УДК 519.6

ПАРАЛЛЕЛЬНЫЙ ПРЕДОБУСЛОВЛИВАТЕЛЬ SSOR ДЛЯ РЕШЕНИЯ ЗАДАЧ ЭЛЕКТРОМАГНЕТИЗМА В ЧАСТОТНОЙ ОБЛАСТИ

Д. С. Бутюгин[1]

Рассматриваются подходы к распараллеливанию предобусловливателя симметричной последовательной верхней релаксации (SSOR) в модификации Айзенштата, используемый при итерационном решении линейных систем, возникающих в результате аппроксимации соответствующих вариационных задач электромагнетизма. Распараллеливание предобусловливателя SSOR основано на декомпозиции расчетной области, в том числе и алгебраической, с совместным переупорядочиванием матрицы. Отдельное внимание уделено обеспечению высокой производительности на NUMA-архитектурах. Результаты проведенной серии численных экспериментов демонстрируют производительность и масштабируемость представленных алгоритмов. Работа выполнена при финансовой поддержке РФФИ (код проекта 08–01–00526). Статья рекомендована к публикации Программным комитетом Международной научной конференции “Параллельные вычислительные технологии” (ПАВТ–2011; http://agora.guru.ru/pavt2011).

Ключевые слова: предобусловливатели, параллельные алгоритмы, декомпозиция области, NUMA-архитектуры.

1. Введение. Задача моделирования трехмерных электромагнитных полей в частотной области возникает при расчетах различных волновых устройств, таких как волноводы, антенны, микроволновые устройства и др. Данная задача является вычислительно сложной, поскольку требует решения систем линейных алгебраических уравнений высоких порядков (10^6–10^7). Особый интерес вызывает возможность эффективного использования итерационных алгоритмов для решения этих систем. Однако это требует использования эффективных предобусловливателей, что в ряде случаев вызывает трудности задействования вычислительных мощностей на многоядерных системах, поскольку для многих предобусловливателей сложно обеспечить необходимую степень масштабируемости.

В настоящей статье рассматриваются подходы к распараллеливанию и оптимизации предобусловливателя симметричной последовательной верхней релаксации (SSOR — Symmetric Successive Overrelaxation) в модификации Айзенштата [1], используемого совместно с итерационными методами сопряженных градиентов (CG — Conjugate Gradient), сопряженных невязок (CR — Conjugate Residual) и другими при решении ряда задач электромагнетизма. Распараллеливание предобусловливателя SSOR основано на декомпозиции расчетной области с совместным переупорядочиванием матрицы. Кроме того, рассматривается алгебраическая декомпозиция области при использовании библиотеки METIS [2]. Отдельное внимание при реализации матрично-векторных операций уделено обеспечению высокой производительности на NUMA-архитектурах (Non-Uniformed Memory Architecture).

Структура данной работы следующая. В разделе 2 описывается постановка задачи и используемые итерационные алгоритмы. Раздел 3 содержит описание подхода к распараллеливанию предобусловливателя SSOR, а раздел 4 — некоторые полезные оптимизации предлагаемого алгоритма. В разделе 5 приведены результаты численных экспериментов. Наконец, в последнем разделе обсуждаются полученные результаты, а также рассматриваются возможные пути дальнейших исследований.

2. Постановка задачи. Решение разреженных систем линейных алгебраических уравнений

$$Ax = b \tag{1}$$

итерационными методами в подпространствах Крылова требует выполнения умножений матрицы A на различные векторы. Ниже приведены схемы вычислений в алгоритмах сопряженных градиентов (случай

[1] Институт вычислительной математики и математической геофизики СО РАН, просп. Лаврентьева, 6, 630090, Новосибирск; инженер, e-mail: dm.butyugin@gmail.com

$q = 0$) и сопряженных невязок ($q = 1$):

$$p_0 = r_0 = f - Au_0, \quad \alpha_j = \frac{\langle A^q r_j, r_j \rangle}{\langle A^q p_j, Ap_j \rangle}, \quad u_{j+1} = u_j + \alpha_j p_j,$$

$$r_{j+1} = r_j - \alpha_j A p_j, \quad \beta_j = \frac{\langle A^q r_{j+1}, r_{j+1} \rangle}{\langle A^q r_j, r_j \rangle}, \quad p_{j+1} = r_{j+1} + \beta_j p_j,$$

где $\langle u, v \rangle$ — скалярное произведение векторов u и v.

Для увеличения производительности итерационных решателей и ускорения их сходимости, как правило, применяют одно из следующих предобусловливаний системы:

$$B^{-1}Ax = B^{-1}b, \quad B_L^{-1}AB_U^{-1}B_U x = B_L^{-1}b, \quad AB^{-1}Bx = b.$$

Здесь $B_L B_U = B$, а B — предобусловливающая матрица, "близкая" в некотором смысле к исходной матрице A. В рамках данной работы рассматривается предобусловливатель симметричной последовательной верхней релаксации:

$$B_L = (L + G)G^{-1/2}, \quad B_U = G^{-1/2}\big(L^{\mathrm{T}} + G\big), \quad G = \frac{1}{\omega} D.$$

Здесь $A = L + D + L^{\mathrm{T}}$ — симметричная матрица системы (1), а $0 < \omega < 2$ — параметр релаксации. Известна модификация этого предобусловливателя, позволяющая свести умножение на исходную матрицу и решение двух приведенных треугольных систем уравнений к решению двух вспомогательных треугольных систем [1]: умножение на матрицу $\overline{A} = B_L^{-1}AB_U^{-1}$ вычисляется как

$$y = \big(I + \overline{L}^{\,\mathrm{T}}\big)^{-1}x, \quad \overline{A}x = \big(I + \overline{L}\big)^{-1}\Big(x - \big(2I - \overline{D}\big)y\Big) + y,$$

где $\overline{L} = G^{-1/2}LG^{-1/2}$, $\overline{D} = G^{-1/2}DG^{-1/2} = \omega I$ и I — единичная матрица.

В то время как распараллеливание векторно-векторных операций в данных итерационных решателях представляет собой довольно простую задачу, распараллеливание предобусловливателя SSOR — как обычного, так и в модификации Айзенштата — сопряжено с определенными трудностями. Сложность эта связана с необходимостью решения треугольных систем, имеющих портрет исходной матрицы. Матрицы, получающиеся в результате конечно-элементных и конечно-разностных аппроксимаций дифференциальных уравнений, имеют обычно, с одной стороны, небольшое число элементов в каждой из строк и, с другой стороны, сложные зависимости между переменными. Все это не позволяет реализовать эффективное распараллеливание в общем случае. Можно отметить, что большинство пакетов итерационных решателей не поддерживают параллельное предобусловливание с помощью SSOR (например, PETSc и др.), а те пакеты, которые имеют распараллеленное решение треугольных систем (такие как Intel MKL — Math Kernel Library), как правило показывают невысокую масштабируемость на реальных задачах. В рамках данной работы предлагаются подходы и ряд оптимизаций, позволяющие получить масштабируемый SSOR на задачах электромагнетизма.

3. Параллелизация SSOR. Основная проблема при разработке масштабируемого предобусловливателя SSOR связана с тем, что при решении треугольных систем уравнений имеются нетривиальные зависимости по данным. Например, нижнетреугольной системе с некоторой матрицей L можно сопоставить граф, вершинами которого являются столбцы матрицы, а ребрами — ненулевые элементы матрицы, так, что граф приобретает вид $G = (V, E)$, где множество вершин $V = \{1, 2, \ldots, n\}$, n — порядок матрицы A, а множество ребер $E = \{(u, v) : u \in V, v \in V, [L]_{u,v} \neq 0\}$. Тогда граф G является ориентированным и показывает зависимости между элементами вектора неизвестных x при решении системы $Lx = y$. Транспонированный граф G^{T} указывает зависимости между элементами вектора неизвестных x при решении верхнетреугольной системы $L^{\mathrm{T}}x = y$.

Для того чтобы устранить трудности, связанные с произвольной структурой графа G и невозможностью в полной мере использовать преимущества мультипроцессорных систем при решении треугольных систем, можно воспользоваться идеей алгоритма вложенных сечений графа G [2]. Алгоритм вложенных сечений основан на следующей процедуре: среди вершин V графа G ищется разделяющее множество $S \subset V$, предпочтительно небольшого по сравнению с V размера, такое, что оставшиеся вершины графа делятся на два множества V_1 и V_2, причем $V_1 \bigcap V_2 = \emptyset$ и $\forall u \in V_1$, $v \in V_2 : (u, v) \notin E, (v, u) \notin E$. Далее процедура повторяется для каждой из двух полученных частей графа до тех пор, пока либо не будет достигнут малый размер получившихся частей, либо пока не будет получено достаточное количество частей. Далее осуществляется перенумерация вершин графа по следующему принципу: сначала рекурсивно нумеруются вершины в V_1, затем — в V_2, а в последнюю очередь нумеруются вершины S.

Алгоритм вложенных сечений графа в данной работе предлагается использовать следующим образом. Рассмотрим граф G для нижнетреугольной части L матрицы A. Построим многоуровневую декомпозицию матрицы G. Пусть $\sigma(i)$ — номер вершины i в новой нумерации. Ниже представлена матрица перестановок P, где e_j — вектор длины n, все компоненты которого равны нулю за исключением компоненты j, значение которой равно единице. Тогда матрица $P^{\mathrm{T}}AP$ будет иметь следующую структуру:

$$P = \begin{bmatrix} e_{\sigma(1)} \\ e_{\sigma(2)} \\ \cdots \\ e_{\sigma(n)} \end{bmatrix}, \quad P^{\mathrm{T}}AP = \begin{bmatrix} D_1 & 0 & B_1^{\mathrm{T}} \\ 0 & D_2 & B_2^{\mathrm{T}} \\ B_1 & B_2 & D_3 \end{bmatrix}.$$

Здесь блоки D_1 и D_2 соответствуют множествам вершин V_1 и V_2, блок D_3 — разделяющему множеству S, а подматрицы B_1 и B_2 — ребрам между V_1 и S и V_2 и S соответственно.

Теперь вместо решения системы (1) можно решать систему

$$\widehat{A}\widehat{x} = \widehat{b}, \quad \widehat{A} = P^{\mathrm{T}}AP, \quad \widehat{x} = P^{\mathrm{T}}x, \quad \widehat{b} = P^{\mathrm{T}}b.$$

Такой алгоритм, как правило, используется для получения системы, у которой число ненулевых элементов в LU-разложении меньше, чем в исходной системе. Для этого необходимо на нижнем уровне алгоритма вложенных сечений использовать некоторую процедуру, которая минимизирует заполнение матриц L и U, например алгоритм минимальных степеней (AMD — Approximate Minimum Degree) [2]. Однако для нас интерес представляет структура матрицы $\widehat{A}$ и, в частности, вид матрицы системы $\bigl(I+\overline{L}\bigr)u = f$, требующей решения в предобусловливателе SSOR с модификацией Айзенштата. Ее структура имеет вид

$$\begin{bmatrix} I+\overline{L}_{D_1} & 0 & 0 \\ 0 & I+\overline{L}_{D_2} & 0 \\ \overline{B}_1 & \overline{B}_2 & I+\overline{L}_{D_3} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}.$$

Отсюда видно, что вычисление коэффициентов u_1 и u_2 можно производить независимо, а коэффициенты u_3 определять уже после вычисления u_1 и u_2. Аналогично в случае системы с матрицей $\bigl(I+\overline{L}^{\mathrm{T}}\bigr)$: сначала определяются коэффициенты u_3, а затем u_1 и u_2 могут быть определены независимо друг от друга.

Один из вариантов алгоритма вложенных сечений реализован в библиотеке METIS [2]. Однако следует отметить, что алгоритм вложенных сечений требует довольно существенных вычислительных ресурсов. Применение его может быть оправдано, например, в случае многоуровневых итерационных процессов, когда внутренние итерации обращают один и тот же предобусловливатель. В этом случае алгоритм вложенных сечений необходимо запустить только один раз в начале итераций, после чего полученное разбиение можно использовать многократно при решении предобусловливающей системы с различными правыми частями. Примером такого многоуровневого процесса может служить предобусловленное решение системы с седловой точкой, предложенное в [3]. В этой работе решается система вида

$$\begin{bmatrix} A-k_0^2M & B^{\mathrm{T}} \\ B & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \quad \text{с предобусловливателем} \quad \mathcal{P}_{M,L} = \begin{bmatrix} A+\bigl(1-k_0^2\bigr)M & 0 \\ 0 & L \end{bmatrix}.$$

Описанную технику можно применить для решения систем с матрицами L и $\mathcal{P}_M = A + \bigl(1-k_0^2\bigr)M$.

Если алгоритм вложенных сечений неприменим по тем или иным причинам, то можно предложить альтернативный подход для конечно-элементных схем. Базисные функции конечно-элементных подпространств пространства H^{rot} соответствуют геометрическим объектам конечно-элементной сетки — узлам, ребрам, граням и объемам [4] (тетраэдрам, шестигранникам, призмам и т.д. в зависимости от типа сетки). В таком случае разделяющее множество имеет очевидную геометрическую интерпретацию — оно соответствует элементам сетки, лежащим на полигональной поверхности, делящей расчетную область на два непересекающихся объема. Тогда возможно реализовать альтернативные, чисто геометрические алгоритмы декомпозиции матрицы на блоки. Например, можно на каждом этапе в текущей расчетной области проводить плоскость по границам элементов сетки, делящую ее на две примерно равные подобласти, причем так, чтобы сечение области плоскостью имело, по возможности, меньшую площадь. После этого данная процедура рекурсивно применяется к двум получившимся подобластям до тех пор, пока не

будет получено достаточное число подобластей. Далее все конечные элементы нумеруются в соответствии с уже изложенным выше принципом: на каждом уровне рекурсии сначала нумеруются элементы в получившихся подобластях, а затем уже — элементы, лежащие на разделяющей плоскости. Данный подход отличается относительной простотой в случае расчетных областей с несложной конфигурацией, когда легко проводить разделяющие плоскости. Кроме того, дополнительный плюс состоит в том, что основные временны́е затраты, связанные с разделением области на подобласти и разделяющие множества, не зависят от порядка используемых базисных функций, а зависят только от количества элементов сетки.

Таким образом, для параллелизации SSOR выполняются следующие действия. Для графа исходной матрицы запускается алгоритм вложенных сечений либо алгоритм геометрической декомпозиции области. Получаемые части рекурсивно делятся до тех пор, пока не будет получено необходимое число частей нижнего уровня (например, большее либо равное числу потоков). Затем выполняется перенумерация вершин графа и с помощью перестановки $\sigma(i)$ вычисляется новая матрица $\widehat{A} = P^{\mathrm{T}}AP$. В блочном виде эта матрица имеет вид

$$\widehat{A} = \begin{bmatrix} D_1 & 0 & B_{31}^{\mathrm{T}} & & & & B_{71}^{\mathrm{T}} & \dots & B_{k1}^{\mathrm{T}} \\ 0 & D_2 & B_{32}^{\mathrm{T}} & & 0 & & B_{72}^{\mathrm{T}} & \dots & B_{k2}^{\mathrm{T}} \\ B_{31} & B_{32} & D_3 & & & & B_{73}^{\mathrm{T}} & \dots & B_{k3}^{\mathrm{T}} \\ & & & D_4 & 0 & B_{64}^{\mathrm{T}} & B_{74}^{\mathrm{T}} & \dots & B_{k4}^{\mathrm{T}} \\ & 0 & & 0 & D_5 & B_{65}^{\mathrm{T}} & B_{75}^{\mathrm{T}} & \dots & B_{k5}^{\mathrm{T}} \\ & & & B_{64} & B_{65} & D_6 & B_{76}^{\mathrm{T}} & \dots & B_{k6}^{\mathrm{T}} \\ B_{71} & B_{72} & B_{73} & B_{74} & B_{75} & B_{76} & D_7 & \dots & B_{k,7}^{\mathrm{T}} \\ & \vdots & & & \vdots & & \vdots & & \vdots \\ B_{k1} & B_{k2} & B_{k3} & B_{k4} & B_{k5} & B_{k6} & B_{k7} & \dots & D_k \end{bmatrix}.$$

Некоторые из блоков B_{ij} могут оказаться нулевыми. Далее предлагается сконденсировать граф исходной матрицы так, чтобы вершинами сконденсированного графа стали разделители и полученные на самом нижнем уровне алгоритма части графа. Пусть сконденсированный граф обозначен как $G' = \{V', E'\}$, причем количество вершин в нем равно k. Ориентировав ребра этого графа естественным образом (задав направление ребра от вершины с бóльшим номером к вершине с меньшим номером для нижнетреугольных систем и от вершины с меньшим номером к вершине с бóльшим номером для верхнетреугольных), получим граф зависимостей блоков. Каждый из блоков D_i и соответствующие ему неизвестные компоненты вектора u назначаются некоторому потоку, после чего граф G' используется для отслеживания зависимостей между блоками.

Описанный подход к параллелизации SSOR по сравнению с обычной параллелизацией решения треугольных систем для исходной матрицы имеет то преимущество, что каждый поток на нижнем уровне обрабатывает большой блок как одно целое, а мощность разделителей предполагается небольшой по сравнению с общим числом вершин. В этом случае существенно снижаются затраты на синхронизацию потоков, что повышает производительность и масштабируемость.

4. Предлагаемые оптимизации. В первую очередь, необходимо отметить следующую особенность задачи распараллеливания и оптимизации предобусловливателя SSOR в модификации Айзенштата при использовании итерационных решателей: такое итерационное решение оказывается bandwidth-limited, т.е. существенно ограничено пропускной способностью шины данных системы. Действительно, легко видеть, что за одну итерацию алгоритмов CG и CR требуется один раз прочитать всю матрицу системы (для решения двух треугольных систем) и несколько раз прочитать и записать различные векторы. Таким образом, возможные оптимизации алгоритма можно разделить на две категории: оптимизация доступа к памяти и оптимизация, направленная на повышение производительности алгоритма на NUMA-системах, приобретающих все бóльшую популярность в последнее время. Эти системы отличаются тем, что имеют несколько блоков памяти и ее контроллеров, что позволяет разным процессорным сокетам одновременно читать различные участки памяти, повышая эффективную пропускную способность шины данных [5]. Однако перед обсуждением оптимизаций необходимо определить форматы данных, в которых будет храниться матрица либо ее блоки.

4.1. Форматы хранения данных. В простейшем случае матрицу A системы удобно хранить в формате CSR (Compressed Sparse Row, сжатый строчный формат), при этом требуется хранить следующую

информацию:

N — порядок системы;

$\mathrm{vals}\,(N_Z)$ — хранит ненулевые элементы матрицы;

$\mathrm{cols}\,(N_Z)$ — для каждого ненулевого элемента матрицы содержит номер столбца, в котором находится соответствующий элемент;

$\mathrm{rowInd}\,(N+1)$ — i-й элемент массива содержит индекс первого элемента i-й строки в массиве vals, N-й элемент (полагается нумерация строк и столбцов с 0) содержит число ненулевых элементов в матрице плюс один;

N_Z — число ненулевых элементов матрицы, вычисляемое по формуле $N_Z = \mathrm{rowInd}\,[N] - \mathrm{rowInd}\,[0]$.

В более общем случае возможна следующая модификация: вместо массива $\mathrm{rowInd}\,(N+1)$ хранятся два массива $\mathrm{rowBegin}\,(N)$ и $\mathrm{rowEnd}\,(N)$ — соответственно индексы начала и конца заданной строки в массивах vals и cols. Такой формат более удобен тем, что позволяет легко вырезать подматрицы из матрицы (например, нижне- и верхнетреугольные части) путем вычисления значений массивов rowBegin и rowEnd. Помимо этого указанный формат позволяет менять местоположение строк в массивах vals и cols, что может пригодиться в дальнейшем. Ниже представлен простейший пример кода, который выполняет умножение матрицы на вектор в рассматриваемом формате:

```
void multiply(int N, const int* rowBegin, const int* rowEnd,
    const int* cols, const double* vals,
    const double* x, double* y) {
        int i, j;
        for(i = 0; i < N; ++i)
            for(y[i] = 0, j = rowBegin[i]; j < rowEnd[i]; ++j)
                y[i] += vals[j] * x[cols[j]];
}
```

4.2. Оптимизации обращений к памяти. Первый этап состоит в явном вычислении матриц $\widehat{L}$, $\widehat{D}$ и $\widehat{U}$, $\widehat{L}+\widehat{D}+\widehat{U} = \widehat{A} = P^{\mathrm{T}}AP$. Несмотря на то что затраты памяти увеличиваются, это дает возможность проведения дальнейших оптимизаций. При решении треугольной системы $\left(I+\overline{L}\right)x = y$ последовательный алгоритм выполняет последовательные обращения к элементам всех массивов, кроме y, однако и для последнего обращения идут по возрастанию индексов элементов. Данное свойство позволяет существенно увеличить производительность, так как большинство аппаратных оптимизаций работы с кэшем процессора ориентированы на последовательное обращение к данным. Неприятность состоит в том, что процесс решения системы $\left(I+\overline{U}\right)x = y$ не отличается такой регулярностью. Предлагаемое решение проблемы состоит в том, чтобы переставить местами строки в массивах vals и cols для матрицы $\overline{U}$ так, чтобы первой шла последняя строка, далее — предпоследняя и т.д. Таким образом, если изначально элементы массивов были расположены следующим образом:

$$\mathrm{vals} = \left(a_{1c_{11}}\, a_{1c_{12}} \ldots a_{1c_{1k_1}} \middle| \ldots \middle| a_{Nc_{N1}}\, a_{Nc_{N2}} \ldots a_{Nc_{Nk_N}} \right),$$

$$\mathrm{cols} = \left(\begin{array}{cccc|c|cccc} c_{11} & c_{12} & \ldots & c_{1k_1} & \ldots & c_{N1} & c_{N2} & \ldots & c_{Nk_N} \end{array} \right),$$

то после перестановки элементы будут расположены как показано ниже:

$$\mathrm{vals} = \left(a_{Nc_{N1}}\, a_{Nc_{N2}} \ldots a_{Nc_{Nk_N}} \middle| \ldots \middle| a_{1c_{11}}\, a_{1c_{12}} \ldots a_{1c_{1k_1}} \right),$$

$$\mathrm{cols} = \left(\begin{array}{cccc|c|cccc} c_{N1} & c_{N2} & \ldots & c_{Nk_N} & \ldots & c_{11} & c_{12} & \ldots & c_{1k_1} \end{array} \right).$$

Ниже представлена процедура решения треугольной системы для матриц вида $\left(I+\overline{U}\right)$ в простейшем случае. Как видно из кода, при этом достигается последовательное обращение к элементам матрицы $\overline{U}$.

```
void solveU(int N, const int* rowBegin, const int* rowEnd,
    const int* cols, const double* vals,
    double* x, const double* y) {
        int i, j;
        for(i = 0; i < N; ++i)
            for(x[N-i-1] = y[N-i-1], j = rowBegin[i]; j < rowEnd[i]; ++j)
```

```
            x[N-i-1] -= vals[j] * x[cols[j]];
}
```

4.3. NUMA-оптимизации. Как уже отмечалось, NUMA-системы (системы с несколькими сокетами) имеют несколько контроллеров памяти. При страничной организации памяти каждая страница физически относится к памяти только одного сокета. Поэтому при необходимости обращения другого сокета к такой странице этот сокет осуществляет такой доступ посредством обращения к сокету, фактически владеющему страницей. При этом в ядре Linux по умолчанию используется принцип привязки страниц к сокетам, называемый first-touch, т.е. первый сокет, который обращается к еще не привязанной странице, предварительно созданной системным вызовом типа mmap, привязывает эту страницу к себе [5].

Из приведенного описания принципов работы NUMA-систем виден их существенный недостаток: если данные инициализируются в одном потоке, а затем используются в нескольких, то фактически данные оказываются привязанными к одному сокету, что может привести к снижению максимальной скорости чтения и записи данных по сравнению со случаем их равномерной привязки к разным сокетами системы. Для преодоления указанной сложности предлагается следующее.

Во-первых, необходимо осуществить статическую привязку потоков к блокам матрицы, получаемых в алгоритме вложенных сечений. Это требуется для того, чтобы на каждой итерации одни и те же потоки читали одни и те же блоки матрицы, что позволит привязать эти блоки к соответствующим сокетам и, соответственно, разместить их в памяти, "близкой" к текущему потоку. Произвести статическую привязку предлагается следующим образом. Вначале выделяются блоки, не имеющие зависимостей от других блоков. Эти блоки делятся равномерно между потоками либо по числу строк, либо по числу ненулевых элементов. После этого эти блоки исключаются из рассмотрения и выделяются блоки, которые не имеют зависимостей, кроме как от уже рассмотренных блоков. Далее процесс повторяется до тех пор, пока все блоки не будут назначены каким-либо потокам. В результате мы получим назначение блоков потокам, которое обеспечивает достаточно равномерную загрузку потоков.

Во-вторых, необходимо осуществить правильную привязку элементов блоков матрицы и частей векторов соответствующим потокам. Добиться этого несложно: достаточно выделить требуемую память с помощью вызова malloc и организовать копирование в массивы матриц $\overline{L}$ и $\overline{U}$, а также инициализацию элементов векторов в параллельном режиме таким образом, что каждый поток копирует и инициализирует только блоки, относящиеся к нему после статического назначения. В этом случае сработает принцип first-touch и страницы окажутся привязанными к необходимым сокетам. С учетом того, что на архитектурах x86 и x86-64 доступны страницы размером 4 килобайта и 2 мегабайта, можно заключить, что при большом объеме данных в матрице A (гигабайты и десятки гигабайт) возможно достичь высокой точности распределения данных по сокетам.

5. Численные эксперименты. Для подтверждения теоретических выводов, сделанных в рамках данной работы, а также для экспериментального тестирования производительности изложенных выше подходов был проведен ряд численных экспериментов на методических задачах. В качестве расчетной области рассматривался параллелепипед размера $1000 \times 100 \times 100$ мм с квазиструктурированной тетраэдральной сеткой, полученной из разбиения расчетной области на кубы с ребром 1 мм и дальнейшего разбиения кубов на 6 тетраэдров. В качестве задачи предлагалось решение уравнения Пуассона, которое в алгебраическом виде соответствует обращению нижнего блока предобусловливателя $\mathcal{P}_{M,L}$. В случае прямоугольной равномерной сетки и конечно-элементной аппроксимации вариационной задачи для этого уравнения узловыми конечными элементами первого порядка получается матрица следующего вида: $L_{ij} = \begin{cases} 6\mathcal{C}, & i = j, \\ -\mathcal{C}, & i \neq j, \end{cases}$ где $\mathcal{C}$ — некоторая константа, пропорциональная объему кубов сетки, i и j — соседние вершины сетки, связанные ребром, остальные элементы матрицы L равны нулю. Далее решалось матричное уравнение $Lx = f$ со случайным вектором x, $0 \leqslant x_i \leqslant 1$, при помощи метода сопряженных градиентов (CG). Производилась геометрическая декомпозиция расчетной области путем деления ее на 16 частей по оси Ox (так что суммарное число частей, включая разделители, оказывалось равным 31). Параметр релаксации ω был выбран равным 1.85, критерием остановки итераций служило условие $|f - Lx| < \varepsilon |f|$, $\varepsilon = 10^{-7}$. Общее число итераций при этом оказывалось равным 65. Итерационный решатель и предобусловливатель SSOR были распараллелены с использованием OpenMP. Результаты численных экспериментов представлены ниже.

Для проведения численных экспериментов использовались следующие серверы:

— Intel Xeon X5670, 2.93 ГГц, 2 сокета × 6 ядер (всего 12 ядер), включен Hyper Threading;

— Intel Xeon X7560, 2.27 ГГц, 4 сокета × 8 ядер (всего 32 ядра).

В качестве библиотеки OpenMP использовалась библиотека, входящая в состав компилятора Intel Composer

XE 2011.

Для сравнительного тестирования производительности был также реализован обычный метод сопряженных градиентов с предобусловливателем SSOR с использованием библиотеки Intel MKL 10.3. В частности, из данной библиотеки использовались функции BLAS уровня 1, а также функции решения треугольных разреженных систем из Sparse BLAS уровня 2. Все эти функции распараллелены внутри библиотеки Intel MKL [6]. Для того чтобы обеспечить такое же число итераций, как и у предлагаемого в данной работе алгоритма, в случае MKL SSOR также производилось переупорядочивание матрицы.

Таблица 1

Время работы решателей, Intel Xeon X5670 @ 2.93 GHz

Предобусловливатель	KMP_AFFINITY	Число потоков						
		1	2	4	6	8	12	24
Par SSOR	compact,0,0	15.8	14.2	11.4	11.7	11.8	12.0	8.26
Par SSOR	compact,1,0	16.0	11.9	11.4	11.6	92.4	7.90	8.44
Par SSOR	scatter,0,0	15.9	10.0	8.07	7.91	7.78	8.03	8.41
MKL SSOR	compact,0,0	22.9	26.4	25.4	26.6	26.6	26.6	26.6
MKL SSOR	compact,1,0	22.1	20.7	21.3	21.6	19.7	18.3	18.4
MKL SSOR	scatter,0,0	21.9	17.7	17.3	17.7	17.9	18.5	18.4

Таблица 2

Время работы решателей, Intel Xeon X7560 @ 2.27 GHz

Предобусловливатель	KMP_AFFINITY	Число потоков					
		1	2	4	8	16	32
Par SSOR	compact,0,0	30.4	18.9	15.5	14.8	10.2	8.56
Par SSOR	scatter,0,0	30.4	17.1	9.69	7.48	8.56	8.57
MKL SSOR	compact,0,0	40.6	34.5	33.5	33.4	28.9	26.3
MKL SSOR	scatter,0,0	40.6	31.8	27.1	26.1	26.3	26.3

В табл. 1 и 2 представлены результаты сравнительного тестирования предлагаемого алгоритма (Par SSOR) и алгоритма, реализованного с использованием библиотеки Intel MKL (MKL SSOR). В таблицах приведены времена работы решателей в секундах в зависимости от числа потоков, а также в зависимости от установки переменной окружения KMP_AFFINITY, определяющей распределение потоков OpenMP по сокетам, ядрам и логическим процессорам (в случае включенного режима Hyper Threading) системы.

На рис. 1 приведены следующие графики:

1 — Par SSOR, compact,0,0; 4 — MKL SSOR, compact,0,0;

2 — Par SSOR, compact,1,0; 5 — MKL SSOR, compact,1,0;

3 — Par SSOR, scatter,0,0; 6 — MKL SSOR, scatter,0,0.

На рис. 2 приведены следующие графики:

1 — Par SSOR, compact,0,0; 3 — MKL SSOR, compact,0,0;

2 — Par SSOR, scatter,0,0; 4 — MKL SSOR, scatter,0,0.

Из полученных результатов видно, что алгоритм в действительности является ограниченным по пропускной способности шины данных: при небольшом числе потоков бóльшую производительность имеют варианты расположения потоков на разных сокетах системы. Кроме того, видно, что слишком большое

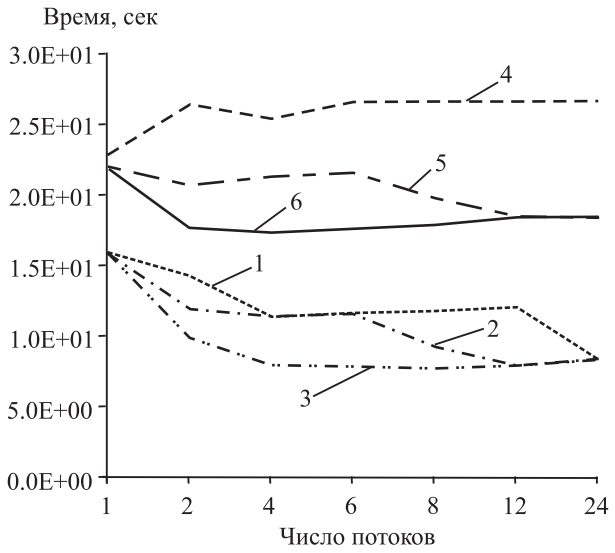

Рис. 1. Сравнение производительности решателей, Intel Xeon X5670 @ 2.93 GHz

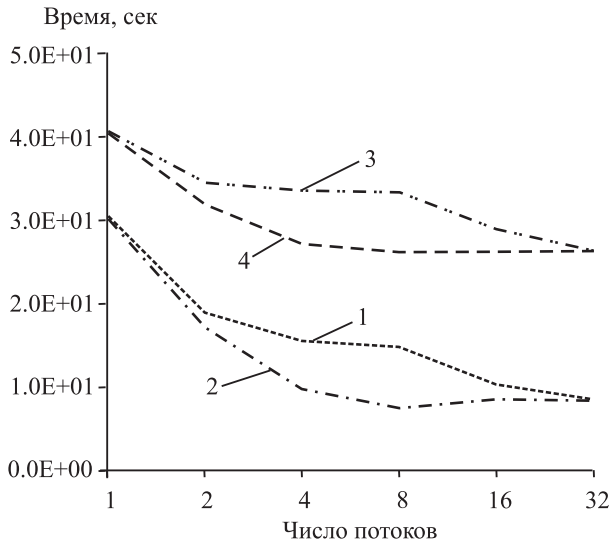

Рис. 2. Сравнение производительности решателей, Intel Xeon X7560 @ 2.27 GHz

число потоков приводит к снижению производительности, поскольку они начинают конкурировать за шину данных. В итоге алгоритм показывает оптимальную производительность в случае, когда на один сокет приходится приблизительно 2 потока.

Кроме того, можно отметить высокую производительность полученного алгоритма, а также его довольно хорошую масштабируемость по сравнению с масштабируемостью алгоритма при использовании библиотеки Intel MKL. В наилучшем варианте алгоритм показывает в 2.6 раза лучшую масштабируемость и в 3.5 раза лучшую производительность, при этом максимальная достигнутая масштабируемость алгоритма составляет 4.1 раза на Intel Xeon X7560 на 8 потоках.

6. Заключение. В работе предложен подход к построению масштабируемого алгоритма решения треугольных систем для предобусловливателя SSOR. Описан ряд оптимизаций, позволяющих повысить производительность алгоритма. Приведенные численные эксперименты показывают, что алгоритм демонстрирует масштабируемость до 4-х раз на некоторых из доступных в настоящее время многопроцессорных машин с общей памятью и существенно эффективнее реализации с использованием параллельных функций решения треугольных систем библиотеки Intel MKL.

Отметим также, что хотя в работе рассматривается случай симметричной матрицы A, предложенный подход может быть применен для любых матриц с симметричным портретом. Более того, данный подход может быть использован для построения кластерной версии предобусловливателя SSOR, однако эта тема требует дальнейших исследований. Кроме того, одним из возможных направлений является исследование вопроса построения наиболее эффективного метода отслеживания зависимостей между блоками во время работы программы, когда некоторые из блоков вектора неизвестных оказываются вычисленными.

СПИСОК ЛИТЕРАТУРЫ

1. *Ильин В.П.* Методы и технологии конечных элементов. Новосибирск: ИВМиМГ СО РАН, 2007.
2. *Karypis G., Kumar V.* A fast and highly quality multilevel scheme for partitioning irregular graphs // SIAM J. on Scientific Computing. 1999. **20**, N 1. 359–392.
3. *Greif C., Schötzau D.* Preconditioners for the discretized time-harmonic Maxwell equations in mixed form // Numer. Linear Algebra Appl. 2007. **14**. 281–297.
4. *Bossavit A.* Computational electromagnetism. Variational formulations, complementarity, edge elements. San Diego: Academic Press, 1998.
5. Optimizing Software Applications for NUMA (http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/).
6. Intel (R) Math Kernel Library Documentation (http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/).

Поступила в редакцию
07.03.2011