МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
**"Южно-Уральский государственный университет
(национальный исследовательский университет)"**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

РАБОТА ПРОВЕРЕНА

Рецензент
к.ф.-м.н., доцент кафедры ТУиО
математического университета
ЧелГУ

_____ С.А. Никитина

“___”_____ 2017 г.

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой,
д.ф.-м.н., профессор

_____ Л.Б. Соколинский

“___”_____ 2017 г.

Q-ЭФФЕКТИВНЫЙ КОДИЗАЙН РЕАЛИЗАЦИИ МЕТОДА ГАУССА-ЖОРДАНА НА СУПЕРКОМПЬЮТЕРЕ «ТОРНАДО ЮУРГУ»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.04.02.2017.115-079.ВКР

Научный руководитель
к.ф.-м.н., доцент

_____В. Н. Алеева

Автор работы,
студент группы КЭ-217

_____Д. Е. Тарасов

Ученый секретарь
(нормоконтролер)

_____ О.Н. Иванова

“___”_____ 2017 г.

Челябинск-2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время уровень развития вычислительной техники и методов математического моделирования дает возможность для перевода научных исследований на новый этап. Масштабность задач на порядок превышает наши представления даже пятилетней давности. Решение задач, требуемых большого объема вычислений может быть выполнено только при использовании суперкомпьютерных вычислительных систем.

Способность суперкомпьютерных вычислительных систем выдвигает на один из первых планов необходимость проведения вычислений большого объема. Быстродействие таких вычислений достигается за счет использования высокопроизводительных систем использующих сотни и тысячи процессоров. Такой подход выполнения вычислений предполагает разделение задачи на несколько подзадач и их выполнение на разных процессорах.

Одним из основных вопросов при распараллеливании вычислений является выбор алгоритма для поиска решений исследуемых задач. Существует мнение, что верхний предел возможности распараллеливания программ задает алгоритм.

При разработке параллельных программ необходимо эффективно использовать десятки миллионов компонентов суперкомпьютерных систем. Одним из таких подходов разработки является суперкомпьютерный кодизайн.

Суперкомпьютерный кодизайн – это процесс разработки параллельных приложений, в котором учитываются и согласуются особенности алгоритмов, технологий распараллеливания, реализации в коде и аппаратном обеспечении для создания высокоэффективных программ для вычислительных систем. [13]. Суперкомпьютерный кодизайн требует учета особенностей функционирования каждой составляющей вычислительной системы.

Актуальность темы исследования

Повышение производительности параллельных вычислительных систем может быть достигнуто за счет максимального распараллеливания алгоритмов. Одним из таких подходов распараллеливания является концепция Q-детерминанта. Концепция Q-детерминанта позволяет анализировать ресурс распараллеливания любого алгоритма, в том числе находить все его реализации, включая максимально быструю. Данная концепция была разработана и предложена В.Н. Алеевой [2]. Данное направление активно развивается много лет и уже достигнуты результаты [2, 3, 7, 11, 12, 14], основанные на концепции Q-детерминанта.

Для любого численного алгоритма можно построить Q-детерминант и найти Q-эффективную реализацию алгоритма. Если Q-эффективная реализация является выполнимой, то ее можно непосредственно программировать. Совместив концепции суперкомпьютерного кодизайна и Q-детерминанта алгоритма можно выполнить Q-эффективный кодизайн алгоритма на суперкомпьютере «Торнадо ЮУрГУ».

Цель и задачи

Целью данной работы является Q-эффективный кодизайн реализации метода Гаусса-Жордана на суперкомпьютере «Торнадо ЮУрГУ».

Для достижения поставленной цели необходимо решить следующие задачи:

1) изучить подход к распараллеливанию алгоритмов на основе их представления в форме Q-детерминанта;

2) построить Q-детерминант метода Гаусса-Жордана;

3) изучить архитектуру суперкомпьютера «Торнадо ЮУрГУ»;

4) выполнить Q-эффективную реализацию метода Гаусса-Жордана на общей и распределенной памяти с использованием технологий распараллеливания OpenMP и MPI + OpenMP;

5) выполнить тестирование разработанных Q-эффективных программ.

Структура и объем работы

Диссертационная работа состоит из введения, 3 глав, заключения и библиографии. Объем диссертации составляет 33 страницы, объем библиографии – 14 наименований, объем приложения – 8 страниц.

Краткий обзор содержания работы

Первая раздел содержит описание концепции Q-детерминанта и метод конструирования параллельных программ на основе данной концепции. Описывается применение метода построения параллельных программ для алгоритма Гаусса-Жордана.

Второй раздел содержит проектирование и реализацию Q-эффективных программ. Рассматриваются технологии программирования и архитектура суперкомпьютера «Торнадо ЮУрГУ».

В третьем разделе представлены результаты тестирования полученных Q-эффективных программ. Выполняется построение графиков зависимостей времени выполнения, ускорения, эффективности и от размера задачи и количества процессоров.

В заключении подводятся итоги проведенного исследования.

1. Q-ЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА

1.1. Концепция Q-детерминанта

Исследования, проводимые в работе, основаны на концепции Q-детерминанта [2]. Представление алгоритма в форме Q-детерминанта описывает все возможные реализации алгоритма, а также позволяет найти максимально быструю реализацию, при которой операции выполняются, как только вычислены значения их операндов.

Пусть A – это некоторый алгоритм для решения алгоритмической проблемы $\bar{y} = F(N, B)$, где N – множество параметров размерности задачи, B – множество входных данных, $\bar{y} = (y_1, \dots, y_n)$ – множество выходных данных.

Пусть Q – это множество операций, используемых алгоритмом A. Все операции из множества Q нульместны (константы), одноместны, или двуместны (арифметические, логические операции, операции сравнения). Над множествами B и Q можно строить выражения, которые могут иметь уровни вложенности. Уровень вложенности выражения w обозначим через T^w.

Под Q-термом будем понимать отображение размерности задачи в множество всех выражений, необходимое для вычисления одной величины из множества входных данных. В зависимости от структуры множества выражений различают безусловные, условные и условные бесконечные Q-термы.

Множество Q-термов, необходимых для вычисления всех выходных данных задачи, называется Q-детерминантом и обозначается, как $y_i = f_i(i = 1 \dots m)$, где f_i – Q-терм для вычисления y_i, m – количество выходных данных

Под реализацией алгоритма, представленного в форме Q-детерминанта, понимается вычисление Q-термов. Реализация алгоритма будет считаться Q-эффективной, если операции из множества Q выполняются сразу по мере готовности к выполнению.

С формальной точки зрения, Q-эффективная реализация алгоритма является максимально быстрой реализацией. И если алгоритм допускает распараллеливание, то его Q-эффективная реализация полностью использует ресурс параллелизма. Если одновременно необходимо выполнять конечное число операций, то реализация алгоритма называется выполнимой.

1.2. Метод конструирования параллельных программ на основе концепции Q-детерминанта

Q-эффективная реализация алгоритма основана на следующих утверждениях [2]:

- для любого численного алгоритма можно построить Q-детерминант.
- Q-детерминант позволяет описать Q-эффективную реализацию алгоритма.
- если Q-эффективная реализация алгоритма является выполнимой, то можно разработать программу для ее выполнения.

Процесс разработки программы состоит из трех этапов:

1) построение Q-детерминанта алгоритма;
2) описание Q-эффективной реализации;
3) если Q-эффективная реализация выполнима, то для нее разрабатывается программа.

Разработанную программу будем называть Q-эффективной, а процесс ее разработки Q-эффективным программированием. Q-эффективная программа полностью использует ресурс параллелизма, так как выполняет его Q-эффективную реализацию. В связи с этим ее дальнейшее распараллеливание невозможно.

На третьем этапе для разработки используются средства параллельного программирования. При использовании распределенной памяти вычисления ограничиваются по принципу «Мастер - Рабочие», который часто применяется на кластерных вычислительных системах.

В данной работе осуществляется разработка Q-эффективной про-

граммы для алгоритма Гаусса-Жордана на основе данного метода с использованием средств распараллеливания OpenMP и MPI для отладки и тестировании на суперкомпьютере Торнадо «ЮУрГУ».

1.3. Метод Гаусса-Жордана

Метод Гаусса-Жордана для решения систем линейных уравнений $A\bar{x} = \bar{b}$ можно применять для любых размерностей. Для простоты будем считать, что $A = \left|a_{ij}\right|_{i,j=1\dots n}$матрица размерности $n \times n$ с ненулевым определителем. $\bar{x} = (x_1, x_2, \dots, x_n)^T$, $\bar{b} = (a_{1,n+1}, \dots, a_{n,n+1})^T$, $\bar{A} = [a_{ij}]$ – расширенная матрица системы.

1.4. Применение метода конструирования параллельных программ для алгоритма Гаусса-Жордана

В данном разделе опишем первые два этапа метода конструирования параллельных программ для алгоритма Гаусса-Жордана.

1.4.1. Этап 1. Построение Q-детерминанта алгоритма

Выберем ведущий элемент $a_{1j_1}\left(a_{1j} = 0 \text{ при } j < j_1 \le n, a_{1j_1} \ne 0\right)$. Получим расширенную матрицу $\overline{A^{J1}} = [a_{ij}^{j_1}]$, элементы которой вычисляются по формулам:

$$a_{1j}^{j_1} = \frac{a_{1j}}{a_{1j_1}},$$

$$a_{ij}^{j_1} = a_{ij} - \frac{a_{1j}}{a_{1j_1}} a_{ij_1} \ (i = 2, \dots, n; j = 1, \dots, n+1).$$

Шаг k. $2 \le k \le n$

В качестве ведущего элемента выбирается $a_{kj_k}^{j_1 \dots j_{k-1}}$ с условием, что $a_{kj}^{j_1 \dots j_{k-1}} = 0$ при $j < j_k \le n, a_{kj_k}^{j_1 \dots j_{k-1}} \ne 0$.

Получим матрицу $\bar{A}^{j_1 \dots j_{k-1}} = [a_{ij}^{j_1 \dots j_k}]_{i=1,\dots,n;j=1,\dots,n+1}$, элементы кот рой вычисляются по формулам:

$$a_{kj}^{j_1 \dots j_k} = \frac{a_{kj}^{j_1 \dots j_k - 1}}{a_{kj_k}^{j_1 \dots j_{k-1}}},$$

$(k = i, j = 1, \dots, n+1)$,

$$a_{ij}^{j_1\dots j_k} = a_{ij}^{j_1\dots j_{k-1}} - \frac{a_{kj}^{j_1\dots j_{k-1}}}{a_{kj_k}^{j_1\dots j_{k-1}}} a_{kj_k}^{j_1\dots j_{k-1}} \quad (i = 1, \dots n; i \neq k; j = 1, \dots, n+1).$$

Шаг n.

После *n*-го шага получаем систему уравнений $A^{j_1\dots j_n}\bar{x} = \bar{b}^{j_1\dots j_n}$, где:

$$A^{j_1\dots j_n} = \left[a_{ij}^{j_1\dots j_n}\right]_{i=1,\dots,n;j=1,\dots,n},$$

$$\bar{b}^{j_1\dots j_n} = \left(a_{1,n+1}^{j_1\dots j_n}, \dots, a_{n,n+1}^{j_1\dots j_n}\right)^T.$$

Введем обозначения:

$$L_{j_1} = \wedge_{j=1}^{j_1-1}(a_{1j} = 0), \text{ если } j_1 \neq 1,$$

$$L_{j_1} = true, \text{ если } j_1 = 1,$$

$$L_{jl} = \wedge_{\substack{j=1 \\ j\notin\{j_1,\dots j_{l-1}\}}}^{j_l-1} (a_{lj}^{j_1\dots j_{l-1}} = 0)(l = 2, \dots, n), \text{ если } j_1 \neq 1,$$

$$L_{jl} = true, \text{ если } j_1 = 1.$$

Пусть *i* номер перестановки $(j_1, \dots, j_n)$. Тогда:

$$w_i^{j_l} = a_{l,n+1}^{j_1\dots j_n}(l = 1, \dots, n)$$

$$u_i = L_{j_1} \wedge (a_{1j_1} \neq 0) \wedge \left(\bigwedge_{l=2}^{n}(L_{j_l} \wedge (a_{lj_l}^{j_1\dots j_{l-1}} \neq 0))\right)$$

являются безусловными Q-термами.

Q-детерминант метода Гаусса-Жордана состоит из *n* условных Q-термов, а представление в форме Q-детерминанта имеет вид:

$$x_j = \left\{\left(u_1, w_1^j\right), \dots, \left(u_{n!}, w_{n!}^j\right)\right\}(j = 1, \dots, n)$$

1.4.2. Этап 2. Описание Q-эффективной реализации алгоритма

По определению Q-эффективной реализации все безусловные Q-термы $\left\{u_i, w_i^j\right\}(i = 1, \dots, n!; j = 1, \dots, n)$ должны вычисляться одновременно, т.е. параллельно.

На каждом k-ом шаге алгоритма предполагается, что любой элемент $a_{kj}^{j_1\dots j_k}$ *(k = 1,..,n; j = k,…,n)* может быть ведущим. Для каждого предполагаемого ведущего элемента параллельно должны выполняться два вычисли-

тельных процесса:

- параллельное вычисление матриц $\bar{A}^{j_1}, \bar{A}^{j_2}, \dots, \bar{A}^{j_1 j_2 \dots j_n}$ для любых возможных значений $j_1, j_2, \dots, j_n$;
- параллельное вычисление Q-термов u_i $(i = 1, \dots, n!)$.

Ведущий элемент для очередного шага определяется по правилу u_i $(i = 1, \dots, n!)$. Как только при вычислении u_i определяется ведущий элемент для очередного шага алгоритма, то вычисление матриц, не соответствующих этому ведущими элементу, прекращается.

Предположим, что на каждом шаге ведущий элемент определяется быстрее, чем происходит вычисление матриц для этого шага. Тогда одновременно начинают вычисляться матрицы $\bar{A}^{j_1}$ $(j_1 = 1, \dots, n)$ и Q-термы u_i $(i = 1, \dots, n!)$.

Вычисление u_i $(i = 1, \dots n!)$ начинается с подвыражений $L_{j_1} \wedge (a_{1 j_1} \neq 0)(j_1 = 1, \dots, n)$, как только их операции готовы к выполнению. Q-терм u_i является логическим и может принимать значения *true* или *false*. Истинное значение будет иметь только одно из подвыражений. Пусть r_1 обозначает значение j_1, для которого u_i принимает значение *true*.

Следует прекратить вычисление всех $\bar{A}^{j_1}$ $(j_1 = 1, \dots, n)$ и u_i $(i = 1, \dots, n!)$, для которых $r_1 \neq j_1$. Во время второго шага вычислений нужно одновременно вычислять матрицы $\bar{A}^{r_1 j_2}$ $(j_1 = 1, \dots, n;\ j_2 \neq r_1)$ и одновременно вычислять Q-термы u_i $(i = 1, \dots, n!)$, для которых $j_1 = r_1$. При вычислении u_i (i = 1,…,n!) нужно вычислять только подвыражения $L_{j_2} \wedge \left(a_{2 j_2}^{r_1} \neq 0\right)(j_2 = 1, \dots, n;\ j_2 \neq r_1)$, т.к. только их операции готовы к выполнению. Истинное значение будет иметь только одно из подвыражений . $L_{j_2} \wedge \left(a_{2 j_2}^{r_1} \neq 0\right)(j_2 = 1, \dots, n;\ j_2 \neq r_1)$. Пусть r_2 обозначает значение j_2, для которого u_i принимает значение *true*.

Следует прекратить вычисление всех $\bar{A}^{r_1 j_2}$ $(j_2 = 1, \dots, n;\ j_2 \neq r_1)$ и u_i $(i = 1, \dots, n!)$, для которых $r_2 \neq j_2$.

Аналогично выполняются следующие $n - 3$ шага вычислений. На

последнем шаге необходимо вычислить одну матрицу $\bar{A}^{r_1 \dots r_{n-1} j_n}$ ($j_n \neq r_1, r_2, \dots, r_{n-1}$), так как значение j_n может быть единственным, обозначим его r_n.

В результате выполнения n-шагов алгоритма будет получено решение системы линейных уравнений $x_{r_j} = a_{j,n+1}^{r_1 \dots r_n} (j = 1, \dots, n)$.

2. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ Q-ЭФФЕКТИВНЫХ ПРОГРАММ ДЛЯ МЕТОДА ГАУССА-ЖОРДАНА

2.1. Используемые технологии программирования

Разработаем Q-эффективные программы для общей и распределенной памяти. Исследование проводится на суперкомпьютере «Торнадо ЮУрГУ».

Для разработки программ используется язык программирования C++. Реализация Q-эффективной программы для общей памяти осуществляется с использованием технологии OpenMP, для распределенной памяти – MPI.

Разработка программы для общей памяти осуществляется в соответствии с планом выполнения Q-эффективной реализации, описанной в разделе 1.

При работе с распределенной памятью организация вычислений осуществляется по принципу «Мастер – Рабочие» (Master – Slave). На практике методика работает по следующей схеме. Для вычисления используется один вычислительный узел M (Мастер) и несколько вычислительных узлов P (Рабочий). Вычислительный процесс разбивается на несколько шагов.

Шаг 0: инициализация.

Шаг 1: посылка задания от узла M всем узлам P.

Шаг 2: вычисление на каждом узле P без обменов с другими узлами.

Шаг 3: посылка результатов от всех узлов P узлу M.

Шаг 4: слияние на узле M полученных результатов.

Для разработки Q-эффективной программы для распределенной памяти план выполнения Q-эффективной реализации дополним планом распределения вычислений по узлам. Каждая матрица $\bar{A}^{j_1}$ $(j_1 = 1, \dots, n)$ и соответствующий ей Q-терм u_i $(i = 1, \dots, n!)$ должны вычисляться на своем узле P. Если количество узлов меньше размерности задачи, то на одном узле могут вычисляться матрицы и соответствующие ей Q-термы для не-

скольких значений j_1.

Узел M посылает узлам P информацию, необходимую для вычисления матриц $\bar{A}^{j_1}$ $(j_1 = 1, \dots, n)$ и соответствующих им Q-термов u_i $(i = 1, \dots, n!)$. Результаты вычисления r_1 и $\bar{A}^{r_1}$ передаются на узел M. Вычисления прекращаются на всех узлах, для которых логический Q-терм u_i $(i = 1, \dots, n!)$ содержит значение *false*.

На следующем шаге вычисления узел M посылает узлам P информацию, необходимую для вычисления $\bar{A}^{r_1 j_2}$ $(j_1 = 1, \dots, n;\ j_2 \neq r_1)$ и соответствующих им Q-термов u_i $(i = 1, \dots, n!)$. Если количество узлов меньше размерности задачи, то на одном узле могут вычисляться матрицы и соответствующие ей Q-термы для нескольких значений j_2. Результаты вычисления r_2 и $\bar{A}^{r_1 r_2}$ передаются на узел M. Вычисления прекращаются на всех узлах, для которых логический Q-терм u_i $(i = 1, \dots, n!)$ содержит значение *false*.

Аналогично выполняются $n - 2$ цикла вычислений.

2.1.1. Технология распараллеливания OpenMP

Технология распараллеливания OpenMP наиболее широко применяется в настоящее время для организации параллельных вычислений. Организация параллельных вычислений с использованием OpenMP осуществляется на многоядерных и многопроцессорных системах с общей памятью.

Для выделения параллелизма в OpenMP применяются специальные директивы, которые позволяют разбивать последовательные участки кода на несколько потоков (распараллеливать). Такой подход позволяет получить представление программы в виде набора последовательных и параллельных участков кода.

В рамках понятия OpenMP, под параллельно программой понимается программа, для которой в специально указываемых при помощи директив местах – параллельных фрагментах – исполняемый программный код может быть разделен на несколько раздельных командных потоков. [6, 9].

В состав технологии OpenMP входят:

- директивы;
- библиотека функций;
- набор переменных окружения.

Для данного исследования выделим директивы и функции, необходимые для Q-эффективной реализации метода Гаусса-Жордана для общей памяти.

Для выделения параллельного участка кода применяется директива `#pragma omp parallel` с указание количества выделяемых потоков. Так как в программе необходимо использовать операторы цикла, то в OpenMP для распараллеливания цикла применяется директива #pragma omp for или #pragma omp parallel for. При таком подходе все операции цикла выполняются одновременно, т.е. параллельно и каждая операция цикла выполняется своим отдельным потоком. В любой параллельной программе возникает необходимость знать общее число потоков и ранг каждого потока. Получения этой информации применяются функции omp_get_num_threads() и omp_get_thread_num().

Для управления режимом выполнения вложенных параллельных участков кода применяется функция omp_set_nested(), которая разрешает или запрещает режим поддержки вложенных параллельных фрагментов.

2.1.2. Технология распараллеливания MPI

Для организации вычислений с распределенной памятью используется технология распараллеливания MPI. Технология MPI в своем названии отражает и принцип организации распараллеливания. Аббревиатура MPI расшифровывается, как Message Passing Interface (интерфейс передачи сообщений). В рамках работы с MPI не осуществляется разделение кода на отдельные фрагменты, как это организовано в OpenMP. А разрабатывается одна программа (именуемая процессом), которая запускается одновременно на всех процессах.

В рамках понятия MPI, под параллельной программой подразумевается множество одновременно выполняемых процессов [6, 8].

Основными операциями в MPI являются операции передачи сообщений. Различают парные и коллективные операции. Операции обмена сообщениями разделяются на блокирующие и неблокирующие.

Так как концепция Q-эффективной реализации подразумевает одновременное выполнение операций вычисления матрицы и соответствующих им Q-термов, то возникает необходимость распараллелить эти процессы. Для реализации алгоритма ориентированной на вычислительную систему с общей памятью используется вложенный параллелизм. Реализация алгоритма для вычислительной системы с распределенной памятью не является исключением, вложенный параллелизм остается обязательным элементом реализации.

На верхнем уровне распараллеливания используется технология MPI. В отличие от реализации с использованием OpenMP матрица и соответствующий ей Q-терм должны вычисляться на своем отдельном узле. Для организации распараллеливания вычислений матрицы и соответствующего ей Q-терма на отдельном узле необходимо снова применить технологию распараллеливания OpenMP.

2.2. Архитектура суперкомпьютера «Торнадо ЮУрГУ»

Суперкомпьютер «Торнадо ЮУрГУ» представляет собой вычислительный кластер, состоящий из 480 вычислительных узлов 2-х серверов (управление и доступ пользователей), объединенных между собой с помощью транспортной сети Infiniband и двух Ethernet сетей (мониторинг и управление заданиями).

Вычислительная система входит в рейтинг СНГ TOP50 и Green500.

Основной функцией суперкомпьютера «Торнадо ЮУрГУ» является решение задач с высокой вычислительной нагрузкой.

Вычислительная система обладает высокой энергоэффективностью за счет отвода тепла с помощью жидкостной системы охлаждения, высокой вычислительной плотностью, высокой отказоустойчивостью за счет отсутствия движущихся частей (вентиляторы, жесткие диски).

Каждый вычислительный узел имеет в своем составе сопроцессор Intel Xeon Phi, сетевой интерфейс Infiniband, сетевой интерфейс Ethernet.

На рисунке 1 представлена функциональная схема вычислительной системы [10].

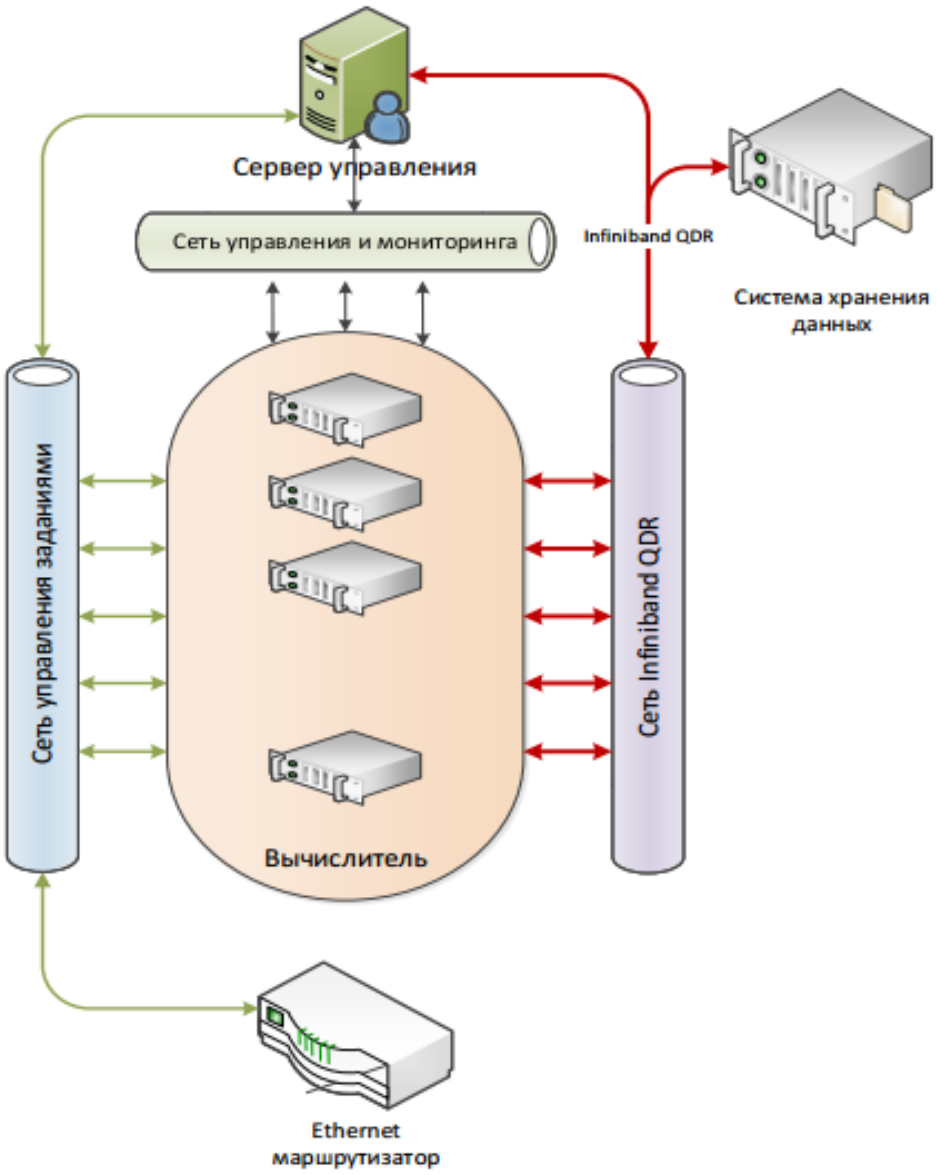

Рис. 1. Функциональная схема суперкомпьютера «Торнадо ЮУрГУ»

В таблице 1 представлены основные технические характеристики вычислительной системы [10].

Табл. 1. Технические характеристики суперкомпьютера

Характеристика	**Значение**
Количество узлов / процессорных ядер	480 / 29184

Окончание табл. 1.

Характеристика	Значение
Тип процессора	Intel Xeon X5680 (Gulftown, 6 ядер по 3.33 ГГц) — 960 шт
Оперативная память	16.9 TB
Дисковая память	300 TB, твердотельные накопители SSD Intel, параллельная система хранения данных Panasas ActiveStor 11
Тип управляющей сети:	Gigabit Ethernet
Пиковая производительность комплекса	473.6 TFlops
Производительность комплекса на тесте LINPACK:	288.2 TFlops
Операционная система	Linux CentOS 6.2

В таблице 2 представлены характеристики одного вычислительного узла суперкомпьютера «Торнадо ЮУрГУ» [10].

Табл. 2. Характеристики одного вычислительного узла суперкомпьютера «Торнадо ЮУрГУ»

Характеристика	Значение
Процессор	Intel Xeon X5680 3.33 GHz
Количество процессоров	2
Количество ядер	12 ядер / 24 потока на узел
Оперативная память	24 Гб ECC DDR3 Full buffered
Дисковое пространство	80 Гб
Производительность, Тфлопс	0.371

2.3. Проектирование и реализация

Для описанного плана выполнения разработаем Q-эффективные программы для общей и распределенной памяти.

Для достижения универсальности программ будем использовать операции динамического распределения памяти. Это позволит выполнять тестирование программ для разных размерностей.

Для разрабатываемых программ в качестве исходных данных выступает матрица любой размерности и размер задачи.

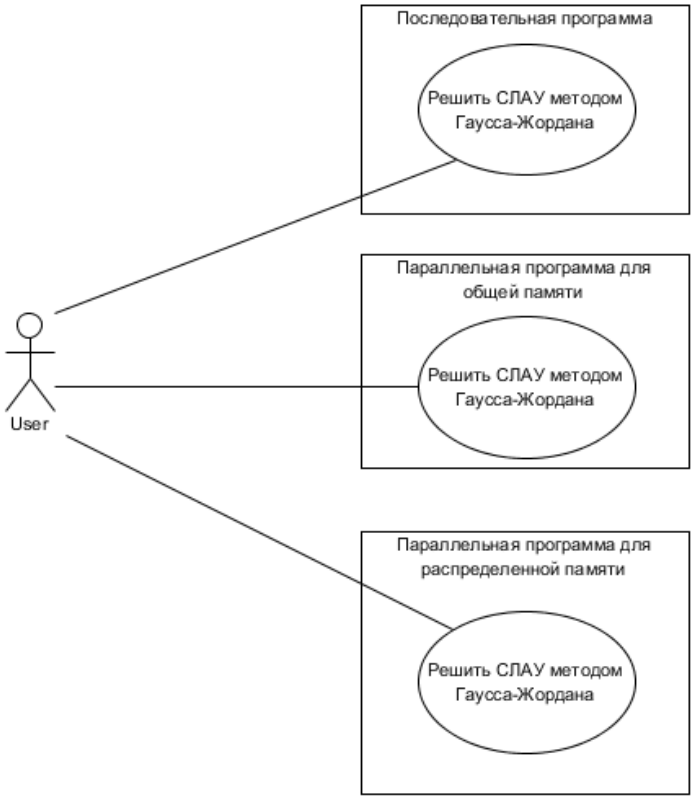

Рис. 2. Диаграмма вариантов использования

Разработаем схему алгоритма реализации Q-эффективной программы на общей памяти.

2.3.1. Модель с общей памятью

Метод Гаусса-Жордана состоит из n шагов. Во время выполнения алгоритма необходимо распределить по процессам n матриц размером $n \times n$, для их одновременного вычисления на каждом шаге. Одновременно с вычислением матриц осуществляется вычисление логических Q-термов. Вычисление логических Q-термов на k-ом шаге алгоритма происходит на основании матрицы, полученной на $k - 1$ шаге алгоритма. Вычисление логических Q-термов для n элементов накладывает ограничения на использование памяти. Ограничения достигаются использованием дополнительных n массивов размером $n \times n$, содержащих матрицу, полученную на предыдущем шаге. После выполнения каждого шага алгоритма результат выполнения передается в матрицу размера $n \times n$.

На рис. 3 представлена схема выполнения Q-эффективной програм-

мы. На первом шаге осуществляется инициализация исходной матрицы. Далее выполняется k-шагов алгоритма. На каждом k-ом шаге алгоритма выполняется параллельное вычисление матриц и соответствующих им Q-термов. При определении ведущего элемента на k-ом шаге осуществляется переход на *k + 1* шаг.

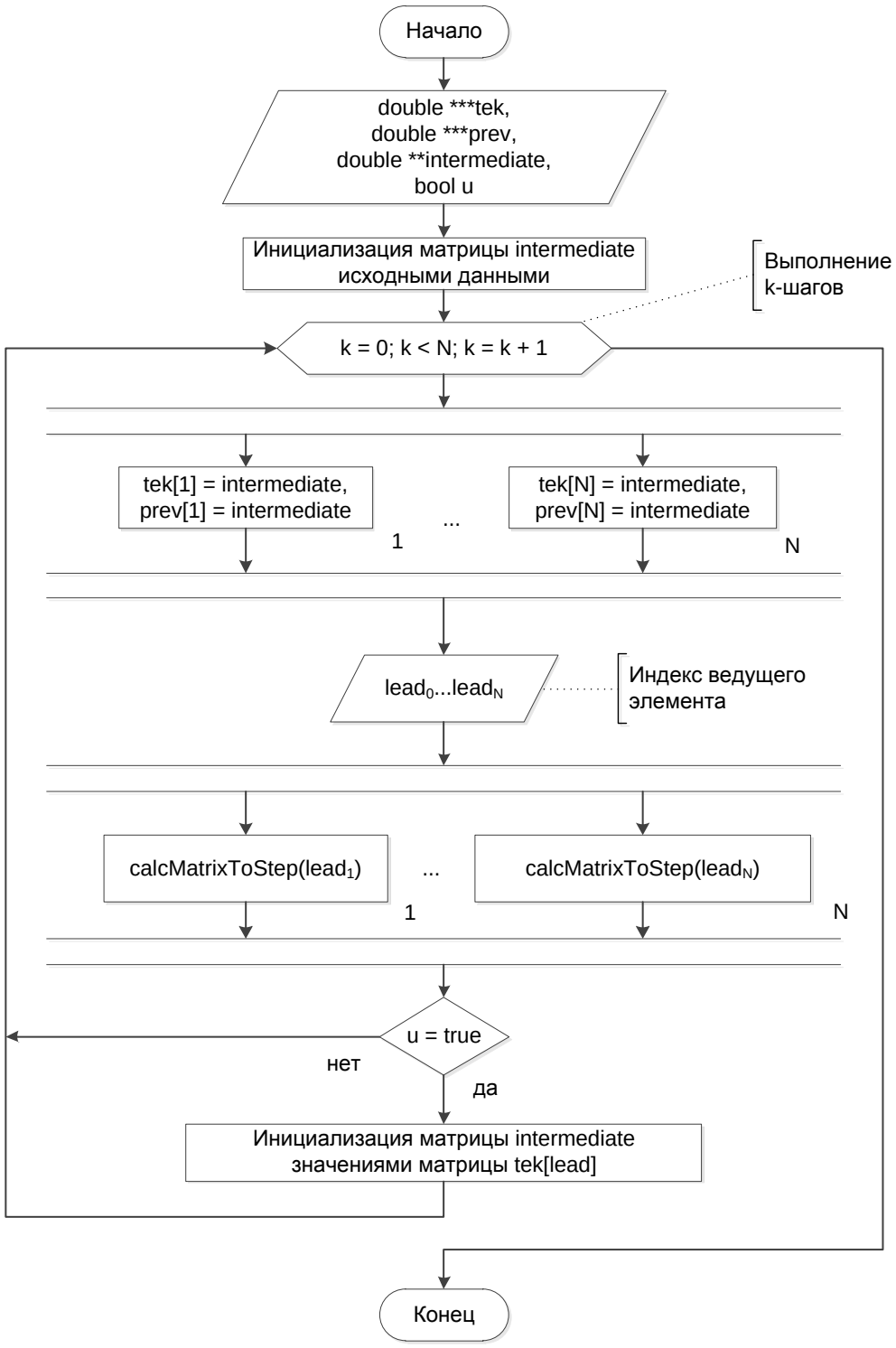

Рис. 3. Q-эффективная реализация алгоритма для общей памяти

Как показано на рис. 3, распараллеливание вычислений осуществляется для ведущих элементов. Данное распараллеливание называется распараллеливанием на верхнем уровне. Для одновременного вычисления матрицы и соответствующего ей логического Q-терма необходимо использовать вложенный параллелизм. На рис. 4 изображена схема параллельного вычисления матрицы и логического Q-терма.

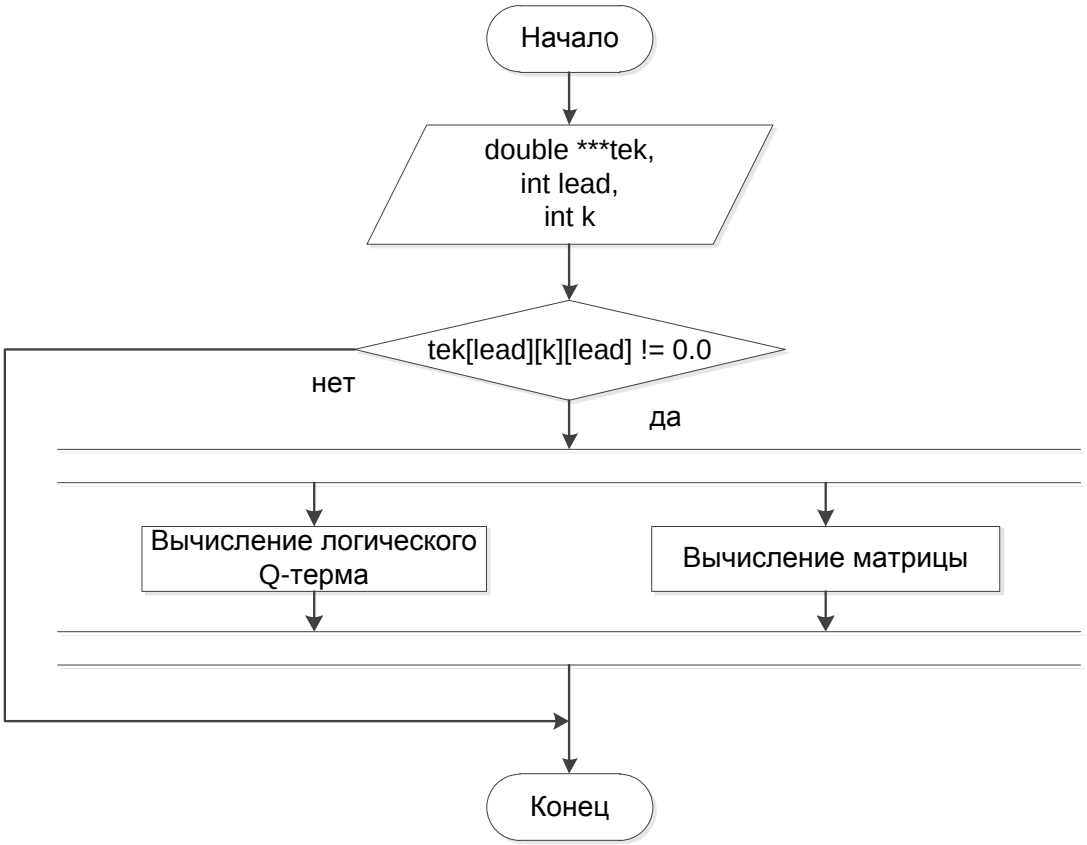

Рис. 4. Параллельное вычисление матрицы и соответствующего ей логического Q-терма

Для каждого ведущего элемента выполняется вычисление матрицы и соответствующего логического Q-терма при условии, что элемент является ненулевым.

Вычисление логического Q-терма осуществляется до тех пор, пока выполняется правило определения ведущего элемента. Схема вычисления логического Q-терма представлена на рис. 5.

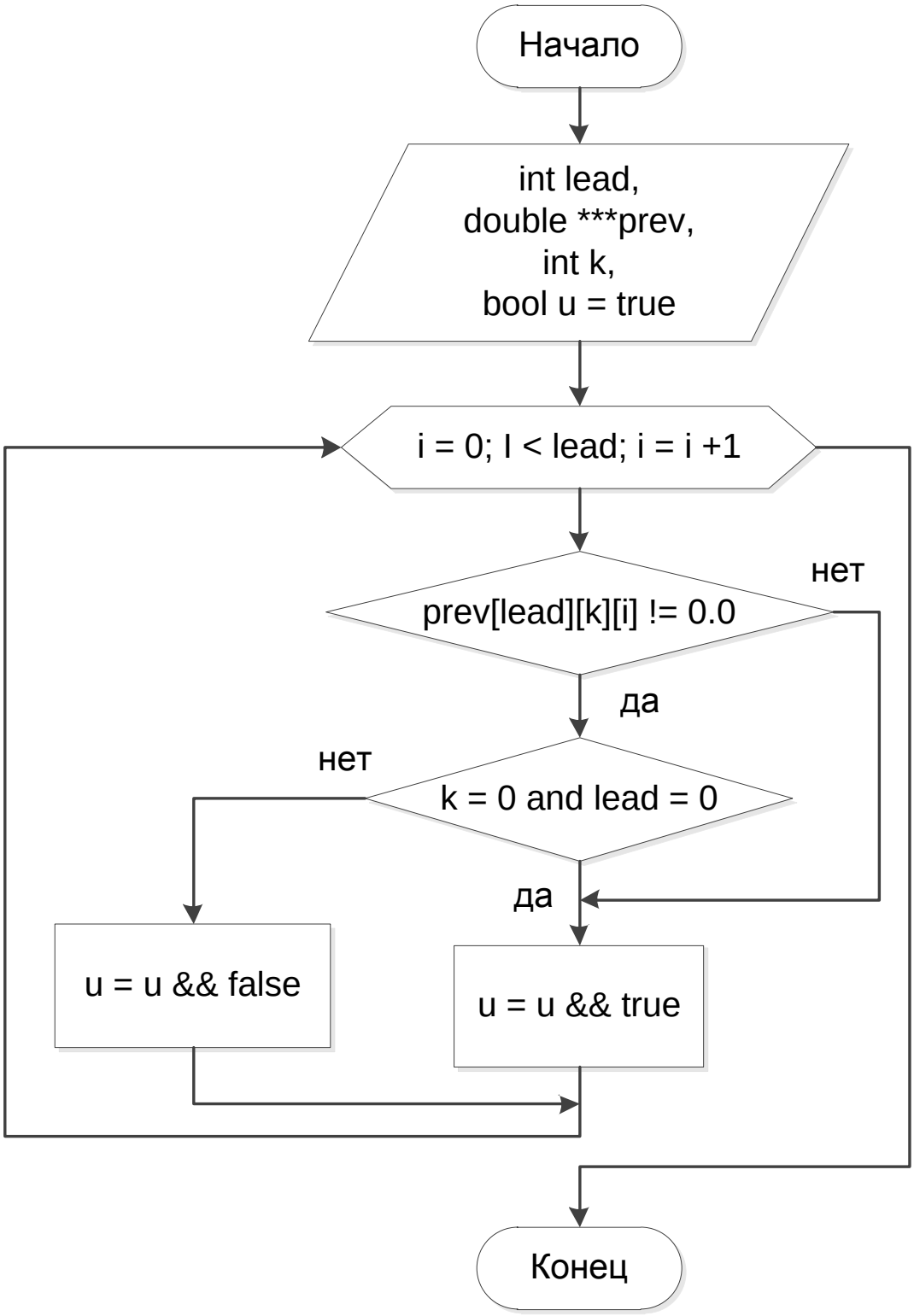

Рис. 5. Вычисление логического Q-терма

Одновременно с вычислением логического Q-терма выполняется вычисление матрицы. При вычислении матрицы используется дополнительная матрица, память для которой выделяется динамически. Схема вычисления матрицы представлена на рис. 6.

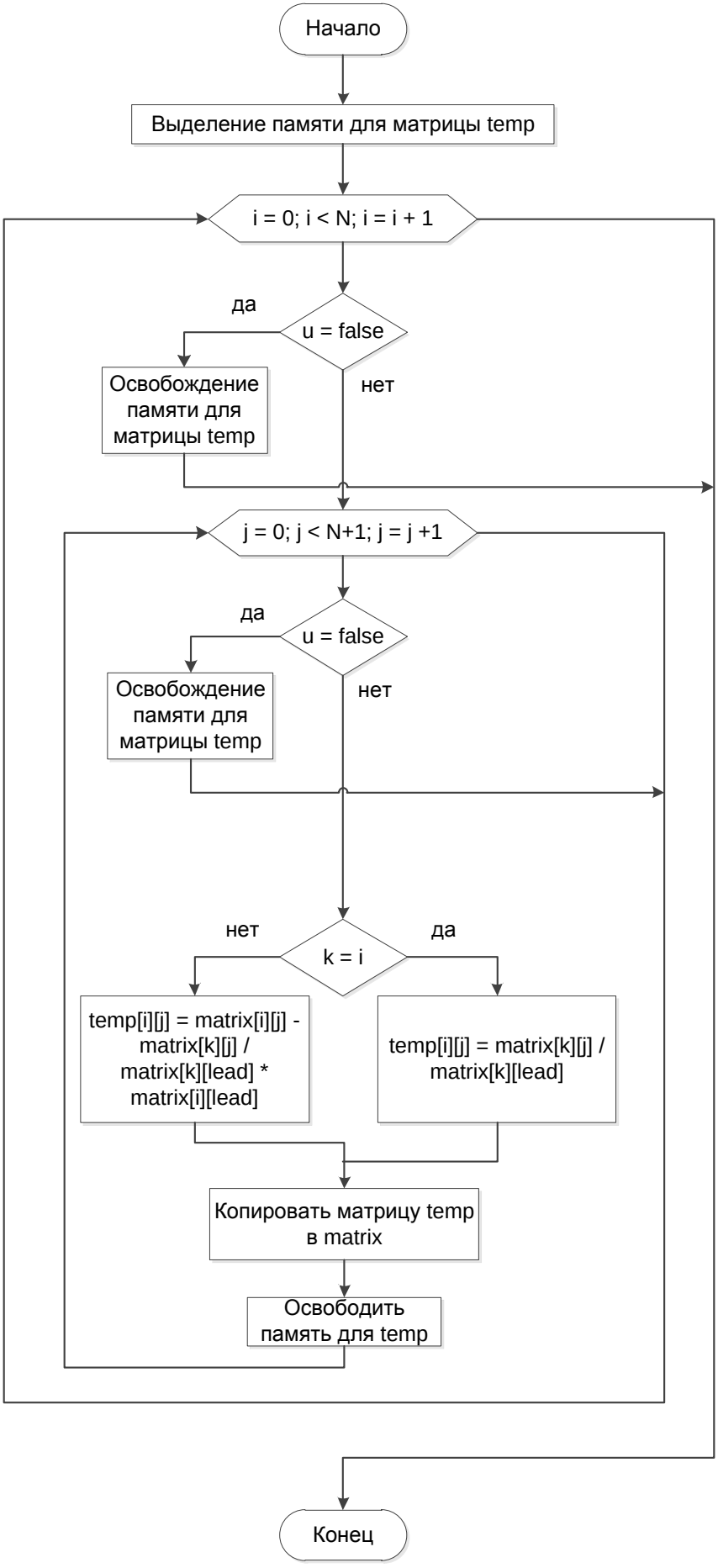

Рис. 6. Вычисление матрицы

2.3.2. Модель с распределенной памятью

Организация вычислений на распределенной памяти осуществляется по принципу «Мастер - Рабочие». Узел M передает узлам P информацию для вычисления матриц и соответствующих им Q-термов. Результаты вычислений передаются на узел M. Количество обменов равно 2n. На рис. 7 представлена схема «Мастер - Рабочие». В качестве узла M выступает Process 0, остальные узлы являются Рабочими. Процесс вычислений выполняется N раз, после каждого k-ого шага результаты вычислений на узлах P передаются на узел M.

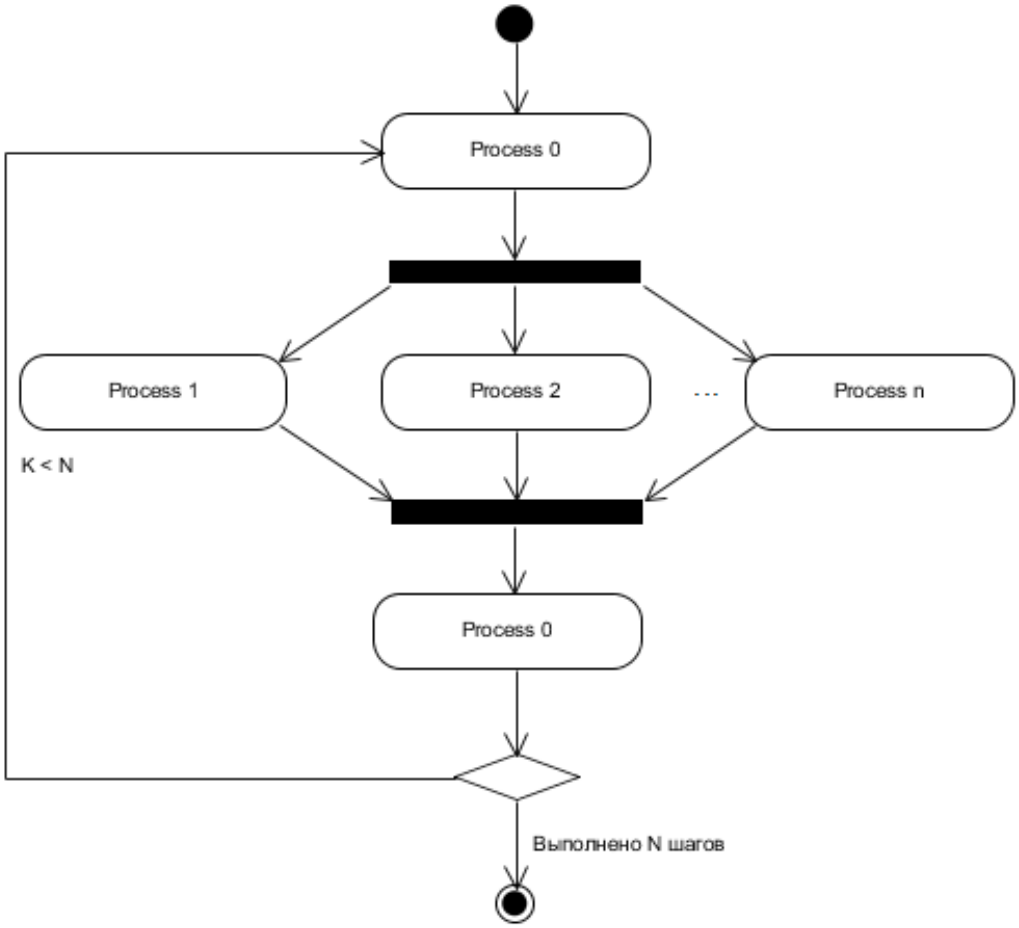

Рис. 7. Выполнение вычислений по принципу «Мастер – Рабочие»

На каждом узле параллельно выполняются вычисление матрицы и логического Q-терма. Если в процессе вычисления логический Q-терм принимает значение *true*, то результат вычисления передается на узел M. Схема вычисления представлена на рис. 8.

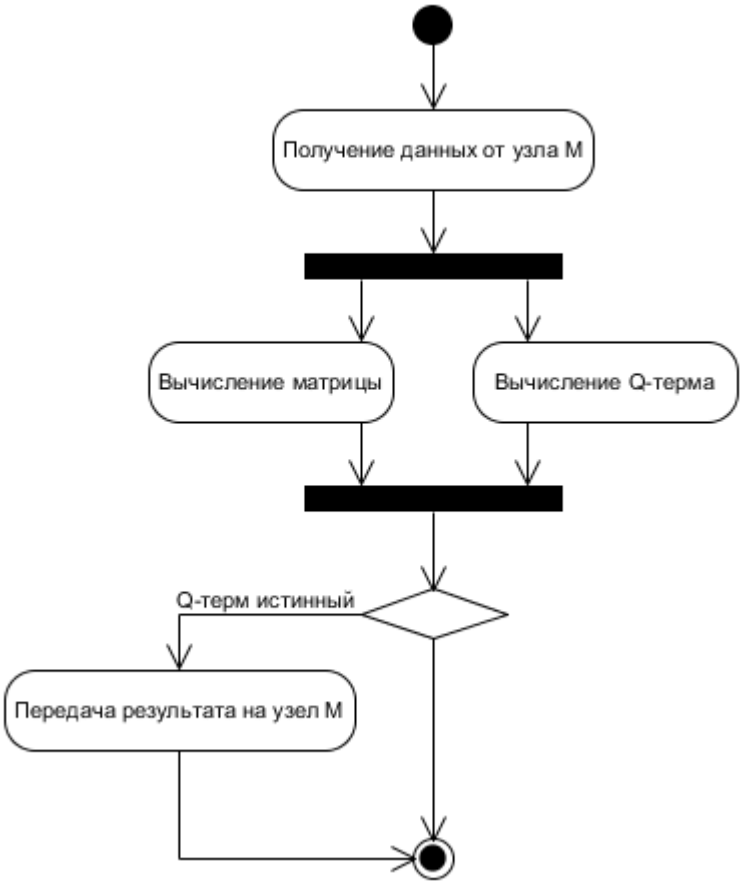

Рис. 8. Вычисление матрицы и логического Q-терма на одном узле

Получены Q-эффективные программы для общей и распределенной памяти. Исходные тексты программ представлены в приложении.

3. ТЕСТИРОВАНИЕ Q-ЭФФЕКТИВНЫХ ПРОГРАММ ДЛЯ МЕТОДА ГАССА-ЖОРДАНА

При тестировании на суперкомпьютере «Торнадо ЮУрГУ» разработанных Q-эффективных программ для метода Гаусса-Жордана были получены результаты, представленные в таблице 3.

Табл. 3. Результаты тестирования Q-эффективных реализаций

Размер	Время выполнения последовательной реализации метода Гаусса-Жордана	Время выполнения Q-эффективной реализации алгоритма с использованием OpenMP	Время выполнения Q-эффективной реализации алгоритма с использованием MPI
5	0.0000005133	0.0001711000	0.0000369549
10	0.0000025663	0.0000371928	0.0000739098
25	0.0001344755	0.0001708710	0.0001709461
50	0.0010029205	0.0005127405	0.0005128384
100	0.0078755434	0.0039997681	0.0008780956
150	0.0301030124	0.0144795634	0.0014829636
200	0.0670853201	0.0282583488	0.0022838116
500	0.16100476824	0.0575428049	0.0035049915

Графическая зависимость размерности задачи от времени выполнения представлена на рис. 9.

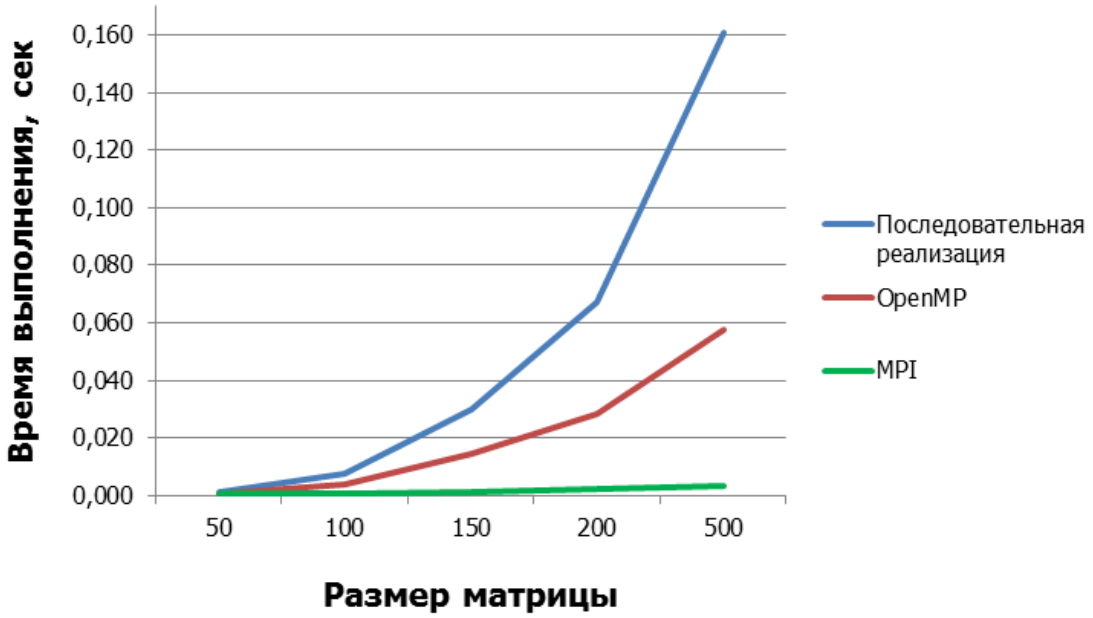

Рис. 9. Зависимость размерности задачи от времени выполнения

Для полученных результатов выполним расчет ускорения и эффективности.

Ускорением параллельного алгоритма называется отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма:

$$S = \frac{T_1}{T_p},$$

где: T_1 – время выполнения последовательной программы;

T_p – время выполнения параллельной программы на p процессорах

На рис. 10 представлен график зависимости ускорения от размера задачи для общей памяти.

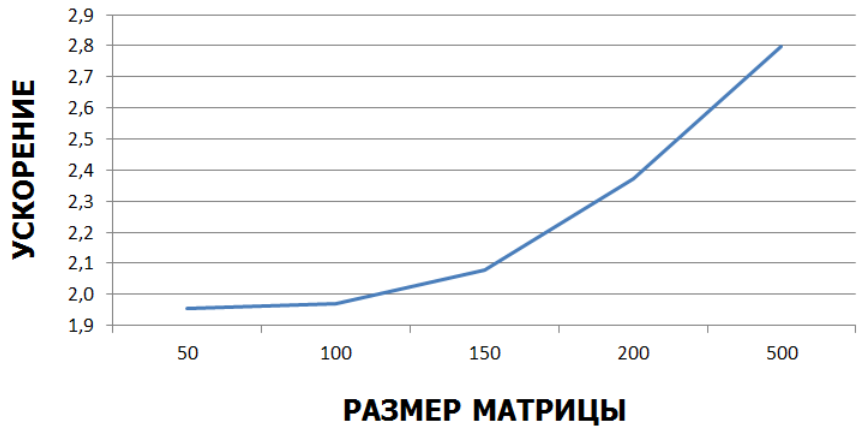

Рис. 10. Зависимость ускорения от размера задачи

Для оценки масштабируемости алгоритма используется понятие эффективности:

$$E = \frac{S}{P},$$

где: S – ускорение;

p – количество процессоров.

На рис. 11 представлен график зависимости эффективности от размера задачи для общей памяти.

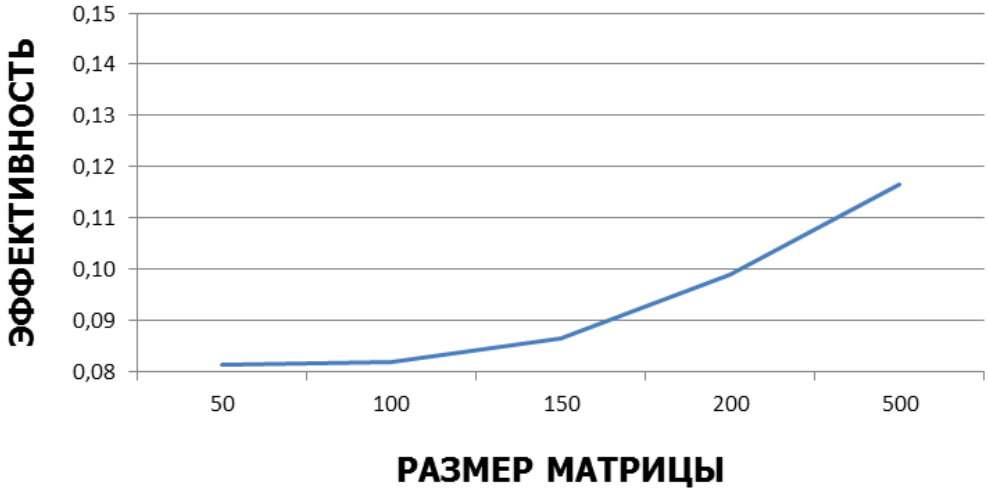

Рис. 11. Зависимость ускорения от размера задачи

Аналогичным образом построим графики зависимости ускорения и эффективности от размера задачи для распределенной памяти. На рис. 12 представлен график зависимости ускорения от размера задачи для распределенной памяти.

Рис. 12. Зависимость ускорения от размера задачи

На рис. 13 представлен график зависимости эффективности от размера задачи для распределенной памяти.

Рис. 13. Зависимость ускорения от размера задачи

На рис. 14 для матрицы размерности 50 построим график зависимости времени вычисления матрицы и логического Q-терма на k-ом шаге.

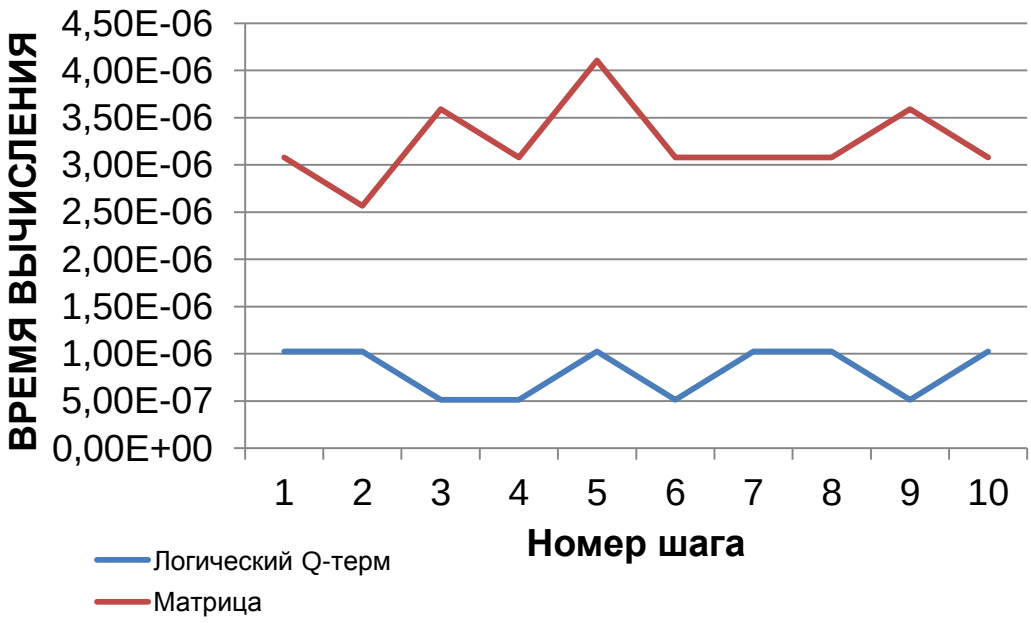

Рис. 14. Зависимость времени вычисления матрицы и логического Q-терма на k-ом шаге

На графике видно, что время вычисления логического Q-терма на порядок меньше времени вычисления матрицы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был выполнен Q-эффективный кодизайн реализации алгоритма Гаусса-Жордана для решения СЛАУ.

Для достижения поставленной цели были решены следующие задачи:

- изучен подход к распараллеливанию алгоритмов на основе их представления в форме Q-детерминанта;
- выполнено построение Q-детерминанта метода Гаусса-Жордана;
- изучена архитектура суперкомпьютера «Торнадо ЮУрГУ»;
- выполнена Q-эффективная реализации метода Гаусса-Жордана на общей памяти с использованием технологии распараллеливания OpenMP;
- выполнена Q-эффективной реализации метода Гаусса-Жордана на распределенной памяти с использованием технологии распараллеливания MPI + OpenMP;
- выполнено тестирование Q-эффективных программ.

Результаты исследования были апробированы на XI международной научной конференции ПаВТ’2017 [14].

Работа выполнена при поддержке гранта Российского Фонда Фундаментальных Исследований, грант № 17-07-00865.

СПИСОК ЛИТЕРАТУРЫ

1. Aleeva V.N., Sharabura I.S., Suleymanov D.E. Software System for Maximal Parallelization of Algorithms on the Base of the Conception of Qdeterminant, 13th International Conference on Parallel Computing Technologies, PaCT 2015, Petrozavodsk, Russian Federation, 31 August - 4 September 2015; Proceedings. Lecture Notes in Computer Science,Vol. 9251. Springer, 2015. – P. 3-9.

2. Алеева В.Н. Анализ параллельных численных алгоритмов: Препринт №590. В надзаг.: ВЦ СО АН СССР. – Новосибирск, 1985. – 23 с.

3. Алеева В.Н. Разработка принципов выполнения Q-эффективных реализаций численных алгоритмов на параллельных вычислительных системах. / В.Н. Алеева, Н.В. Валькевич, Ю.С. Лаптева, Д.Е. Тарасов. // Параллельные вычислительные технологии – XI международная конференция, ПаВТ’2017, г. Казань, 3–7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017. - С. 503.

4. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

5. Воеводин, Вл.В. Параллельные алгоритмы под микроскопом / Вл.В. Воеводин // Доклад на международной конференции «Параллельные вычислительные технологии (ПаВТ’2016)» (28 марта - 1 апреля 2016 г., г. Архангельск).

6. Гергель В.П. Современные языки и технологии параллельного программирования: Учебник/Предисл.: В.А. Садовничий. – М.: Издательство Московского университета, 2012. – 408 с.

7. Игнатьев С.В. Определение ресурса параллелизма алгоритмов на базе концепции Q-детерминанта: Вып. квалиф. работа магистра прикладной математики и информатики: 010500.68 / [Электронный ресурс] URL: http://sp.susu.ru/student/masterthesises.html#2010 (дата обращения: 20.05.2017).

8. Официальный сайт проекта MPI. / [Электронный ресурс]

URL: https://www.open-mpi.org (дата обращения: 20.03.2017).

9. Официальный сайт проекта OpenMP. / [Электронный ресурс] URL: http://www.openmp.org (дата обращения: 15.03.2017).

10. Пользовательская документация суперкомпьютера «Торнадо ЮУрГУ». / [Электронный ресурс]
URL: http://supercomputer.susu.ru/users/support/
(дата обращения 15.04.2017).

11. Сулейманов Д.Э. Разработка программной системы для максимального распараллеливания алгоритмов на основе концепции Q-детерминанта. Вып. квалиф. работа магистра: 010300.68 / [Электронный ресурс]
URL:http://omega.sp.susu.ru/publications/masterthesis/15-Suleymanov.pdf
(дата обращения: 20.05.2017).

12. Сулейманов Д.Э., Алеева В.Н., Шарабура И.С. Разработка программной системы QStudio для получения максимально быстрой реализации алгоритма // Параллельные вычислительные технологии (ПаВТ'2015): труды международной научной конференции (31 марта - 2 апреля 2015 г., г. Екатеринбург).Челябинск: Издательский центр ЮУрГУ, 2015. - С. 523.

13. Теплов А.М. Анализ масштабируемости параллельных приложений на основе технологий суперкомпьютерного кодизайна: Диссертация на соискание ученой степени к.ф.-м.н. аспиранта: 05.13.11 / [Электронный ресурс]
URL: http://www.srcc.msu.su/nivc/sci/dissert/aref/2015-11-13-teplov.pdf (дата обращения: 15.04.2017).

14. Шарабура И.С. Разработка программного обеспечения для анализа ресурса распараллеливания алгоритмов сортировки на основе концепции Q-детерминанта: Курсовая работа по дисциплине «Программная инженерия». – Челябинск: Южно-Уральский государственный университет, 2014. – 22 с.

ПРИЛОЖЕНИЕ

Приложение 1

Текст программы для общей памяти

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

double ***allocationThree(int size){
 double ***temp;
 temp = (double ***)malloc(size * sizeof(double**));

 for(int i = 0; i < size; i++){
  temp[i] = (double**)malloc(size * sizeof(double*));
 }

 for(int i = 0; i < size; i++){
  for(int j = 0; j < size; j++){
   temp[i][j] = (double*)malloc((size+1) * sizeof(double));
  }
 }

 return temp;
}

void closeThree(double ***arr, int size){
 for(int i = 0; i < size; i++){
  for(int j = 0; j < size; j++){
   free(arr[i][j]);
  }
 }

 for(int i = 0; i < size; i++){
  free(arr[i]);
 }

 free(arr);
}

double **allocationTwo(int size){
 double **temp;
 temp = (double**)malloc(size * sizeof(double*));
 for(int i = 0; i < size; i++){
  temp[i] = (double*)malloc((size+1) * sizeof(double));
 }

 return temp;
}

void closeTwo(double **arr, int size){
 for(int i = 0; i < size; i++){
  free(arr[i]);
 }
 free(arr);
}

void printMatrix(double **arr, int size){
 for(int i = 0; i < size; i++){
  for(int j = 0; j < size + 1; j++){
   printf("%.2lf ", arr[i][j]);
  }
```

```
  printf("\n");
 }
}

void copyMatrix(double **in, double **out, int size){
 for(int i = 0; i < size; i++){
  for(int j = 0; j < size + 1; j++){
   out[i][j] = in[i][j];
  }
 }
}

void calcMatrixToStep(double **matrix, double **temp, int size, int k, int
lead, bool *u){
 for(int i = 0; i < size; i++){
  if(*u == false) break;
  for(int j = 0; j < size + 1; j++){
   if(*u == false) break;

   if(k == i){
    //для избежания получения отрицательного нуля
    if(matrix[k][j] == 0.0){
     temp[i][j] = 0.0;
     continue;
    }

    temp[i][j] = matrix[k][j] / matrix[k][lead];
   }
   else
    temp[i][j] = matrix[i][j] - matrix[k][j] / matrix[k][lead] * ma-
trix[i][lead];
  }
 }

 copyMatrix(temp, matrix, size);
}

void calc(double **arr, int size){
 double ***tek = allocationThree(size);
 double ***prev = allocationThree(size);
 double **intermediate = allocationTwo(size);
 double **temp;
 bool *u;
 double t1, t2, t3, t4;

 copyMatrix(arr, intermediate, size);

 for(int k = 0; k < size; k++){
#pragma omp parallel num_threads(size)
  {
#pragma omp for schedule(static, 1)
   for(int i = 0; i < size; i++){
    copyMatrix(intermediate, tek[i], size);
    copyMatrix(intermediate, prev[i], size);
   }
  }

#pragma omp parallel num_threads(size)
  {
#pragma omp for schedule(static, 1) private(temp, u)
   for(int lead = 0; lead < size; lead++){
    temp = allocationTwo(size);
    u = new bool;
```

```
    if(tek[lead][k][lead] != 0.0){
#pragma omp parallel num_threads(2)
     {
      if(omp_get_thread_num() == 0){
       t1 = omp_get_wtime();
       for(int i = 0; i < lead; i++){
        if(prev[lead][k][i] != 0.0){
         if(k == 0 && lead == 0)
          *u = *u && true;
         else
          *u = *u && false;
        }
       }
       t2 = omp_get_wtime();
      }
      else{
       t3 = omp_get_wtime();
       calcMatrixToStep(tek[lead], temp, size, k, lead, u);
       t4 = omp_get_wtime();
      }
     }

     if(u){
      copyMatrix(tek[lead], intermediate, size);
      //printf("k - %d, lead - %d\n", k, lead);
      //printf("u: %.20lf\nw: %.20lf\n", t2-t1, t4-t3);
     }

     delete u;
     closeTwo(temp, size);

    }
   }//end for
  }//end parallel
  //printMatrix(intermediate, size);
  //printf("\n");
 }//end k

 closeTwo(intermediate, size);
 closeThree(tek, size);
 closeThree(prev, size);
}

int main(){

 omp_set_nested(1);

 int sizes[] = {5, 10, 25, 50, 100};

 for(int s = 0; s < sizeof(sizes)/sizeof(sizes[0]); s++){
  int size = sizes[s];
  printf("size: %d\n", size);
  double **test = allocationTwo(size);

  for(int i = 0; i < size; i++){
   for(int j = 0; j < size+1; j++){
    if(i == j)
     test[i][j] = 1.0;
    else
     test[i][j] = 0.0;
    if(j == size)
     test[i][j] = (double)(i * j + 4);
   }
  }
```

```
  //printMatrix(test, size);
  //printf("\n");
  double t1 = omp_get_wtime();
  calc(test, size);
  double t2 = omp_get_wtime();
  printf("s: %d, t = %.20lf\n", s, t2-t1);

  closeTwo(test, size);

  printf("\n\n");
 }

 return 0;
}
```

Приложение 2

Текст программы для распределенной памяти

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

double *matrix, *vector, *result, *matrix_temp, *vector_temp, *result_temp,
*m;
int  size = 500;
int p=1, status, str_count;
double  start, finish;
int *k;
int *lead;
int numproc, rank;

void init()

{
 matrix = new double [size*size];
 vector = new double [size];
 result = new double [size];
 int nstrings;

 MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);

 nstrings = size;

 for (int i=0; i<rank; i++)
  nstrings = nstrings-nstrings/(numproc-i);

 str_count = nstrings/(numproc-rank);
 matrix_temp = new double [str_count*size];
 vector_temp = new double [str_count];
 result_temp = new double [str_count];
 k = new int [size];
 lead = new int [str_count];

 for (int i=0; i<str_count; i++)
  lead[i] = -1;
}

void dist()
{
 int *firstsendindex;
 int *sendrange;
 int CountProcessstrings;
 int nstrings;
 firstsendindex = new int [numproc];
 sendrange = new int [numproc];

 MPI_Scatterv(matrix, sendrange, firstsendindex, MPI_DOUBLE, matrix_temp,
sendrange[rank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

 MPI_Bcast(&status, 1, MPI_INT, 0, MPI_COMM_WORLD);
 MPI_Bcast(&m, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 delete [] sendrange;
 delete [] firstsendindex;
}


void StoE(int IterationNumber, double *globmaxSystemstring)
```

```
{
 double delen;
 for (int i=0; i<str_count; i++)
 {
  if (lead[i] == -1)
  {
   for (int j=IterationNumber; j<size; j++)
   {
    matrix_temp[i*size + j] -= globmaxSystemstring[j]*delen;
   };
   vector_temp[i] -= globmaxSystemstring[size]*delen;
  };
 };
}

void calcMatrixToStep(double **matrix, double **temp, int size, int k, int
lead, bool *u){
 for(int i = 0; i < size; i++){
  if(*u == false) break;

  for(int j = 0; j < size + 1; j++){
      if(*u == false) break;

   if(k == i){
        if(matrix[k][j] == 0.0){
     temp[i][j] = 0.0;
     continue;
    }

    temp[i][j] = matrix[k][j] / matrix[k][lead];
   }
   else
    temp[i][j] = matrix[i][j] - matrix[k][j] / matrix[k][lead] * ma-
trix[i][lead];
  }
 }

 copyMatrix(temp, matrix, size);
}


double gauss_jordan_q(double **arr, int size){
  double ***tek = allocationThree(size);        double ***prev = alloca-
tionThree(size);
  double **intermediate = allocationTwo(size);
  double t1, t2;

//инициализаци¤ промежуточной матрицы
  copyMatrix(arr, intermediate, size);



  start_algorithm(tek, prev, intermediate, size);

  closeTwo(intermediate, size);
  closeThree(tek, size);
  closeThree(prev, size);

  return t2-t1;

}
void GaussJ()
{
 int VedIndex;     struct { double MaxValue; int rank; } Localmax, globmax;
```

```
 double* pglobmaxstring = new double [size+1];
 for (int i=0; i<size; i++)
 {

  double MaxValue = 0;
  int tmp_index = -1;
  for (int j=0; j<str_count; j++)
  {
   tmp_index = j;
   if ((lead[j] == -1)&&(MaxValue < fabs(matrix_temp[i+size*j])))
   {
    MaxValue = fabs(matrix_temp[i+size*j]);
    VedIndex = j;
   };
  };

  Localmax.MaxValue = MaxValue;
  Localmax.rank = rank;

  MPI_Allreduce(&Localmax, &globmax, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
MPI_COMM_WORLD);

  if ( rank == globmax.rank )
  {
   if (globmax.MaxValue == 0)
   {
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Send(&status, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD);
    lead[tmp_index] = i;
    continue;

   }
   else
   {
    lead[VedIndex] = i;
   };
  };
  MPI_Bcast(&k[i], 1, MPI_INT, globmax.rank, MPI_COMM_WORLD);
  if (rank == globmax.rank)
  {
   for (int j=0; j<size; j++)
    pglobmaxstring[j] = matrix_temp[VedIndex*size + j];
   pglobmaxstring[size] = vector_temp[VedIndex];
  };
  MPI_Bcast(pglobmaxstring, size+1, MPI_DOUBLE, globmax.rank,
MPI_COMM_WORLD);
 };
}

void GaussJordan()
{
 GaussJ();
}

void main(int argc, char* argv[])
{
 MPI_Init(&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 MPI_Comm_size (MPI_COMM_WORLD, &numproc);
 MPI_Barrier(MPI_COMM_WORLD);
 init();
 dist();
 MPI_Barrier(MPI_COMM_WORLD);
 start = MPI_Wtime();
```

```
 GaussJordan();
 MPI_Barrier(MPI_COMM_WORLD);
 finish = MPI_Wtime();
 MPI_Finalize();
 delete [] matrix;
 delete [] vector;
 delete [] result;
}
```