x\| \le \|A^{-1}\| \cdot \|r\| = \|A^{-1}\| \cdot \|b - A\hat{x}\| \tag{18}$$

Эти оценки показывают связь между ошибкой δx и числом обусловленности $cond(A)$: чем оно больше, тем больше и ошибка δx. Для получения этих оценок нужно вычислить матрицу A^{-1}, что потребует $O(n^3)$ операций. На практике чаще вычисляется *обратная ошибка*, менее вычислительно трудоемкая.

Относительная обратная ошибка – это наименьшее число $\varepsilon > 0$, для которого существуют возмущения δA и δb, удовлетворяющие условиям $\|\delta A\| < \varepsilon \|A\|$, $\|\delta b\| < \varepsilon \|b\|$, $(A + \delta A)\hat{x} = b + \delta b$, определяемое по формуле

$$\varepsilon = \frac{\|b - A\hat{x}\|}{\|A\|\|\hat{x}\| + \|b\|} \tag{19}$$

Покомпонентной относительной обратной ошибкой называется величина

$$w = \max_{i=1\dots n} \frac{|b - A\hat{x}|_i}{(|A||\hat{x}| + |b|)_i} \tag{20}$$

Для вычисления оценки (19) можно использовать различное определение матричной и векторной нормы. Например, в решателе MUMPS относительная обратная ошибка считается по формуле (21):

$$\varepsilon = \frac{\|b - A\hat{x}\|_\infty}{\|A\|_\infty \|\hat{x}\|_\infty} \tag{21}$$

Подробные теоретические выводы ошибок прямого решения СЛАУ приведены в [25, 29, 33].

1.2 Форматы хранения разреженных матриц

Для эффективной обработки разреженных матриц применяются специальные форматы хранения, позволяющие экономить память и быстро выполнять различные матрично-векторные и матричные операции над матрицами с малым числом ненулевых элементов. Описание основных общепринятых форматов можно найти в [7, 28, 33]. Приведем здесь описание тех форматов, которые в дальнейшем будут использованы при запуске решателей.

Предположим, что исходная матрица состоит из n строк, содержит nz ненулевых элементов, для хранения индексов используются 4-байтовые целые числа (тип `int`), для хранения ненулевых значений – 8-байтовые числа с плавающей точкой (тип `double`), массивы нумеруются с 0.

Координатный формат (coordinate, COO) – самый простой способ хранения разреженной матрицы. В нем для хранения информации о ненулевых элементах используются три массива размера nz:

- `columns` – массив индексов номеров столбцов ненулевых элементов;
- `rows` – массив индексов номеров строк ненулевых элементов;
- `values` – массив значений ненулевых элементов.

Если значение матрицы $a_{i,j}$ хранится по некоторому индексу `k` в массиве `values`, `values[k]` = $a_{i,j}$, то `columns[k]=j`, `rows[k]=i`. Для хранения в этом формате требуется $16nz$ байт. Как правило, элементы матрицы хранятся в массивах упорядоченными по строкам или по столбцам. Рисунок 1, б показывает пример хранения матрицы в координатном формате.

Разреженный строчный формат (sparse compressed rows, CSR) – один из самых широко используемых форматов хранения. В нем ненулевые элементы хранятся подряд, упорядоченными по строкам. Для этого используются три массива:

- `columns` – массив индексов номеров столбцов ненулевых элементов, размер nz;
- `rowIndex` – массив индексов начала новой строки в массиве `columns`, размер $n + 1$;
- `values` – массив значений ненулевых элементов, размер nz.

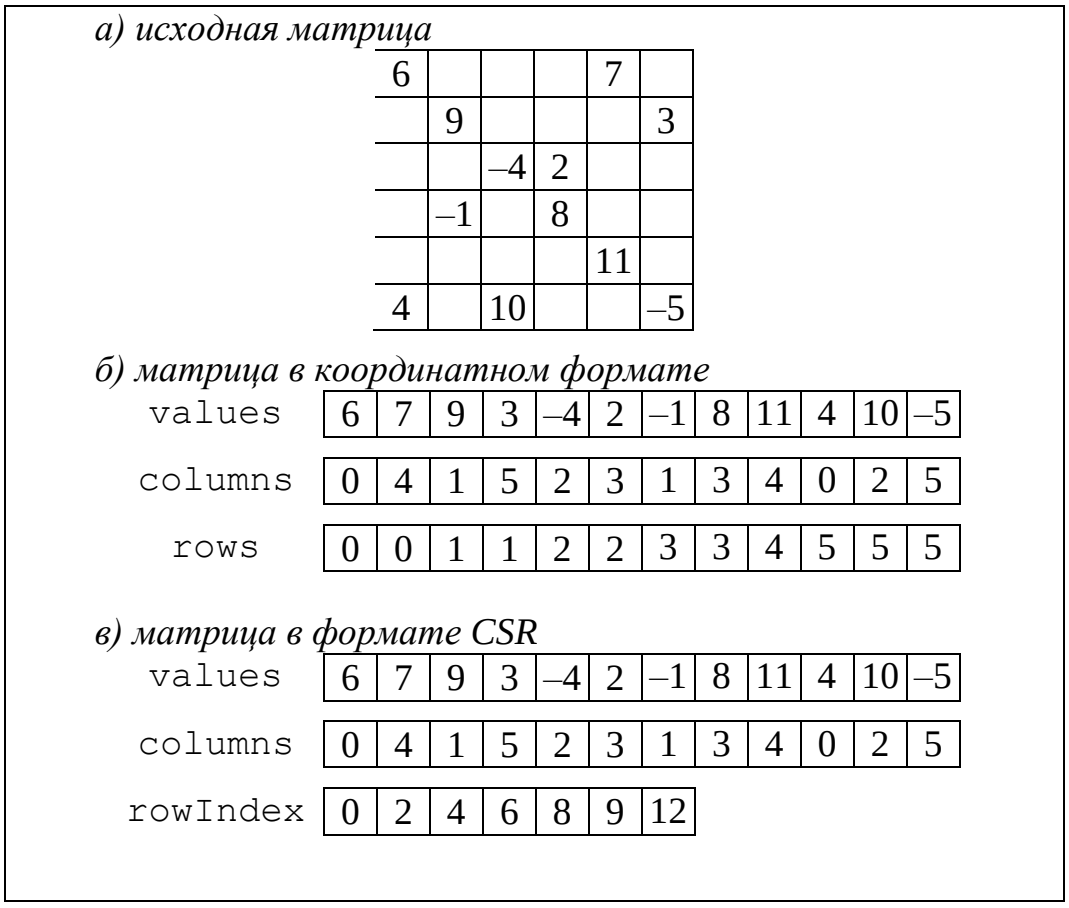

Рисунок 1. Пример хранения разреженной матрицы в форматах COO и CSR

В этом формате все ненулевые элементы строки матрицы `i` хранятся в массиве `values` по индексам с `rowIndex[i]` по `rowIndex[i+1]-1` включительно. Для единообразия и удобства обработки последней строки матрицы массив `rowIndex` хранит дополнительный, $(n + 1)$-вый элемент, `rowIndex[n]=nz`. Для хранения матрицы в этом формате потребуется $12nz + 4(n + 1)$ байт. Как правило, в библиотечных функциях предполагается, что все ненулевые элементы матрицы хранятся упорядоченно, в порядке обхода матрицы «слева направо, сверху вниз», то есть элементы массивов `values` и `column` для каждой строки упорядочены по возрастанию номеров столбцов.

а) исходная матрица

6				7	
	9				3
		–4	2		
	–1		8		
				11	
4		10			–5

б) матрица в распределенном формате COO

Процесс 1

values	6	7	9	3	–4	2
columns	0	4	1	5	2	3
rows	0	0	1	1	2	2

Процесс 2

values	–1	8	11	4	10	–5
columns	1	3	4	0	2	5
rows	3	3	4	5	5	5

в) матрица в распределенном формате CSR

Процесс 1

values	6	7	9	3	–4	2
columns	0	4	1	5	2	3
rowIndex	0	2	4	6		

Процесс 2

values	–1	8	11	4	10	–5
columns	1	3	4	0	2	5
rowIndex	0	2	3	6		

Рисунок 2. Пример хранения матрицы, распределенной между двумя процессами. а) исходная матрица, б) формат COO, в) формат CSR.

Отметим, что для симметричных матриц, как правило, хранятся только элементы нижнего или верхнего треугольника. Для некоторых алгоритмов более рационален разреженный столбцовый формат, аналогичный строчному.

Если матрица распределена между несколькими процессами, то, как правило, для задачи решения СЛАУ на каждом процессе хранится непрерывная группа строк («полоса») исходной матрицы. Дополнительно

хранится информация о распределении строк между процессами в виде массива или диапазона индексов строк, принадлежащих данному процессу.

В распределенном формате COO, который поддерживается в MUMPS, на каждом процессе хранятся массивы `column`, `rows`, `values` для элементов матрицы, относящихся к данному процессу (их называют *локальными элементами*), используется глобальная нумерация индексов. В распределенном формате CSR, который используется в Intel oneMKL, на каждом процессе массивы `column`, `rowIndex`, `values` также содержат информацию о локальных элементах, индексация в массиве `rowIndex` на каждом процессе начинается с 0. В некоторых библиотеках (в частности, Intel oneMKL и MUMPS) допускается перекрытие соседних полос, то есть элементы одной строки матрицы могут храниться на двух соседних процессах, а итоговое значение элементов на пересекающихся позициях получается в результате суммирования локальных значений (см. пример в [22], стр. 2629). Пример хранения матрицы на двух процессах показан ниже (рисунок 2, с. 17).

1.3 Программные пакеты для решения СЛАУ

Задача решения СЛАУ возникает при решении различных инженерных задач, моделировании в естественно-научных областях, поэтому разработка специализированного программного обеспечения ведется с середины 1980х годов и по настоящее время. С появлением суперкомпьютеров, развитием различных архитектур вычислительных систем (многоядерные центральные процессоры, графические процессоры и др.) академические исследователи и коммерческие компании предлагали новые алгоритмы, эффективно выполняющие решение СЛАУ с матрицами порядка десятков миллионов строк. В России и в мире разработано большое число так называемых *решателей* СЛАУ, входящих в состав инженерных пакетов, математических библиотек, а также в виде отдельных программ. Актуальный перечень академических разработок можно найти на странице [13]. Информация о решателях, описанных в данном пособии, дана ниже (таблица 1). Эти пакеты позволяют решить СЛАУ с вещественными или комплексными числами, одинарной или двойной точности, поддерживают различные виды разложений для разных типов матриц, позволяют выбрать алгоритмы переупорядочения матрицы, выполнить итерационное уточнение и анализ вычислительных ошибок. Они также включаются в библиотеки итерационных решателей hypre, PETSc (США), LParSol (Россия) и др. В

данном пособии будут показаны примеры использования пакетов Intel oneMKL PARDISO, MUMPS.

Таблица 1. Характеристики решателей разреженных СЛАУ

Название пакета	**Авторы**	**Тип параллелизма**	**Типы матриц**	**Форматы матриц**	**Ссылка**
oneMKL PARDISO, DSS	Коммерческий пакет, входит в Intel oneAPI MKL	Общая память, распределенная память	Симметричные, с симметричной структурой, общего вида	CSR, BCSR, variable BCSR	[10]
MUMPS	Академический пакет, CERFACS, ENS Lyon; Амстей, Дафф, Л`Экселлент	распределенная память	Симметричные, с симметричной структурой, общего вида	COO, elemental – аналог BCSS	[16]
USPARS	Коммерческий пакет, ООО НЦИТ Унипро, г. Новосибирск	Общая память	Симметричные положительно определённые, эрмитовы, симметричные, общего вида	CSR	[14]

Отметим, что отдельно решается вопрос о нахождении перестановки, минимизирующей размер фактора матрицы. В большинстве пакетов есть встроенные алгоритмы переупорядочения, интерфейс для передачи пользовательской перестановки, а также интерфейсы для использования отдельных специализированных пакетов. Для решения этой задачи наибольшее распространение получили пакеты METIS [11] и Scotch [4] и их аналоги для систем с распределенной памятью ParMETIS и PT-Scotch. В ННГУ для решения этой задачи разработаны библиотеки MORSy, PMORSy и DMORSy для работы в последовательном, многопоточном и многопроцессорном + многопоточном режиме, соответственно [19, 32]. Для пакета USPARS разработан тесно связанный с ним переупорядочиватель Atlant, а также могут использоваться METIS и PMORSy. В 2024 г. предполагается реализация версии Atlant и в целом версии пакета USPARS для работы в многопроцессорном + многопоточном режиме.

2 Программная реализация

Поставим задачу решения СЛАУ с разреженной матрицей общего вида продемонстрируем возможность решения этой задачи с использованием упомянутых выше программных комплексов. В разделе 2 будет описана общая структура программы для однократного решения СЛАУ, в разделе 3 – методика тестирования производительности решателя, в разделах 4 и 5 – решение СЛАУ с помощью решателей из библиотек Intel OneMKL и MUMPS соответственно. Для каждого решателя будет описан порядок сборки самой библиотеки, подключения ее к программе, дан пример кода вызова решателя и приведены результаты вычислительных экспериментов. Вопросы подбора числа процессов и потоков для получения лучшей производительности решателя будут обсуждаться в разделе 3.

Пусть дана СЛАУ вида (1), в которой A – невырожденная матрица общего вида с вещественными числами. Поскольку целью данного пособия является освоение программных инструментов, а не расчет реального явления, саму СЛАУ будем моделировать так, чтобы она была совместна и имела единственное решение. Матрицу системы A будем брать из широко используемой коллекции SuiteSparse [5], в которой загружены матрицы из разных предметных областей. Для того, чтобы получить совместную систему, зададим вектор неизвестных x, а затем найдем вектор правой части b, выполнив умножение: $b = Ax$. После этого выполним решение системы, «забыв» о том, что вектор x был известен, и оценим вычислительную ошибку решения.

Таким образом, программа для решения СЛАУ будет состоять из следующих модулей:

1. Чтение матрицы из текстового файла;
2. При необходимости, конвертация матрицы из координатного формата;
3. Задание векторов x, b;
4. Настройка параметров решателя;
5. Решение СЛАУ;
6. Анализ вычислительной ошибки;
7. Печать информации и освобождение памяти.

Создадим функцию `main()`, в которой реализована описанная выше схема решения, подходящая для решателей, ориентированных на системы с общей памятью (

листинг 1).

Листинг 1. Код основной функции main()

```
1.  // параметризация типа чисел с плавающей точкой
2.  #define FLOAT_TYPE double

3.  // argv[1] - файл с матрицей
4.  // argv[2] - тип матрицы
5.  int main(int argc, char **argv)
6.  {
7.  int n;               // число строк матрицы
8.  int nz;            // число ненулевых элементов
9.  int error;         // код ошибки
10. // структура матрицы в формате COO:
11. // RowCoo, ColIndex, Value
12. int* RowCoo;  // индексы строк ненулевых элементов
13. int* ColIndex;    // индексы столбцов ненулевых элементов
14. FLOAT_TYPE* Value;  // значения ненулевых элементов
15. // структура матрицы в формате CSR:
16. // RowIndex, ColIndex, Value
17. int* RowIndex;    // индексы начала строк
18. FLOAT_TYPE* b;    // вектор правой части
19. FLOAT_TYPE* x;    // вектор неизвестных
20. FLOAT_TYPE err;   // численная ошибка решения
21. int type;         // тип матрицы

22. if (argc < 2)
23. {
24.   printf("Args: [1] matrix_file [2] matrix_type\n");
25.   return 0;
26. }
27. type = atoi(argv[2]);

28. printf("matrix: %s\n", argv[1]);
29. // чтение матрицы
30. error = ReadMTXFile(argv[1], &n, &nz, &ColIndex, &RowCoo,
      &Value);
31. // создание массива RowIndex для конвертации в CSR
32. error = getRowIndex(n, nz, RowCoo, &RowIndex);

33. if (error != 0)
34. {
35.   printf("error in read file\n");
36.   return 1;
37. }
38. printf("n: %d\nnz: %d\n", n, nz);
39. printf("matrix type: %d\n", type);

40. b = (FLOAT_TYPE*) malloc(sizeof(FLOAT_TYPE) * n);
41. x = (FLOAT_TYPE*) malloc(sizeof(FLOAT_TYPE) * n);

42. // генерация совместной СЛАУ
```

```
43. GenerateSolution(ColIndex, RowIndex, Value, x, b, n);
44. // приведение индексации с 1
45. toOneBase(n, RowIndex, ColIndex);

46. // вызов решателя
47. Solve(type, n, nz, RowIndex, ColIndex, Value, b, x);

48. // приведение индексации с 0
49. toZeroBase(n, RowIndex, ColIndex);

50. // получение вычислительной ошибки решения СЛАУ
51. err = CheckError(n, ColIndex, RowIndex, Value, b, x);
52. printf("\nCalc error : (%e)\n", err);

53. // освобождение памяти
54. free(RowCoo);
55. free(RowIndex);
56. free(ColIndex);
57. free(Value);
58. free(x);
59. free(b);

60. return 0;
61. }
```

Отметим некоторые особенности кода.

1. Функция `main()` принимает в качестве параметров название файла с матрицей и тип матрицы. От типа матрицы зависит конфигурация параметров решателя и тип разложения. Для вещественных матриц в решателе MUMPS различается 3 типа матриц, в oneMKL PARDISO – 4 типа матриц.
2. Из коллекции SuiteSparse матрицу можно загрузить в текстовых форматах Matrix Market (*.mtx), MATLAB (*.mat) и Rutherford-Boeing (.rb). В данной работе использовался формат Matrix Market, его краткое описание дано в [15]. По сути, такой файл содержит описание разреженной матрицы в координатном формате. Как правило, файл содержит преамбулу, описывающую тип матрицы, авторов, предметную область или задачу, для которой построена матрица. Затем записана строка с характеристиками матрицы: число строк, столбцов, ненулевых элементов. Затем, построчно, индекс строки, столбца, значение каждого ненулевого элемента. Официальную библиотеку для чтения и записи *.mtx файлов можно найти на странице [1].
3. Перед вызовом решателя изменяется индексация массивов, поскольку оба решателя по умолчанию поддерживают нумерацию с 1.

4. Вызов решателя выполняется из функции `Solve()` со следующим прототипом:

```
void Solve(int type, int n, int nz, int * RowIndex,
  int *ColIndex, FLOAT_TYPE *Value, FLOAT_TYPE *b,
  FLOAT_TYPE * x);
```

Здесь

`int type` – тип матрицы;

`int n` – число строк матрицы;

`int nz` – число ненулевых элементов;

Матрица передается в формате CSR или COO:

`int* RowIndex` – индексы начала строк (CSR) или индексы строк ненулевых элементов (COO);

`int* ColIndex` – индексы столбцов ненулевых элементов;

`FLOAT_TYPE* Value` – значения ненулевых элементов;

`FLOAT_TYPE* b` – вектор правой части;

`FLOAT_TYPE* x` – вектор неизвестных;

`FLOAT_TYPE` – параметризация типа с плавающей точкой.

5. Оценка вычислительной ошибки выполняется в функции `CheckError()`. В данной работе использовалась оценка по формуле (21), которая вычисляется и в решателе MUMPS.

Далее, в разделах 4, 5 дадим реализацию функции `Solve()` для каждого решателя.

Полный код программы, разработанной для данного пособия, доступен по ссылке https://hpc-education.unn.ru/files/research/software/sparse_solvers.zip.

3 Методика тестирования

Для демонстрации основных элементов работы с программными комплексами решения СЛАУ и иллюстрации некоторых аспектов, связанных с настройкой и выбором подходящих параметров запуска проведем вычислительные эксперименты на СК «Лобачевский» со следующими характеристиками узлов: два 8-ядерных процессора Intel Sandy Bridge E5-2660 с частотой 2.20 ГГц, оперативная память 64 ГБ. Используем компилятор Intel oneAPI v.2023.0.0.

В дальнейшем в таблицах используем следующие обозначения:

- n – число строк матрицы,
- nz – число ненулевых элементов,
- nz/n^2 – заполненность матрицы,
- S – лучшее полученное ускорение относительно работы на одном ядре,
- K × M = число процессов × число потоков,
- # – программа завершила работу с ошибкой на этапе численной факторизации.

Для тестирования решателей выберем несимметричные матрицы из коллекции SuiteSparse (таблица 2):

Таблица 2. Характеристики тестовых матриц.

Название	**n**	**nz**	**nz/n^2**
ASIC_680k	682 862	3 871 773	8,30E-06
tmt_unsym	917 825	4 584 801	5,44E-06
vas_stokes_1M	1 090 664	34 767 207	2,92E-05
atmosmodl	1 489 752	10 319 760	4,65E-06
Transport	1 602 111	23 500 731	9,16E-06
ss	1 652 680	34 753 577	1,27E-05
memchip	2 707 524	14 810 202	2,02E-06

При тестировании решателя на кластерных системах следует внимательно отнестись к выбору соотношения числа процессов и потоков. Нерациональный выбор этих параметров может существенно увеличить время решения задачи. Для начинающих пользователей характерны как недооценка необходимых вычислительных ресурсов, так и ее переоценка, приводящая к аналогичным последствиям. Как правило, наилучшая конфигурация подбирается экспериментально, с учетом архитектуры вычислительных узлов, размеров решаемой задачи, а также особенностей параллельной реализации решателя. Полезными являются предварительные эксперименты, позволяющие оценить время решения задачи и подходящее

число узлов кластера, процессов и потоков ОС. Важно обратить внимание на запуск решателей на многосокетных системах, к которым, как правило, относятся современные суперкомпьютеры. В таких системах на одном узле может использоваться несколько многоядерных процессоров, работающих в рамках NUMA (non-unified memory access, неравномерный доступ к памяти) архитектуры. Это предполагает, что ядра одного процессора могут обращаться к памяти, локализованной на других процессорах того же узла, но за существенно большее время. Данный эффект можно учесть при разработке решателя, однако а) это не всегда возможно сделать в полной мере; б) не все разработчики уделяют этому достаточное внимание. Поэтому при использовании многопроцессорного узла кластера имеет смысл попробовать вариант запуска, в котором число процессов на узел соответствует числу процессоров на этом узле, а число потоков на каждый процесс – числу имеющихся ядер. Дополнительно имеет смысл проверять возможный выигрыш времени от удвоения числа используемых потоков в связи с тем, что на некоторых кластерах включен режим гипертредининга (hyper-threading), реализованный аппаратно во многих современных процессорах. Все это требует определенных предварительных экспериментов для выбора наиболее подходящих режимов работы. Кроме этого, практически всегда имеет смысл «привязывать» потоки ОС к ядрам процессора. В случае использования компилятора фирмы Intel для этого при запусках программы устанавливается переменная окружения:

```
export KMP_AFFINITY="granularity=fine,compact,1,0"
```

В данной работе один узел тестовой системы содержит два 8-ядерных процессора (всего 16 ядер). Это значит, что для решателей, ориентированных на общую память, следует проверить время работы решателя на одном процессоре (1, 4, 8 ядер) и на двух процессорах (16 ядер) одного вычислительного узла, по одному потоку на каждое ядро. Для решателей, ориентированных на системы с распределенной памятью, проведем эксперименты на одном и двух вычислительных узлах. При запусках на одном узле сначала проверим масштабируемость на одном процессоре (1 MPI процесс с 1, 4, 8 потоками OpenMP), затем – на двух процессорах (1 MPI процесс с 16 потоками OpenMP, 2 MPI процесса с 8 потоками OpenMP). После этого проверим, как влияет увеличение числа MPI процессов на время работы программы. Запуски с большим числом MPI-процессов на одном узле (8, 16 MPI процессов) подойдут для решателей, мало использующих многопоточный BLAS в MPI-реализации. Для задач большого порядка рационально использовать несколько вычислительных узлов. Поэтому

проверим работу решателей на двух узлах, используя от 2 до 32 MPI процессов.

Кроме вопросов производительности (и, разумеется, в первую очередь), важно проверить, что СЛАУ была решена корректно. Для этого можно реализовать вычисление ошибки самостоятельно или получить статистику от решателя. В дальнейших разделах вычислительная ошибка не приводится, поскольку для всех тестовых матриц относительная ошибка решения имела порядок на уровне машинной точности $\sim 10^{-15}$.

4 Пример решения СЛАУ помощью библиотеки Intel oneMKL

В библиотеке Intel oneMKL реализовано два прямых решателя СЛАУ: oneMKL PARDISO – решатель для систем с общей памятью, а также Parallel Direct Sparse Solver for Clusters Interface – решатель для систем с распределенной памятью. oneMKL PARDISO также имеет альтернативный интерфейс, Direct Sparse Solver Interface (DSS), в котором используются специальные типы данных для передачи матриц. Его оставим за рамками рассмотрения данной работы.

4.1 Вызов решателя Intel oneMKL PARDISO для общей памяти

4.1.1 Подключение решателя к программе

Для того, чтобы использовать решатель Intel oneMKL PARDISO, необходимо подключить к программе библиотеку Intel oneMKL. Наиболее простой способ – скомпилировать программу компилятором Intel oneAPI и указать ключ для подключения библиотеки oneMKL. Для этого:

1. Укажем компилятор и его параметры через переменные окружения,

```
CXX  = icpx  -std=c++11
CC  = icx
```

2. Добавим ключ

```
CFLAGS += -qmkl=parallel
```

3. В своем файле с вызовом решателя подключим заголовочный файл `"mkl_pardiso.h"` или `"mkl.h"`.

4.1.2 Интерфейс решателя

Основная функция решателя Intel oneMKL PARDISO имеет следующий прототип:

```
void pardiso(_MKL_DSS_HANDLE_t pt,
  const MKL_INT* maxfct, const MKL_INT* mnum,
  const MKL_INT* mtype, const MKL_INT* phase,
  const MKL_INT* n, const void* a, const MKL_INT* ia,
  const MKL_INT* ja, MKL_INT* perm,
  const MKL_INT* nrhs, MKL_INT* iparm,
  const MKL_INT* msglvl, void* b, void* x,
  MKL_INT* error);
```

Расшифруем параметры функции:

pt – дескриптор внутренней структуры данных;

maxfct – число факторов, которые надо хранить в памяти одновременно;

mnum – номер матрицы для численной факторизации;

mtype – тип матрицы;

phase – фаза решения;

n – размер матрицы;

a, ia, ja – массивы, хранящие матрицу. Для формата CSR a – массив значений, ia – массив индексов начала строк, ja – массив индексов столбцов ненулевых элементов;

perm – пользовательская перестановка для минимизации заполнения;

nrhs – число правых частей уравнения;

iparm – массив параметров решателя, 64 элемента;

msglvl – уровень печати информации;

b – вектор правой части;

x – вектор неизвестных;

error – код ошибки.

Как видно из прототипа, решатель позволяет решить серию СЛАУ с разными правыми частями или несколькими матрицами с разной или одинаковой структурой разреженности.

Параметр mtype, тип матрицы, определяет тип факторизации и набор параметров по умолчанию. Для действительных матриц выделяются 4 типа: симметричные по структуре, симметричные неопределенные, симметричные положительно определенные, несимметричные общего вида.

Параметр phase, фаза решения, определяет, какие из фаз решения СЛАУ необходимо выполнить. У пользователя есть возможность выполнить все фазы за один вызов решателя (phase=13) или выполнить только отдельную фазу. Последнее может быть полезно, например, при решении нескольких СЛАУ с одинаковым портретом матрицы. В этом случае решатель для фазы анализа можно вызвать один раз, а для фаз факторизации и решения – несколько раз для разных значений в матрице. Если предполагается решение нескольких СЛАУ с одной и той же матрицей, но разными правыми частями уравнения, то можно один раз вызвать решатель для фаз анализа и факторизации (phase=12), и несколько раз для фазы решения (phase = 22). Разделение решения по фазам значительно сократит время моделирования и затраты памяти на хранение факторов.

Приведем код функции `Solve()` для решателя Intel oneMKL PARDISO (листинг 2).

Листинг 2. Код вызова решателя Intel oneMKL PARDISO

```
#include "mkl_pardiso.h"
#include "mkl_types.h"

//------------------------------------------------------------//
//------------------------------------------------------------//
void Solve(int type, int n, int nnz, int * RowIndex, int
    *ColIndex, double *Value, double *b, double * res) {
MKL_INT mtype = type;    // тип матрицы
MKL_INT nrhs = 1;      // число правых частей уравнения
void* pt[64]; // дескриптор внутренней структуры данных
MKL_INT iparm[64]; // параметры решателя
// параметры решателя
MKL_INT maxfct, mnum, phase, error, msglvl;
// вспомогательные переменные
MKL_INT i, j;
double ddum;
MKL_INT idum;
double start, finish, time;

// создать резервную копию вектора b
memcpy(res, b, sizeof(FLOAT_TYPE)*n);

/* ------------------------------------------------------------ */
/* Установка параметров решателя */
/* ------------------------------------------------------------ */

// задание параметров по умолчанию
for (i = 0; i < 64; i++)
{
  iparm[i] = 0;
}
iparm[0] = 0;  /* использовать параметры по умолчанию */
iparm[1] = 3;  /* параллельный алгоритм переупорядочения
    для минимизации заполнения фактора */
iparm[5] = 0;  /* сохранить решение в x */
iparm[6] = 1;  /* выходной параметр: число итераций
    итерационного уточнения */
iparm[7] = 0;  /* две итерации итерационного уточнения
    (по умолчанию)*/
iparm[9] = 13; /* возмущение для статического выбора главного
    элемента (по умолчанию) */
iparm[10] = 1; /* масштабирование */
iparm[11] = 0; /* транспонированная матрица (нет) */
iparm[12] = 1; /* использовать перестановку для перемещения
    больших элементов к диагонали (да, по умолчанию) */
iparm[13] = 1; /* вывод: число возмущенных  */
```

```
37. iparm[14] = 0; /* вывод: пиковый объем памяти в символьной
    факторизации */
38. iparm[17] = -1;/* выводить число ненулевых элементов фактора
    (да) */
39. iparm[18] = -1;/* выводить число операций для факторизации
    (да) */
40. iparm[29] = 1; /* вывод: число нулевых или отрицательных
    вращений */

41. maxfct = 1;     /* число хранимых факторов */
42. mnum = 1;       /* номер матрицы для численной факторизации */
43. msglvl = 1;     /* вывод информации */
44. error = 0;      /* код ошибки */

45. // инициализация внутреннего дескриптора */
46. for (i = 0; i < 64; i++)
47. {
48.   pt[i] = 0;
49. }
50. printf("Pardiso start\n");
51. start = omp_get_wtime();

52. /* ------------------------------------------------------- */
53. /* решение системы уравнений */
54. /* ------------------------------------------------------- */
55. phase = 13;
56. pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, Value,
57. RowIndex, ColIndex, &idum, &nrhs, iparm, &msglvl, b, res,
58. &error);
59. if (error != 0)
60. {
61.   printf("\nERROR during solution: %d", error);
62.   exit(3);
63. }

64. /* ------------------------------------------------------- */
65. /* печать статистики */
66. /* ------------------------------------------------------- */
67. finish = omp_get_wtime();
68. time = finish - start;
69. printf("Pardiso finished\n");
70. printf("Pardiso Time: %f\n", time);
71. printf("Number of nonzeros in factors: %d\n", iparm[17]);
72. printf("Number of factorization MFLOPS: %d\n", iparm[18]);
73. printf("Number of zero or negative pivots: %d\n", iparm[29]);

74. /* ------------------------------------------------------- */
75. /* очистка памяти */
76. /* ------------------------------------------------------- */
77. phase = -1;
```

```
78. pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, &ddum,
    RowIndex, ColIndex, &idum, &nrhs, iparm, &msglvl, &b, &res,
      &error);
79. }
```

Здесь в строках 1–2 указаны необходимые заголовочные файлы. В функции `Solve()` в строках 27–40 устанавливаются параметры решателя по умолчанию для несимметричных матриц. В строках 41–42 значения переменных `maxfct`, `mnum` показывают, что будет выполняться решение СЛАУ для одной матрицы. В строках 55–58 вызывается сам решатель для матрицы, которая хранится в массивах `RowIndex`, `ColIndex`, `Value`. Поскольку повторных вызовов решателя не будет, достаточно вызвать все фазы решателя подряд с параметром `phase = 13`. В строках 77–78 выполняется вызов решателя для очистки внутренней памяти (`phase = -1`).

Обратим внимание на значения некоторых параметров.

Параметр `iparm[1]` определяет выбор алгоритма переупорядочения. По умолчанию устанавливается значение iparm[1]=1: использовать последовательный алгоритм из встроенной в oneMKL библиотеки METIS. Значение `iparm[1]=2` позволяет выбрать параллельный многопоточный алгоритм переупорядочения и сократить время фазы анализа.

Параметр `iparm[23]` отвечает за выбор алгоритма для параллельной численной факторизации. Для больших хорошо обусловленных систем целесообразно установить значение `iparm[23]=1` (двухуровневый параллельный алгоритм), который позволяет использовать многопоточные матричные операции и сократить время численной факторизации на многоядерных системах.

Параметры `iparm[10]`, `iparm[12]` определяют использование алгоритмов масштабирования и перемещения больших элементов к диагонали соответственно. Их следует устанавливать равными `1` для плохо обусловленных матриц общего вида и симметричных неопределенных матриц. Однако подключение этих опций накладывает ограничения на некоторые другие параметры решателя, в частности, значение `iparm[23]`.

Подробно все возможности решателя oneMKL PARDISO описаны в инструкции пользователя [22].

4.2 Результаты вычислительных экспериментов для Intel oneMKL PARDISO

Чтобы корректно запустить решатель на узле кластера, необходимо передать число доступных OpenMP потоков в библиотеку и привязать потоки к физическим ядрам процессора. Для этого установим переменные окружения:

```
export KMP_AFFINITY="granularity=fine,compact,1,0"
```

Далее, запустим приложение следующей командой (использовано 16 потоков OpenMP):

```
OMP_NUM_THREADS=16 ./pardiso <матрица> <тип>
```

Приведем результаты работы решателя Intel oneMKL PARDISO на одном вычислительном узле (таблица 3).

Таблица 3. Время работы решателя Intel oneMKL PARDISO (в секундах). Обозначения приведены в разделе 3. Лучший результат выделен серым цветом.

			Число потоков				
Матрица	**n, млн.**	**nz/n^2**	**1**	**4**	**8**	**16**	**S**
ASIC_680k	0,68	8,30E-06	18,41	9,82	7,91	6,46	2,85
tmt_unsym	0,92	5,44E-06	7,84	5,04	4,46	4,28	1,83
vas_stokes_1M	1,09	2,92E-05	1329,79	484,82	412,57	174,74	7,61
atmosmodl	1,49	4,65E-06	796,56	263,62	186,06	97,39	8,18
Transport	1,60	9,16E-06	717,09	236,01	182,52	80,84	8,87
ss	1,65	1,27E-05	3438,75	1066,78	821,36	430,26	7,99
memchip	2,71	2,02E-06	16,75	11,63	11,00	10,83	1,55

Из результатов экспериментов видно, что время работы решателя значительно зависит от заполнения матрицы (величина nz/n^2), а не от ее порядка. Для матриц с заполнением $\sim 10^{-5}$ (ss, vas_stokes_1M) время работы на один-два порядка больше, чем для матриц аналогичного размера, но с заполнением $\sim 10^{-6}$. При увеличении числа потоков время работы решателя сокращается. На матрицах порядка более 1 млн. строк с большим заполнением получено лучшее ускорение. Таким образом, при использовании решателя Intel oneMKL PARDISO рационально задействовать все вычислительные ядра узла.

4.3 Вызов решателя Intel oneMKL для распределенной памяти

Опишем интерфейс для вызова прямого решателя, ориентированного на кластерные системы, Parallel Direct Sparse Solver for Clusters Interface. Для краткости название решателя будем обозначать DSS_Cluster.

4.3.1 Подключение решателя к программе

Для того, чтобы скомпилировать программу, выполним следующие действия:

1. Укажем компилятор и его параметры через переменные окружения,

```
CXX  = micxx -cxx=icpx -std=c++11
CC   =  mpicc -cc=icx
```

2. Добавим ключи для подключения oneMKL и распределенного BLAS,

```
CFLAGS += -qmkl=parallel -lmkl_blacs_intelmpi_lp64
```

3. В своем файле с вызовом решателя подключим заголовочный файл `"mkl_cluster_sparse_solver.h"`.

4.3.2 Интерфейс решателя

Функция для вызова решателя, `cluster_sparse_solver()`, имеет интерфейс, аналогичный функции `pardiso()`, за исключением дополнительного параметра – коммуникатора MPI сообщений, и размера массива входных параметров `iparm` (65 элементов):

```
void cluster_sparse_solver(_MKL_DSS_HANDLE_t pt,
  const MKL_INT* maxfct, const MKL_INT* mnum,
  const MKL_INT* mtype, const MKL_INT* phase,
  const MKL_INT* n, const void* a, const MKL_INT* ia,
  const MKL_INT* ja, MKL_INT* perm, const MKL_INT* nrhs,
  MKL_INT * iparm, const MKL_INT* msglvl, void* b, void* x,
  const int* comm, MKL_INT* error);
```

Внесем изменения в функцию `main()`, чтобы подключить передачу данных между процессами при помощи MPI. Для этого добавим вызов функций `MPI_Init()`, `MPI_Finalize()` в начале и в конце функции.

Приведем измененный код функции `Solve()` для случая, когда матрица хранится целиком на одном процессе (`iparam[39]=0`, листинг 3).

Листинг 3. Код вызова решателя Intel oneMKL DSSC

```
1.  #include <mpi.h>
2.  #include "mkl_cluster_sparse_solver.h"
3.  #include "mkl_types.h"
```

```
4.  //------------------------------------------------------------//
5.  //------------------------------------------------------------//
6.  void Solve(int type, int n, int nz, int * RowIndex,
      int *ColIndex, double *Value, double *b, double * res) {
7.  /* ------------------------------------------------------------ */
8.  /* объявление переменных */
9.  /* аналогично строкам 6-18 Листинга 2
10. /* ------------------------------------------------------------ */
11. // внутри решатель использует Fortran MPI коммуникатор
12. // конвертируем MPI_COMM_WORLD
13. int comm = MPI_Comm_c2f(MPI_COMM_WORLD);

14. /* ------------------------------------------------------------ */
15. /* установка параметров решателя по умолчанию */
16. /* аналогично строкам 22-49 Листинга 2
17. /* ------------------------------------------------------------ */

18. /* ------------------------------------------------------------ */
19. /* решение системы уравнений */
20. /* ------------------------------------------------------------ */
21. phase = 13;
22. cluster_sparse_solver(pt, &maxfct, &mnum, &mtype, &phase,
      &n, Value, RowIndex, ColIndex, &idum, &nrhs, iparm,
      &msglvl, b, res, &comm, &error);

23. /* ------------------------------------------------------------ */
24. /* очистка памяти */
25. /* ------------------------------------------------------------ */
26. phase = -1;
27. cluster_sparse_solver(pt, &maxfct, &mnum, &mtype, &phase,
      &n, &ddum, RowIndex, ColIndex, &idum, &nrhs, iparm,
      &msglvl, b, res, &comm, &error);

28. }
```

Отметим некоторые отличия в значениях массива параметров `iparam` от тех, что передаются в функцию `pardiso()`.

Решатель DSS_Cluster поддерживает несколько способов распределения матрицы и векторов `x` и `b` между процессами, которые регулируются параметром `iparam[39]`. По умолчанию считается, что матрица и вектора хранятся на одном процессе с рангом 0 (`iparam[39]=0`). Если каждый процесс хранит только полосу матрицы, то, в зависимости от того, распределены ли вектора `x` и `b`, устанавливается значение `iparam[39]=1`, `2` или `3` и передаются границы полосы через `iparm[40]` и `iparm[41]`. Как и oneMKL PARDISO, решатель поддерживает форматы CSR и BCSR.

В решателе DSS_Cluster есть несколько встроенных параллельных переупорядочивателей, выбор которых определяется параметром `iparm[1]`. Значение `iparm[1]=10` позволяет использовать распределенную MPI-версию метода вложенных сечений и распределенную символьную факторизацию, что ускорит вычисления для матриц больших порядков. Однако эту опцию можно использовать с ограничениями на масштабирование `iparm[10]`, перестановку больших элементов к диагонали `iparm[12]`, формат матрицы `iparm[36]` (см. инструкцию пользователя).

4.4 Результаты вычислительных экспериментов

Чтобы корректно запустить решатель DSS_Cluster, используя параллелизм на уровне потоков OpenMP и процессов MPI, используем следующие команды:

1. Привяжем потоки к физическим ядрам процессора через переменные окружения,

```
export KMP_AFFINITY="granularity=fine,compact,1,0"
```

2. Запустим приложение. Например, используем два вычислительных узла, на каждом запустим по два MPI процесса с 8 OpenMP потоками (всего 4 процесса):

```
OMP_NUM_THREADS=8 mpiexec -np 4 -ppn 2
-hosts <названия узлов> ./dss_cluster <матрица> <тип>
```

Сравним результаты, полученные при различном соотношении числа процессов и потоков на одном и двух вычислительных узлах (таблица 4, таблица 5).

Таблица 4. Время работы решателя Intel oneMKL DSS_Cluster (в секундах) на одном вычислительном узле.
Обозначения приведены в разделе 3. Лучший результат выделен серым цветом.

	Один узел							
Матрица	**1×1**	**1×4**	**1×8**	**1×16**	**2×8**	**8×2**	**16×1**	**S**
ASIC_680k	18,46	9,81	7,90	6,46	5,74	6,45	8,15	3,22
tmt_unsym	7,87	5,05	4,51	4,28	2,73	3,68	7,34	2,89
vas_stokes_1M	1343,12	476,58	408,13	174,96	163,84	200,42	#	8,20
atmosmodl	783,92	257,10	182,86	97,26	85,67	142,12	120,02	9,15
Transport	714,57	232,55	180,61	80,84	64,42	79,40	98,93	11,09
ss	3149,03	1067,08	836,55	433,77	#	#	#	7,26
memchip	14,15	11,64	10,99	10,79	6,70	9,15	20,26	2,11

Как видно из таблицы 4, на одном узле для большинства матриц лучшие по времени результаты получены при соотношении 2 MPI процесса по 8 потоков, что соотносится с архитектурой вычислительного узла (два процессора по 8 ядер). При дальнейшем увеличении числа MPI-процессов на узел время работы программы возрастало. Это вызвано особенностями параллельного алгоритма, использованного в решателе: двухуровневый параллелизм, на этапе численной факторизации использованы многопоточные операции BLAS.

Аналогичная тенденция сохраняется и при запусках на двух вычислительных узлах (таблица 5). Для всех матриц, кроме одной, лучшие результаты получены при использовании 2 MPI-процессов и максимально доступного числа OpenMP потоков. При использовании большого числа MPI процессов на наиболее заполненных матрицах (ss, Transport, vas_stokes_1M) программа завершила работу с ошибкой, причина – нехватка памяти на этапе численной факторизации.

Таблица 5. Время работы решателя Intel oneMKL DSS_Cluster (в секундах)
на двух вычислительных узлах.
Обозначения приведены в разделе 3. Лучший результат выделен серым цветом.

	Два узла					
Матрица	**2×16**	**4×8**	**8×4**	**16×2**	**32×1**	**S**
ASIC_680k	4,63	4,81	4,92	5,63	7,56	3,98
tmt_unsym	2,91	3,27	3,15	4,08	5,63	2,71
vas_stokes_1M	105,90	113,36	129,65	138,75	#	12,68
atmosmodl	60,19	62,16	79,40	78,16	76,38	13,02
Transport	44,47	49,36	51,09	55,96	#	16,07
ss	224,08	216,94	#	#	#	14,52
memchip	7,40	8,56	7,08	9,53	29,82	1,91

5 Пример решения СЛАУ помощью библиотеки MUMPS

5.1 Сборка библиотеки

Решатель MUMPS предоставляется в открытых исходных кодах и доступен для скачивания по запросу со страницы [17]. Решатель разработан на языке программирования Fortran и имеет C-совместимую оболочку.

MUMPS не имеет встроенного параллельного переупорядочивателя, поэтому для наиболее эффективной работы решателя на системах с распределенной памятью рекомендуется предварительно установить переупорядочиватель ParMETIS или PT-Scotch. В данной работе использовался переупорядочиватель ParMETIS, доступный со страницы [18].

Для сборки ParMETIS выполним следующие действия:

1. При сборке библиотек мы будем использовать компилятор Intel(R) oneAPI DPC++/C++ Compiler 2023.0.0. Установим переменные окружения для использования компилятора Intel и gcc последней версии.[5]

```
1. . /common/software/intel/oneapi/setvars.sh intel64
2. module load gcc-9.5.0
```

2. Скачаем последнюю версию библиотек[6]

```
1. git clone https://github.com/KarypisLab/GKlib.git
2. git clone https://github.com/KarypisLab/METIS.git
3. git clone https://github.com/KarypisLab/ParMETIS.git
```

3. Выполняем последовательно конфигурацию библиотек и их сборку

```
1. cd GKlib/
2. make config cc=icx  openmp=set \
        CFLAGS=-D_POSIX_C_SOURCE=199309L
3. make install
4. cd ../METIS/
5. make config cc=icx
6. make install
7. cd ../ParMETIS/
8. I_MPI_CC=icx make config
9. I_MPI_CC=icx make install
```

По умолчанию собранные библиотеки устанавливаются в домашнюю директорию в папку «~/local».

```
1. ls ~/local/lib/
libgklib.a libmetis.a libparmetis.a
```

[5] Компилятор Intel использует ряд библиотек из gcc. Старая версия компилятора gcc приводит к ошибкам в линковке приложений.

[6] В строке 1 скачивается вспомогательная библиотека, используемая в реализации METIS и ParMETIS.

```
2. ls ~/local/include/
gk_arch.h gk_externs.h GKlib.h gk_mkblas.h gk_mkpqueue2.h
gk_mkrandom.h  gk_mkutils.h gk_ms_stat.h gk_proto.h  gk_struct.h
metis.h gk_defs.h  gk_getopt.h gk_macros.h  gk_mkmemory.h
gk_mkpqueue.h gk_mksort.h gk_ms_inttypes.h  gk_ms_stdint.h
gkregex.h gk_types.h   parmetis.h
```

Далее перейдем к сборке непосредственно решателя MUMPS. Сборка решателя выполняется в несколько этапов:

1. Как и в случае со сборкой пакетов METIS и ParMETIS, вначале установим переменные окружения для использования компилятора Intel и gcc последней версии.

```
1. . /common/software/intel/oneapi/setvars.sh intel64
2. module load gcc-9.5.0
```

2. В корне пакета MUMPS создадим конфигурационный файл Makefile.inc. В качестве основы можно использовать конфигурации, предложенные в папке Make.inc, например, Makefile.INTEL.PAR. При компиляции MUMPS использован следующий конфигурационный файл (листинг 4).

Листинг 4. Конфигурационный файл библиотеки MUMPS

```
1. ############################  Makefile.inc
2. #
3. #  This file is part of MUMPS 5.6.1, released
4. #  on Tue Jul 11 07:51:28 UTC 2023
5. #
6. ###################################
7.
8. LPORDDIR = $(topdir)/PORD/lib/
9. IPORD    = -I$(topdir)/PORD/include/
10. LPORD    = -L$(LPORDDIR) -lpord
11.
12. LMETISDIR = ${HOME}/local/lib
13. IMETIS    = -I${HOME}/local/include
14. LMETIS    = -L$(LMETISDIR) -lparmetis -lmetis
15.
16. ORDERINGSF  = -Dpord -Dmetis -Dparmetis
17. ORDERINGSC  = $(ORDERINGSF)
18.
19. LORDERINGS = $(LMETIS) $(LPORD) $(LSCOTCH)
20. IORDERINGSF = $(ISCOTCH)
21. IORDERINGSC = $(IMETIS) $(IPORD) $(ISCOTCH)
22.
23. PLAT    =
24. LIBEXT  = .a
25. LIBEXT_SHARED  = .so
26. SONAME = -soname
27. FPIC_OPT = -fPIC
28. OUTC    = -o
29. OUTF    = -o
30. RM = /bin/rm -f
```

```
31. CC = mpicc
32. FC = mpifc
33. FL = mpifc
34. AR = ar vr
35. RANLIB  = echo
36.
37. CDEFS   = -DAdd_
38.
39. OPTF    = -O -nofor-main -qopenmp -DGEMMT_AVAILABLE -
qmkl=parallel -lmkl_scalapack_lp64 -lmkl_cdft_core -
lmkl_blacs_intelmpi_lp64
40. OPTL    = -O -nofor-main -qopenmp -qmkl=parallel -
lmkl_scalapack_lp64 -lmkl_cdft_core -lmkl_blacs_intelmpi_lp64
41. OPTC    = -O -qopenmp -qmkl=cluster $(IORDERINGSC)
42.
43. INCS = $(INCPAR)
44. LIBS = $(LIBPAR)
45. ############################  Makefile.inc
```

В строках 8-14 указаны пути к библиотекам, осуществляющим перестановку матриц. Строка 16 определяет, какие переупорядочиватели будут использованы в сборке (PORD, METIS, ParMETIS). Строки 23-37 определяют параметры сборки и используемый компилятор. Строки 39-41 показывают, как можно подключить BLAS из библиотеки MKL.[7]

3. После конфигурации пакета сборка осуществляется командой

```
I_MPI_FC=ifx I_MPI_CC=icx make d
```

5.2 Вызов решателя MUMPS

5.2.1 Подключение решателя к программе

Подключение решателя MUMPS требует определенной аккуратности. Кроме непосредственно MUMPS, к программе необходимо подключить реализацию параллельного BLAS с технологией MPI и библиотеку для переупорядочения.

Приведем фрагмент файла Makefile.inc, в котором настроены переменные окружения для системы сборки (листинг 5):

Листинг 5. Фрагмент Makefile.inc для сборки программы с библиотекой MUMPS

```
1. CXX  = mpicxx -cxx=icpx -std=c++11
2. CC  =  mpicc -cc=icx
3. FC  =  mpifc -fc=ifx
4. LMETIS = -L${HOME}/local/lib -lparmetis -lmetis
5. IMUMPS = -I../MUMPS_5.6.1/include
```

[7] Кроме BLAS из Intel oneMKL можно использовать другие библиотеки, например, открытую многопоточную библиотеку OpenBLAS.

```
6.  LMUMPS = -L../MUMPS_5.6.1/lib \
             -ldmumps -lmumps_common -lpord
7.  DMUMPS = $(IMUMPS) $(LMUMPS) $(LMETIS)
8.  CFLAGS +=  -qmkl=parallel  -lmkl_blacs_intelmpi_lp64
```

Здесь определены следующие переменные окружения:

- компиляторы языков C и Fortran (строки 1–3),
- библиотеки METIS, ParMETIS и путь к ним (строка 4),
- путь к заголовочным файлам библиотеки MUMPS (строка 5).
- файлы библиотеки MUMPS (`dmumps`, `mumps_common`), файл библиотеки PORD и путь к ним (строка 6),
- общая сборка MUMPS (строка 7),
- ключи для использования BLAS из oneMKL (строка 8).

При линковке MUMPS к программе необходимо указать компилятор языка Fortran:

```
    $(FC) -nofor-main $(OPTC) -o $@ $? $(DMUMPS)
```

5.2.2 Интерфейс решателя

Для использования решателя MUMPS подключим заголовочный файл `"dmumps_c.h"`.

Передача параметров в решатель выполняется через структуры языка C, которые определены в файле `dmumps_struc.h`. Для типа `double` это структура `DMUMPS_STRUC_C`. Расшифруем значение некоторых ее полей:

`int sym` – тип матрицы (0 – несимметричная, 1 – симметричная положительно определенная, 2 – симметричная общего вида);

`int par` – можно ли использовать процесс 0 для численной факторизации (0 – нет, 1 – да);

`int job` – этап решения системы;

`int comm_fortran` – коммуникатор на языке `Fortran`;

`int icntl[60], real cntl[15]` – массивы параметров;

`int n` – размер матрицы;

`int nz; int64_t nnz;` – число ненулевых элементов;

`int *irn; int *jcn; double *a;` – централизованная матрица в координатном формате (индексы строк, столбцов, значения ненулевых элементов);

`int nz_loc; int *irn_loc; int *jcn_loc; double *a loc;` – распределенная матрица в координатном формате (локальное число

ненулевых элементов, глобальные индексы и значения для локальных ненулевых элементов);

`int *perm in;` – пользовательская перестановка;

`double *colsca; double *rowsca;` – масштабирование столбцов и строк;

`double *rhs` – вектор правой части. На выходе в него сохраняется найденное решение;

`int nrhs` – число векторов правой части;

`int info[80],infog[80];` – выходные параметры;

`double rinfo[40], rinfog[40];` – выходные параметры;

`int *sym_perm, *uns_perm;` – массивы, в которые на выходе сохраняется перестановка для минимизации заполнения и перестановка для удаления нулей с диагонали;

`int *mapping;` – массив, в который на выходе с фазы анализа сохраняется перенумерация строк матрицы для уменьшения коммуникации между процессами.

Все вызовы решателя выполняются через вызов одной функции, в которую передается указатель на переменную типа `DMUMPS_STRUC_C`:

```
void dmumps_c(DMUMPS_STRUC_C * dmumps_par );
```

Этап решения системы регулируется параметром `job`. До начала вычислений обязательно вызвать решатель с параметром `job=-1` на всех процессах, это этап инициализации. Затем можно вызвать фазы решения по отдельности (1 – фаза анализа, 2 – факторизация, 3 – фаза решения) или в комбинации (4 – анализ и факторизация, 5 – факторизация и решение, 6 – все фазы). Значение `job=-2` завершает работу решателя на всех процессах.

Приведем пример кода функции `Solve()` для случая, когда матрица хранится целиком на одном процессе, все стадии решения вызываются подряд.

```
1.  #include <mpi.h>
2.  #include <time.h>
3.  #include "dmumps_c.h"
4.  #define JOB_INIT -1 // номер этапа инициализации
5.  #define JOB_END -2  // номер этапа финализации
6.  #define USE_COMM_WORLD -987654 // коммуникатор по умолчанию
                                   для языка Fortran

7.  //-------------------------------------------------------//
8.  //-------------------------------------------------------//
```

```
9.  void Solve(int type, int n, int nnz, int * RowIndex,
      int *ColIndex, double *Value, double *b, double * res) {

10. DMUMPS_STRUC_C id; // структура для параметров MUMPS
11. double s, f;       // переменные для замеров времени
12. int myid, error, nProcs;
13. error = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
14. MPI_Comm_size(MPI_COMM_WORLD, &nProcs);

15. // резервная копия вектора b
16. memcpy(res, b, sizeof(FLOAT_TYPE)*n);

17. // инициализация решателя
18. id.job = JOB_INIT;
19. id.par = 1; // использовать хост для численной факторизации
20. id.sym = 0; // тип системы - несимметричная
21. id.comm_fortran = USE_COMM_WORLD;

22. dmumps_c(&id);

23. // передача матрицы и вектора правой части в решатель
24. id.n = n;
25. id.nnz = nnz;
26. id.irn = RowIndex;
27. id.jcn = ColIndex;
28. id.a = Value;
29. id.rhs = res;

30. // перенумерация параметров с Fortran на C
31. #define ICNTL(I) icntl[(I)-1]
32. #define CNTL(I) cntl[(I)-1]

33. id.ICNTL(18) = 0; // матрица хранится на хосте
34. id.ICNTL(28) = 2; // использовать параллельный алгоритм
                        переупорядочения
35. id.ICNTL(29) = 2; // использовать ParMETIS

36. // вызов MUMPS для решения СЛАУ
37. s = MPI_Wtime();
38. id.job=6;
39. dmumps_c(&id);
40. // проверка кода ошибки
41. if (id.infog[0]<0) {
42.   printf(" (PROC %d) ERROR RETURN: \tINFOG(1)=
                %d\n\t\t\t\tINFOG(2)= %d\n",
                myid, id.infog[0], id.infog[1]);
43.   error = 1;
44.   MPI_Abort(MPI_COMM_WORLD, 1);
45. }

46. // завершение работы решателя
47. id.job=JOB_END;
```

```
48. dmumps_c(&id);
49. f = MPI_Wtime();

50. // печать времени
51. if (myid == 0) {
52.   printf("\n MUMPS time : %.5lf\n", (double)(f - s));
53. }
54. }
```

Здесь в строках 1–3 подключаются необходимые заголовочные файлы, в строках 17–22 выполняется инициализация решателя, в строках 23–29 передается матрица, которая хранится целиком на всех процессах, в строках 33–35 указано, что будет использована параллельная фаза анализа с переупорядочивателем ParMETIS, в строках 36–39 вызываются все фазы решения СЛАУ, в строках 40–45 проверяется успешность решения, в строках 47–49 завершается работа решателя.

Обсудим некоторые параметры решателя.

Формат хранения матрицы (централизованный или распределенный) регулируется параметром `ICNTL(18)`. MUMPS позволяет передать матрицу, хранящуюся целиком на одном процессе (`ICNTL(18) = 0`) или распределенную (`ICNTL(18) = 1, 2, 3`). При этом отдельно выделяется случай, когда на фазе анализа вся структура матрицы доступна на одном процессе (хосте), а на этапе численной факторизации матрица распределена между всеми процессами.

Формат хранения вектора правой части регулируется параметром `ICNTL(20)`. Его можно хранить в плотном или разреженном формате, централизованно на одном процессе или распределенно. По умолчанию вектор правой части плотный и хранится на хосте (`ICNTL(20)=0`). Распределение вектора решения регулируется параметром `ICNTL(21)`.

Параметр `ICNTL(28)` отвечает за тип фазы анализа – последовательный или параллельный. От этого параметра зависит выбор алгоритма переупорядочения. Если фаза анализа последовательная, `ICNTL(28)=0`, то алгоритм переупорядочения указывается параметром `ICNTL(7)`, иначе – параметром `ICNTL(29)`. В решатель встроены интерфейсы для использования библиотек METIS/ParMETIS, Scotch/PT-Scotch.

Перестановка для перемещения больших элементов к диагонали устанавливается параметром `ICNTL(6)`, масштабирование элементов – параметром `ICNTL(8)`. Значения `CNTL(1) - CNTL(5)` регулируют

параметры вращений. Как правило, значения по умолчанию для этих параметров позволяет получить решение с достаточной точностью.

Для повышения точности решения можно подключить малоранговую аппроксимацию блоков матрицы (Block Low Rank Approximation), регулируемую вещественным параметром CNTL(7) $\geq 10^{-15}$ – порогом аппроксимации и рядом целочисленных параметров. Этот прием снижает потребление памяти и повышает точность результата, полученного итерационным уточнением.

Решатель возвращает пользователю подробную информацию о ходе решения через параметры INFO, RINFO, INFOG, RINFOG. В частности, можно получить статистику по объему использованной памяти, выполненных вращениях и др. Если было установлено значение `ICNTL(11)=1` или `ICNTL(11)=2`, то решатель выполнит анализ вычислительных ошибок и вернет их через массив `RINFOG`.

Подробно все возможности решателя MUMPS описаны в инструкции пользователя [16].

5.3 Результаты вычислительных экспериментов

Порядок запуска решателя MUMPS на вычислительной системе аналогичен запуску DSS_Cluster. Используем следующие команды:

1. Привяжем потоки к физическим ядрам процессора через переменные окружения,

```
export KMP_AFFINITY="granularity=fine,compact,1,0"
```

2. Запустим приложение, используя mpiexec. Например, задействуем два вычислительных узла, на каждом запустим по два MPI процесса с 10 OpenMP потоками (всего 4 процесса, 40 потоков):

```
OMP_NUM_THREADS=10 mpiexec -np 4 -ppn 2
-hosts <названия узлов> ./mumps <матрица> <тип>
```

Таблица 6 (стр. 45) показывает время работы решателя на одном вычислительном узле. Как видно из таблицы, при работе MUMPS на одном вычислительном узле лучшие результаты для каждой матрицы были получены при использовании 1 MPI процесса на ядро, что отличается от результатов, полученных DSS_Cluster. Однако систему с самой заполненной матрицей, ss, решить не удалось.

Таблица 7 (стр. 45) показывает время решения СЛАУ на двух вычислительных узлах. Для большинства матриц лучшие результаты были получены при работе 16 или 32 MPI процессов. Также видно, что благодаря

распределению данных между двум узлами удалось решить систему с матрицей ss.

Таблица 6. Время работы решателя MUMPS (в секундах)
на одном вычислительном узле.
Обозначения приведены в разделе 3. Лучший результат выделен серым цветом.

	Один узел							
Матрица	**1×1**	**1×4**	**1×8**	**1×16**	**2×8**	**8×2**	**16×1**	**S**
ASIC_680k	221,88	218,69	221,10	221,29	248,50	138,22	130,72	1,70
tmt_unsym	9,37	9,37	9,57	9,59	6,99	3,70	3,20	2,92
vas_stokes_1M	796,38	773,46	787,91	782,76	420,97	147,95	#	5,38
atmosmodl	545,44	532,89	541,37	541,97	254,30	90,24	57,88	9,42
Transport	559,31	545,16	557,21	555,09	292,21	104,03	65,24	8,57
ss	#	#	#	#	#	#	#	#
memchip	23,57	23,09	23,39	24,77	19,50	10,05	8,78	2,68

Таблица 7. Время работы решателя MUMPS (в секундах)
на двух вычислительных узлах.
Обозначения приведены в разделе 3. Лучший результат выделен серым цветом.

	Два узла					
Матрица	**2×16**	**4×8**	**8×4**	**16×2**	**32×1**	**S**
ASIC_680k	256,47	142,61	188,09	149,37	217,53	1,56
tmt_unsym	7,06	4,49	3,37	2,78	3,86	3,37
vas_stokes_1M	425,93	266,82	146,91	110,88	#	7,18
atmosmodl	264,48	156,40	86,36	56,69	46,13	11,82
Transport	295,90	175,53	106,33	68,47	51,53	10,85
ss	1110,42	660,25	#	#	#	#
memchip	19,82	13,82	11,75	8,02	8,16	2,94

Заключение

Решение больших разреженных СЛАУ – сложный многостадийный процесс. Изначальная постановка задачи выглядит достаточно просто, но первое впечатление, как это нередко бывает, обманчиво. На каждом этапе решения СЛАУ приходится преодолевать многочисленные трудности, в основе которых лежат совершенно разные причины: особенности и ограничения машинного представления чисел; объем доступной памяти и проблемы ее рационального использования; сложности организации параллельных вычислений. Данный перечень проблем может быть продолжен. Для решения разреженных СЛАУ в России и за рубежом разработано немало программных комплексов, позволяющих если не полностью, то в значительной степени преодолеть указанные выше проблемы. Программные комплексы (решатели разреженных СЛАУ) успешно применяются в различных задачах, которые требуют решения соответствующих систем с использованием высокопроизводительной вычислительной техники различной архитектуры. Примеры таких программных комплексов и некоторые сведения касательно их использования приведены в данной методической разработке. В заключение мы бы хотели отметить один очень важный факт: для успешного решения задачи иногда достаточно лишь разобраться с программным интерфейсом решателя СЛАУ и использовать его как «черный ящик», однако во многих случаях необходимо достаточно хорошо понимать математику и общую логику используемых методов, а также обладать навыками и опытом использования высокопроизводительной вычислительной техники. В этой связи мы призываем студентов, на которых ориентирована данная разработка, обращать внимание на всестороннее изучение современных численных методов, учиться планировать вычислительные эксперименты и делать выводы из их результатов. Полученные знания и навыки будут исключительно полезны в дальнейшей профессиональной карьере.

Литература

1. ANSI C library for Matrix Market I/O [Электронный ресурс]. – Режим доступа: https://math.nist.gov/MatrixMarket/mmio-c.html, свободный.
2. Bunch J. R., Kaufman L. Some stable methods for calculating inertia and solving symmetric linear systems //Mathematics of computation. – 1977. – Т. 31. – №. 137. – С. 163–179.
3. Bunch J. R., Parlett B. N. Direct methods for solving symmetric indefinite systems of linear equations //SIAM J. on Numerical Analysis. – 1971. – Т. 8. – №. 4. – С. 639–655.
4. Chevalier C., Pellegrini F. PT-SCOTCH: a tool for efficient parallel graph ordering // Parallel Computing. – 2008. – Vol. 34, No. 6–8. – P. 318–331.
5. Davis T. A., Hu Y. The University of Florida sparse matrix collection //ACM Transactions on Mathematical Software (TOMS). – 2011. – Т. 38. – №. 1. – С. 1–25.
6. Davis T. A., Rajamanickam S., Sid-Lakhdar W. M. A survey of direct methods for sparse linear systems // Acta Numerica. – 2016. – Т. 25. – С. 383–566.
7. Davis T.A. Direct methods for sparse linear systems. – Society for Industrial and Applied Mathematics, 2006. – 230 p.
8. Duff I. S., Koster J. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices //SIAM Journal on Matrix Analysis and Applications. – 1999. – Т. 20. – №. 4. – С. 889–901.
9. George A., Ng E. An implementation of Gaussian elimination with partial pivoting for sparse systems // SIAM J. on scientific and statistical computing. – 1985. – Т. 6. – №. 2. – С. 390–409.
10. Intel MKL PARDISO – Parallel Direct Sparse Solver Interface [Электронный ресурс]. – Режим доступа: https://www.intel.com/content/www/us/en/develop/documentation/ onemkl-developer-reference-c/top/sparse-solver-routines/onemkl-pardiso-parallel-direct-sparse-solver-iface.html, свободный.
11. Karipis, G. METIS. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 5.0. [Электронный ресурс]. // Technical report, University of Minnesota, Department of Computer Science and Engineering. – 2011. Режим доступа: http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf, свободный.

12. Li X.S., Demmel J.W. Making sparse Gaussian elimination scalable by static pivoting //SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing. – IEEE, 1998. – С. 34-34.
13. Li X.S. Direct Solvers for Sparse Matrices. [Электронный ресурс]. – Режим доступа: https://portal.nersc.gov/project/sparse/superlu/Sparse DirectSurvey .pdf, свободный.
14. Cherepanov M., Kostin V., Semenov A., Solovyev S. Efficient direct sparse solver for different processor architectures. //Тезисы международной конференции «Марчуковские научные чтения-2022», с. 99, Новосибирск, 2022.
15. Matrix Market Text File Formats [Электронный ресурс]. – Режим доступа: https://math.nist.gov/MatrixMarket/formats.html, свободный.
16. MUltifrontal Massively Parallel Solver (MUMPS 5.6.1) User's guide // Tech. rep. ENSEEINT-IRIT. – 2023. [Электронный ресурс]. – Режим доступа: https://mumps-solver.org/doc/userguide_5.6.1.pdf, свободный.
17. MUMPS Download Page [Электронный ресурс]. – Режим доступа: https://mumps-solver.org/index.php?page=dwnld, свободный.
18. ParMETIS Official Repository. [Электронный ресурс]. – Режим доступа: https://github.com/KarypisLab/ParMETIS, свободный.
19. Pirova A., Meyerov I., Kozinov E., Lebedev S. PMORSy: parallel sparse matrix ordering software for fill-in minimization // Optimization Methods and Software. – 2017. – Vol. 32. – No. 2. – P. 274–289.
20. Saad Y. Iterative Methods for Sparse Linear Systems. 2nd edition. – NY: PWS Publish, 2000. – 460 p.
21. Sherman A. H. Algorithms for sparse Gaussian elimination with partial pivoting //ACM Transactions on Mathematical Software (TOMS). – 1978. – Т. 4. – №. 4. – С. 330–338.
22. User Developer Reference for Intel® oneAPI Math Kernel Library for C. [Электронный ресурс]. – Режим доступа: https://www.intel.com/ content/www/us/en/docs/onemkl/developer-reference-c/2023-2/overview .html, свободный
23. Алейников А.Ю., Барабанов Р.А., Бартенев Ю.Г. и др. Параллельные решатели СЛАУ в пакетах программ Российского Федерального Ядерного Центра – Всероссийского Научно-Исследовательского Института Экспериментальной Физики // Вестник ПНИПУ. Аэрокосмическая техника. – 2016. – Вып. 47. – С. 73–92.
24. Александрова О.Л., Барабанов Р.А., Дьянов Д.Ю. и др. Пакет программ Логос. Конечно-элементная методика расчета задач

статической прочности конструкций с учетом эффектов физической и геометрической нелинейности // Вопросы атомной науки и техники, сер.: мат. моделирование физ. процессов. – Саров, 2014. – Вып. 3. – С. 3–17.
25. Белов С. А., Золотых Н. Ю. Численные методы линейной алгебры. Лабораторный практикум. – Н. Новгород, Изд-во Нижегородского госуниверситета, 2005. – 264 с.
26. Быков Д.О., Соловьёв С.А., Костин В.И. и др. Программный пакет USPARS для больших разреженных систем» // I Всероссийская школа-семинар Национального центра физики и математики для студентов, аспирантов, молодых ученых и специалистов «Центр исследования архитектур суперкомпьютеров». – Саров, 2023.
27. Волин В.С., Волконский В.Ю., Груздов Ф.А. и др. Микропроцессоры и вычислительные комплексы семейства «Эльбрус». Сер. «Учебное Пособие». – СПб.: Питер, 2013. – 272 с.
28. Гергель В.П., Баркалов К.А., Мееров И.Б., Сысоев А.В., и др. Параллельные вычисления. Технологии и численные методы: Учебное пособие в 4 томах. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013.
29. Деммель Д. Вычислительная линейная алгебра: теория и приложения. – Мир, 2001. – 430 с.
30. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984. – 333 с.
31. Логос Прочность. [Электронный ресурс]. – Режим доступа: http://logos.vniief.ru/products/strength/, свободный
32. Пирова А.Ю. Гибридный MPI + OpenMP алгоритм переупорядочения симметричных разреженных матриц для систем с распределенной памятью // Математическое моделирование и суперкомпьютерные технологии. Труды XXI Международной конференции (Н. Новгород, 22–26 ноября 2021 г.) / Под ред. проф. Д.В. Баландина – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2021. – 423 с. – С. 268–273.
33. Писсанецки С. Технология разреженных матриц. – М.: Мир,1988. – 410 с.
34. Решатель USPARS. [Электронный ресурс]. – Режим доступа: http://unipro.ru/uspars/, свободный.

35. Саад Ю. Итерационные методы для разреженных линейных систем: Учеб. пособие. – в 2-х томах. Том 1 // Пер. с англ.: Х.Д. Икрамов – М.: Издательство Московского университета, 2013. – 346 с.
36. Саад Ю. Итерационные методы для разреженных линейных систем: Учеб. пособие. – в 2-х томах. Том 2 // Пер. с англ.: Х.Д. Икрамов – М.: Издательство Московского университета, 2013. – 328 с.

Юрий Германович **Бартенев**,
Евгений Александрович **Козинов**,
Иосиф Борисович **Мееров**,
Анна Юрьевна **Пирова**

РЕШЕНИЕ СЛАУ С РАЗРЕЖЕННОЙ МАТРИЦЕЙ ПРЯМЫМИ МЕТОДАМИ НА СУПЕРКОМПЬЮТЕРЕ

Учебное пособие

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
603022, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать 02.10.2023 г. Формат 60.84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л. 3. Уч.-изд. л. 3,8. Заказ № 312. Тираж 100 экз.

Отпечатано в типографии Нижегородского госуниверситета
им. Н.И. Лобачевского.
603000, г. Нижний Новгород, ул. Большая Покровская, 37