МИНИСТЕРСТВО ОБРАЗОВАНИЯ РФ
РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА АЛГЕБРЫ И ДИСКРЕТНОЙ МАТЕМАТИКИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на степень бакалавра по направлению подготовки
«Математика»

«Трехмерная визуализация решетчатых графов»

студентки 4-го курса П. В. Шатерниковой

Научный руководитель
д.т.н., доц. Б. Я. Штейнберг

г. Ростов-на-Дону
2006 г.

Введение

Большую роль в прогрессе общества играют параллельные суперкомпьютеры. Создание самолетов, автомобилей, исследование семантики ДНК, прогноз погоды и изменений климата невозможно без применения суперЭВМ. В последнее время принципы суперЭВМ все больше используются в персональных компьютерах, которые становятся все доступнее.

Для того чтобы программа выполнялась на суперкомпьютере быстрее, чем на последовательном компьютере, она должна быть специальным образом написана. В настоящее время разработано и разрабатывается большое количество средств создания параллельных программ.

Использование распараллеливающих компиляторов является наиболее удобным для пользователя т.к. в этом случае пользователь освобождается от необходимости изучения новых языков и систем программирования, архитектуры суперкомпьютера, а также от написания самой параллельной программы.

Разработка распараллеливающих компиляторов – одна из задач автоматического распараллеливания. Для её решения необходима детальная информация о структуре алгоритма, взаимодействии отдельных операций между собой и с памятью. Неотъемлемой частью получения этой информации является, так называемый, анализ информационных зависимостей программы. Такой анализ позволяет определять операции, итерации (циклов) и операторы в последовательной программе, которые могут быть исполнены независимо (одновременно, параллельно). Существует множество представлений информационных зависимостей, которые различаются по степени информативности. Одним из наиболее тонких представлений информационной зависимости является решетчатый граф (граф алгоритма [3]). В нашей стране методы распараллеливания, основанные на анализе решетчатого графа, активно исследуются академиком РАН В.В. Воеводиным и его учениками. Работать с этим графом, используя традиционные методы, такие как матрица смежности или список дуг, крайне сложно из-за его больших размеров: число вершин решетчатого графа не меньше, чем число исполняемых операций в программе.

В.В. Воеводин разработал алгоритм, позволяющий получить описание решетчатого графа в виде конечного числа аффинных отображений. Из других исследователей, внесших значительный вклад в теорию решетчатых графов, следует отметить П. Фотрье (P. Feautrier, Франция), который также разработал алгоритм построения решетчатого графа, но в виде квазиаффинного дерева решений [6, 7]. До появления указанных алгоритмов построения решетчатого графа, применимых к широкому классу программ, этот граф использовался лишь в простых частных случаях, например, в методе гиперплоскостей Лампорта и методах параллелепипедов и пирамид В.А. Вальковского.

В настоящий момент в Открытой распараллеливающей системе (ОРС) реализовано построение элементарных и простых снизу решетчатых графов для линейных программ, не содержащих внешних переменных и условных операторов.

Построение элементарного снизу решетчатого графа реализовано на основе параметрического целочисленного программирования, разработанного П. Фотрье [6].

Элементарный решетчатый граф в настоящий момент визуализирован в ОРС. Простой решетчатый граф строиться только в преобразованиях, и в настоящий момент не визуализирован.

До недавнего времени в ОРС был визуализирован лишь двумерный случай решетчатого графа, но очень часто эта размерность оказывается неподходящей для многих примеров. Поэтому был реализован уже и трехмерный случай, на который в дальнейшем планируется проектировать примеры, состоящие из четырех и более циклов.

Глава 1. Определение решетчатого графа

Решетчатый граф – одна из наиболее тонких моделей информационной зависимости. Решетчатый граф в частности позволяет для любой операции в правой части оператора присваивания указать итерацию и генератор, выход которого является аргументом данной операции.

1.1 Опорные пространства. Порядок исполнения операторов

Рассмотрим некоторый фрагмент программы из линейного класса. Занумеруем операторы циклов в этом фрагменте *сверху вниз по тексту программы*. Обозначим счетчик i-го цикла – I_i.

Определение. ([2]) Множество вложенных друг в друга циклов, в теле каждого из которых содержится некоторый оператор *Stmt*, называют *опорным гнездом* циклов для оператора *Stmt* и для всех вхождений, которые этот оператор содержит.

Определение. Множество значений вектора счетчиков циклов опорного гнезда, при которых исполняется оператор *Stmt*, будем называть *опорным пространством* для данного оператора и всех вхождений переменных, которые он содержит.

Определение. ([2]) Рассмотрим множество всех условий, задаваемых условными операторами, при которых исполняется оператор *Stmt*. Множество этих условий разбивает линейное пространство итераций опорного гнезда данного оператора на множество выпуклых многогранников. Объединение этих многогранников называется *линейным опорным пространством* оператора *Stmt*.

Если некоторый оператор присваивания не содержится в гнезде циклов, то можно считать, что этот оператор содержится в цикле, у которого верхняя граница равна нижней. Не умаляя общности, будем считать, что все операторы присваивания содержатся в гнездах циклов.

Далее будем считать, что в каждой строке программы находится только один оператор. Занумеруем все операторы присваивания фрагмента программы сверху

вниз по тексту программы и обозначим их через $Stmt_1$, $Stmt_2$, …, $Stmt_m$. Обозначим опорное пространство оператора $Stmt_i$ через V_i, размерность которого равна s_i. Подчеркнем, что пространства V_i и V_j при $i \neq j$ считаются различными, даже если они образованы одними и теми же операторами циклов и условными операторами.

Определение. ([2]) *Пространством итераций фрагмента программы* назовем совокупность опорных пространств V_i для i=1, 2, …, m.

Определение. ([2]) Будем называть исполнение оператора $Stmt_i$ при конкретных значениях счетчиков циклов его опорного гнезда $I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}$ *срабатыванием* и обозначать $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ или $Stmt_i[\boldsymbol{I}]$ считая, что вектор $\boldsymbol{I}$ равен $(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}})$.

Каждому срабатыванию $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ поставим в соответствие точку опорного пространства V_i с теми же значениями координат $(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}})$. Таким образом, устанавливается взаимнооднозначное соответствие между множеством точек пространства итераций фрагмента программы и множеством всех срабатываний всех операторов присваивания этого фрагмента. Заметим, что все точки одного опорного пространства V_i и только они соответствуют срабатываниям одного оператора $Stmt_i$.

Определение. Пусть имеются векторы $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$, и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$. Пусть $m=min(n1, n2)$. Тогда:

1. если существует $k \in [1, m]$, что $I_1 = J_1$, $I_2 = J_2$, …, $I_{k-1} = J_{k-1}$, $I_k < J_k$, то будем говорить, что вектор $\boldsymbol{I}$ *лексикографически меньше* вектора $\boldsymbol{J}$ ($\boldsymbol{I} <_{lex} \boldsymbol{J}$);
2. если $n1=n2$ и для любого $k \in [1, m]$ $I_k = J_k$, то будем говорить, что вектор $\boldsymbol{I}$ *лексикографически равен* вектору $\boldsymbol{J}$ ($\boldsymbol{I} =_{lex} \boldsymbol{J}$).

Зафиксируем некоторое число d, что $1 \le d \le m$. Если условие 1 выполняется для $k \in [1, d]$, то будем говорить, что вектор $\boldsymbol{I}$ *лексикографически меньше* вектора $\boldsymbol{J}$ *по первым d координатам* ($\boldsymbol{I} <^{d}_{lex} \boldsymbol{J}$). Если условие 2 выполняется для $k \in [1, d]$, то

будем говорить, что вектор $\boldsymbol{I}$ *лексикографически равен* вектору $\boldsymbol{J}$ *по первым* d *координатам* ($\boldsymbol{I}=^{d}_{lex}\boldsymbol{J}$).

При выполнении линейной программы на «последовательном» компьютере все ее срабатывания $Stmt_i[\boldsymbol{I}]$ выполняются в строго определенном порядке. Для его описания введем отношение порядка в пространстве итераций фрагмента программы. Этот порядок будем также называть *лексикографическим*, и обозначать тем же символом $<_{lex}$.

Рассмотрим любую пару различных точек $\boldsymbol{I}=(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}) \in V_i$ и $\boldsymbol{J}=(I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}) \in V_j$. Будем говорить, что $\boldsymbol{I}$ *лексикографически меньше* $\boldsymbol{J}$ ($\boldsymbol{I}<_{lex}\boldsymbol{J}$), если:

1. либо $i<j$ и пересечение совокупностей номеров i_1, i_2, …, i_{s_i} и j_1, j_2, …, j_{s_j} пусто; в этом случае операторы $Stmt_i$ и $Stmt_j$ не имеют общих циклов, и оператор $Stmt_i$ находится по тексту программы раньше, чем оператор $Stmt_j$;
2. либо $i<j$ и $\boldsymbol{I}=^{d}_{lex}\boldsymbol{J}$, где d ($d\geq 1$) – количество общих циклов для операторов $Stmt_i$ и $Stmt_j$; в этом случае $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ и $Stmt_j[I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}]$ исполняются на одной итерации общего гнезда циклов, и $Stmt_i$ находится по тексту раньше, чем $Stmt_j$;
3. либо $\boldsymbol{I}<^{d}_{lex}\boldsymbol{J}$, где d ($d\geq 1$) – количество общих циклов для операторов $Stmt_i$ и $Stmt_j$; в этом случае $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ исполняется на более ранней итерации, чем $Stmt_j[I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}]$.

Таким образом, если $\boldsymbol{I}\in V_i$ лексикографически меньше $\boldsymbol{J}\in V_j$, то при последовательном исполнении фрагмента линейной программы срабатывание $Stmt_i[\boldsymbol{I}]$ произойдет раньше, чем срабатывание $Stmt_j[\boldsymbol{J}]$.

1.2 Максимальные и минимальные решетчатые графы

Введем определение максимального решетчатого графа [2, с. 346] фрагмента программы.

Определение. ([2]) Вершины *максимального решетчатого графа* – пространство итераций фрагмента программы. Дуга направляется из вершины $\boldsymbol{I}\in V_i$ в вершину $\boldsymbol{J}\in V_j$, если $\boldsymbol{I}<_{lex}\boldsymbol{J}$ и существуют такие вхождения $u\in Stmt_i$ и $v\in Stmt_j$, что $u[\boldsymbol{I}]$ и $v[\boldsymbol{J}]$ обращаются к одной и той же ячейке памяти. При этом говорят, что вхождение *u* определяет начало дуги, а вхождение *v* – конец дуги.

Будем считать, что каждая пара вхождений (*u*, *v*) определяет отдельную дугу из ***I*** в ***J***. Таким образом, максимальный решетчатый граф может иметь кратные дуги.

Из определения следует, что максимальный решетчатый граф есть представление информационной зависимости по памяти.

Определение. ([2]) Вершины *максимального решетчатого графа потоковой зависимости* – пространство итераций фрагмента программы. Дуга направлена из вершины ***I*** в вершину ***J*** ($\boldsymbol{I}\in V_i$, $\boldsymbol{J}\in V_j$, $\boldsymbol{I}<_{lex}\boldsymbol{J}$), если существуют генератор $u\in Stmt_i$ и использование $v\in Stmt_j$, что $u[\boldsymbol{I}]$ и $v[\boldsymbol{J}]$ обращаются к одной ячейке памяти.

Аналогичным образом можно ввести максимальные решетчатые графы выходной, анти- и входной зависимости.

Определение *минимальных решетчатых графов* дадим конструктивно ([2]).

Возьмем максимальный решетчатый граф зависимости типа *T*. Для каждой вершины *s* этого графа выполним:

1. разобьем множество дуг, *входящих в вершину s* на группы: к одной группе отнесем дуги, конец которых определялся одним и тем же вхождением;
2. в каждой группе выберем дугу, у которой *начальная* вершина лексикографически ближе всего к вершине *s*; остальные дуги удалим.

Полученный граф называется *минимальным снизу решетчатым графом* зависимости типа *T* [2, с. 346].

Снова возьмем максимальный решетчатый граф зависимости типа *T*. Для каждой вершины *s* этого графа выполним:

3. разобьем множество дуг, *выходящих из вершины s* на группы: к одной группе отнесем дуги, начало которых определялось одним и тем же вхождением;

4. в каждой группе выберем дугу, у которой *конечная* вершина лексикографически ближе всего к вершине *s*; остальные дуги удалим.

Полученный граф называется *минимальным сверху решетчатым графом* зависимости типа *T* [2, с. 347].

Для зависимостей по выходу минимальные снизу и сверху решетчатые графы совпадают [2, с. 347]. Совпадают между собой минимальные снизу и сверху решетчатые графы для зависимости по входу [2, с. 347]. В общем случае для потоковой и антизависимости минимальные сверху и снизу решетчатые графы различаются [2, с. 347].

1.3 Простые и элементарные решетчатые графы

Для того, чтобы получить алгоритмы построения минимальных решетчатых графов, вводится понятие простого и элементарного решетчатого графа [2, с. 351]. Эти понятия более удобны при использовании, чем понятие минимального решетчатого графа, они будут широко использоваться в данной работе.

Рассмотрим программу из линейного класса. Возьмем какую-либо дугу (***I***, ***J***) минимального снизу решетчатого графа потоковой зависимости. Ясно, что не все вхождения переменных в программу участвуют в определении данной дуги. В определении этой дуги участвует использование, читающее из памяти на итерации ***J***, и генераторы той же переменной, что и указанное использование.

Исходя из данной программы, построим новую программу по следующим правилам. Зафиксируем какое-нибудь использование некоторой переменной *X* в операторе *Stmt*. В операторе *Stmt* вычеркнем всю правую часть кроме зафиксированного вхождения. Во всех остальных операторах присваивания полностью вычеркнем правые части. В левых частях операторов присваивания вычеркнем все вхождения переменных, имена которых отличны от *X*. Теперь вычеркнем все операторы, у которых оказались пустыми левые и правые части, все условные операторы, охватывающие только пустые операторы, и все циклы с пустыми телами.

С формальной точки зрения оставшаяся часть программы не является программой на языке С, т.к. левые и правые части некоторых операторов могут быть пустыми. Эту часть программы можно сделать программой на языке С, если пустые части заменить вхождениями отличных между собой переменных (чтобы не возникло новых зависимостей), не совпадающих с X.

Любая программа такого типа называется *простой*. Построенный для нее минимальный снизу решетчатый граф называется *простым снизу решетчатым графом*. Далее в работе, если специально не оговорено, под *простым решетчатым графом* будем понимать простой снизу решетчатый граф.

Каждый простой решетчатый граф описывает зависимость одного фиксированного вхождения от набора других вхождений. Поэтому далее в работе простой решетчатый граф будет идентифицироваться вхождениями, зависимость между которыми он описывает.

Утверждение 1. ([2, с. 353]) Для любой программы из линейного класса минимальный решетчатый граф любого типа есть объединение минимальных графов того же типа всех простых программ.

Процесс расщепления линейных программ и соответствующих им минимальных решетчатых графов можно продолжить. В получившейся ранее простой программе вычеркнем все операторы присваивания, кроме оператора, содержащего фиксированное использование переменной X, и какого-нибудь оператора, содержащего генератор переменной X. В частности, второй оператор может совпадать с первым. Теперь вычеркнем все условные операторы, охватывающие только пустые операторы, и все циклы с пустыми телами.

Любая программа такого типа называется *элементарной* [2]. Построенный для нее минимальный снизу решетчатый граф называется *элементарным снизу решетчатым графом* [2]. Далее в работе, если специально не оговорено, то под элементарным решетчатым графом будем понимать элементарный снизу решетчатый граф.

Далее в работе элементарный решетчатый граф будет идентифицироваться вхождениями, зависимость между которыми он описывает.

Утверждение 2. ([2, с. 354]) Для любой программы из линейного класса минимальный решетчатый граф любого типа есть остовный подграф объединения минимальных графов того же типа всех элементарных программ.

1.4 Представление решетчатых графов

По способам представления (хранения) нет принципиальной разницы между минимальными снизу и сверху решетчатыми графами. Поэтому далее в этой части будем рассматривать только минимальные снизу решетчатые графы, которые в отсутствии оговорок будем называть минимальными решетчатыми графами.

Для хранения минимального решетчатого графа не подходят такие традиционные методы как матрица смежности или список дуг графа. Это происходит из-за того, что множество вершин минимального решетчатого графа равно итерационному пространству программы. Затраты памяти на хранение такого количества вершин или дуг для реальных программ не приемлемы. Кроме того, время просмотра такого графа будет сопоставимо со временем исполнения программы, что также недопустимо. Для представления минимальных решетчатых графов используются другие методы.

Для упрощения изложения, далее в этой части без дополнительных оговорок будем рассматривать решетчатые графы только потоковой зависимости.

Любой минимальный решетчатый граф может быть представлен как объединение всех простых решетчатых графов (утв. 1). Простой решетчатый граф можно получить из объединения элементарных решетчатых графов, удалив некоторые дуги (утв. 2). И в простом и в элементарном решетчатом графе в одну вершину может входить не более одной дуги. Кроме этого, как было указано ранее, конечная вершина дуги используется для определения начальной вершины дуги. Отсюда возникает идея представлять (хранить) дуги простого и элементарного решетчатых графов в виде функции F, аргумент у которой – вершина конца дуги, а возвращаемое значение – вершина начала дуги. Эта функция определена на множестве вершин решетчатого графа H, в которые входят дуги.

В общем случае функция F является кусочной. Поэтому функцию F представляют в виде набора функций F_k (k=1..n), которые определены на множествах H_k (k=1..n). Множества H_k (k=1..n) образуют разбиение множества H. Далее в работе будем говорить о наборе функций, описывающих дуги решетчатого графа. Вершины графа представляются как все целые точки некоторых областей H_k, каждая из которых задана системой неравенств.

Дуги элементарного решетчатого графа, описывающего зависимость использования $v \in Stmt_j$ от генератора $u \in Stmt_i$, представляются (хранятся) в виде набора функций F_k (k=1..n). Эти функции определены на подмножествах H_k (k=1..n) опорного пространства V_j, в которые входят дуги решетчатого графа. Функция F_k, определенная на H_k, ставит в соответствие вершине $\boldsymbol{J} \in H_k$ начальную вершину $\boldsymbol{I} \in V_i$ единственной дуги входящей в $\boldsymbol{J}$.

Часто в реальных программах, дуги элементарного решетчатого графа могут быть описаны функциями, для которых выполняются условия:

C.1) функции имеют вид $\boldsymbol{I}=\boldsymbol{A}*\boldsymbol{J}+\boldsymbol{b}$, где $\boldsymbol{A}$ – числовая матрица, вектор $\boldsymbol{b}$ в общем случае линейно зависит от внешних переменных;

C.2) функции заданы на всех целых точках некоторых выпуклых линейных многогранников;

C.3) количество функций и выпуклых линейных многогранников, на которых задана отдельная функция, конечно, не зависит от значений внешних переменных и не пропорционально размерам опорного пространства.

Если исходный фрагмент программы содержит внешние переменные, то функции, описывающие дуги некоторого решетчатого графа для этого фрагмента, могут быть параметризованны этими переменными.

Однако существуют программы, принадлежащие линейному классу, в которых дуги элементарного решетчатого графа некоторой зависимости не могут быть описаны функциями, удовлетворяющими условиям C.1) – C.3).

<u>Определение.</u> ([7]) *Квазиаффинная форма* – это форма, полученная из параметров и целочисленных констант с помощью операций сложения, умножения на целое, деления на целое.

Дуги элементарного решетчатого графа любой зависимости в любой линейной программе могут быть описаны функциями, заданными с помощью квазиаффинных форм [6], [7] (некоторые из форм, описывающих функции или/и их области определения, есть квазиаффинные формы).

Так как простой решетчатый граф есть объединение элементарных решетчатых графов (с отбрасыванием некоторых дуг), то простые решетчатые графы представляются (хранятся) аналогично элементарным решетчатым графам. Все вышесказанное относительно вида функций, описывающих дуги элементарного графа, относится и к простому решетчатому графу. Единственно, с каждой функцией F_k, описывающей дуги простого решетчатого графа, связывается генератор, записи в память которого определяют начала дуг, описываемых F_k. Аналогично описывается и минимальный решетчатый граф. С каждой функцией F_l этого графа связывается пара вхождений, обращения к памяти которых порождают дуги графа, описываемые функцией F_l.

Глава 2. Средства трехмерной визуализации.

Визуализирован трехмерный решетчатый граф в Microsoft Visual Studio .NET 2003 при помощи подключенной библиотеки OpenGL, а именно: opengl32.lib, glu32.lib, glaux.lib. Данные библиотеки помогли построить трехмерные объекты. Для этого использовались следующие функции и процедуры из них:

- Построение вершин осуществлялось при помощи сферы auxSolidSphere(r).
- Стрелки рисовались конусом auxSolidCone(r,height), цилиндром auxSolidCylinder(r,height), тором auxSolidTorus(r,R).
- Кроме того, в создании стрелки использовалась обычная линия GL_LINES, которая задается при помощи указания вершин начала и конца.
- Поскольку все фигуры рисуются в начале координат и в строго определенном положении относительно координат, то для того, чтобы нарисовать их в другом месте пространства и ином положении,

необходимо использовать функции переноса glTranslated($\Delta x, \Delta y, \Delta z$) и поворота glRotated(φ, x_0, y_0, z_0) системы координат.

- А для того, чтобы запомнить и вернуться к старым координатам, используются glPushMatrix() и glPopMatrix() соответственно.
- Чтобы нарисовать тор урезанным используются плоскости отсечения glClipPlane(GL_CLIP_PLANE0, equation).
- Цвет объектам задается функцией glColor3d(c1,c2,c3).
- Для освещения объектов, чтобы их возможно было увидеть, применяются лампы glLightfv(GL_LIGHT0, GL_POSITION, pos).

Глава 3. Описание программы и её возможностей

После построения Шульженко А.М. элементарного снизу решетчатого графа программа получает на вход список дуг LGV.RowList этого графа. И уже из этого списка дуг LGV.RowList получаем координаты вершин начала и конца каждой дуги, занося соответственно их в массивы matr1[ii][3] и matr2[ii][3].

Далее рисуем пространство итераций и прямоугольную систему координат, позволяющую ориентироваться в нашем пространстве. После чего начинаем построение дуг. Происходит это следующим образом.

Берем координаты вершин начала и конца дуги. Переносим систему координат в вершину начала дуги и подаем номер дуги в функцию choice_of_arc(in).

void LatticeGraphVisual3d::choice_of_arc(int str) - эта функция позволяет:

- повернуть систему координат так, чтобы вершина конца дуги лежала на оси Oz на положительном луче;
- выбрать какой вид дуги из трех возможных будем использовать в данном случае. На данный момент реализованы следующие виды дуг: петля, прямая и изогнутая стрелки.

Для этого при помощи массива razn[3] , в который заносятся разности соответствующих координат вершин конца и начала дуги, находятся расстояние dl между вершинами и требуемые углы поворота системы

координат по известным формулам геометрии. Выбирается тип дуги следующим образом:

- void LatticeGraphVisual3d::ring() – если все значения массива razn нулевые, то строится петля, которая рисуется тором и конусом. Поскольку петля – это дуга, которая входит в ту же вершину, из которой вышла.
- void LatticeGraphVisual3d::arrow(double dlin,int i1,int j1,int k1) – если расстояние между вершинами дуги равно единице, то рисуется прямая стрелка с помощью цилиндра и конуса.
- void LatticeGraphVisual3d::arc(double dlin/*,int i1,int j1,int k1*/) – во всех остальных случаях изображается изогнутая дуга. Поскольку иначе прямая дуга может пересекать вершины, не входящие в неё. Используются прямая линия, тор с плоскостями отсечения и конус.

Построенный граф вращается при помощи клавиатуры. При этом выполняются функции:

void CALLBACK Key_LEFT(void)

void CALLBACK Key_RIGHT(void)

void CALLBACK Key_UP(void)

void CALLBACK Key_DOWN(void)

Для вращения используются углы поворота alpha и beta. Уже разработана и функция, вращающая граф при помощи мышки:

void CALLBACK mouse(AUX_EVENTREC *event).

В дальнейшем планируется разработать возможность приближения графа и перемещение вдоль одной из осей координат.

Глава 4. Примеры

Пример 1.

Умножение матриц.

Пусть имеем матрицы A и B размера N*N. Перемножая их, получим матрицу C с элементами, вычисленными по формуле:

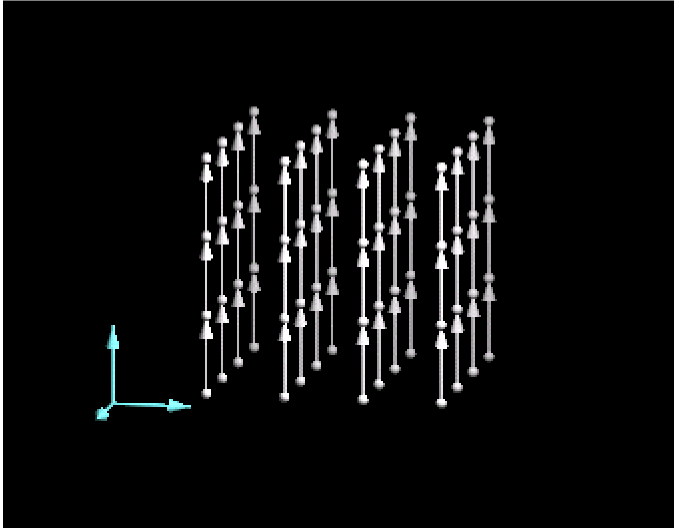
Do i = 1, N

Do j = 1, N

Do k = 1, N

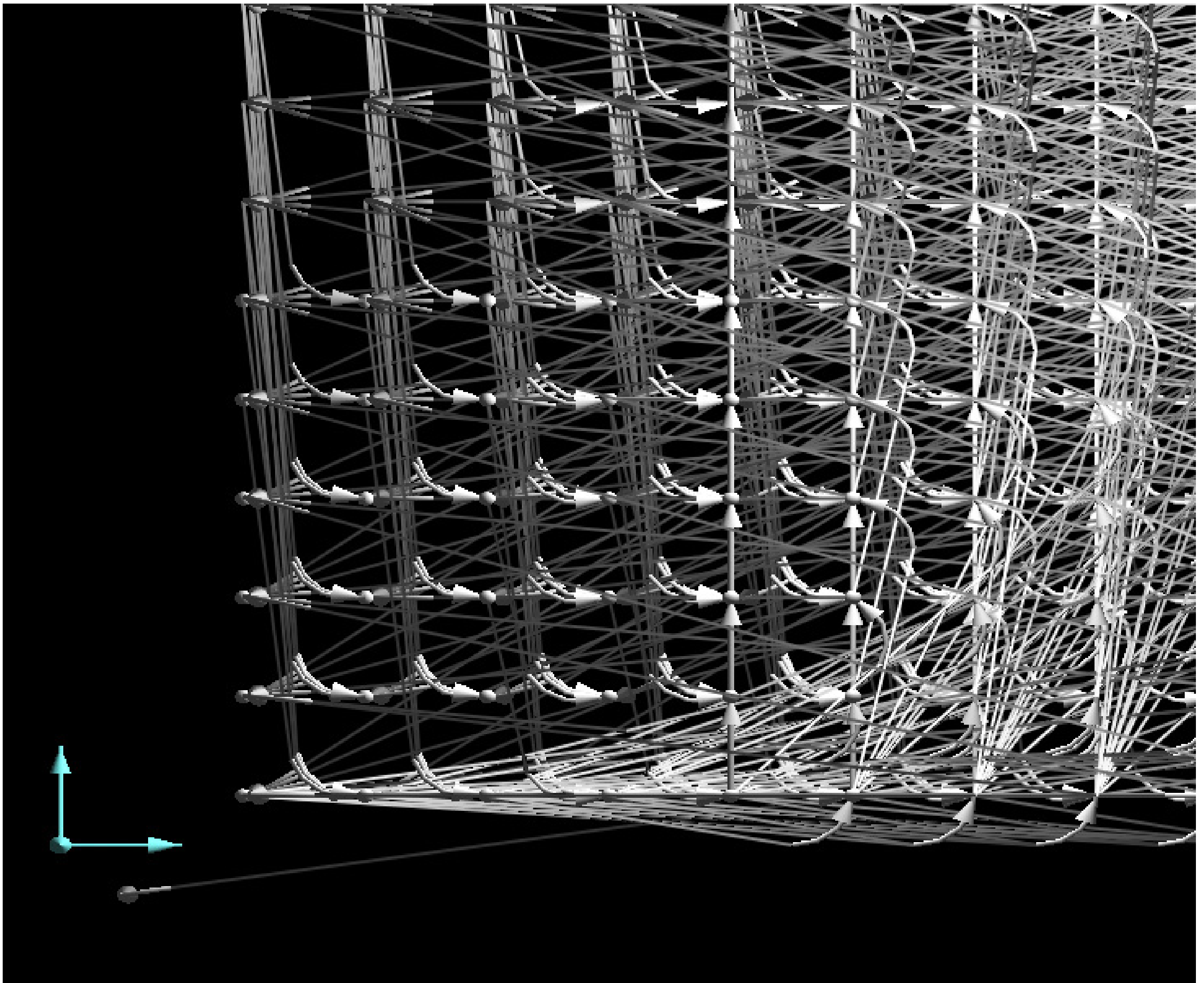
C (i , j) = C (i , j) + A (i , k) * B (k , j)

Рассмотрим случай, когда N = 4. Полученный граф имеет вид:



Пример 2.

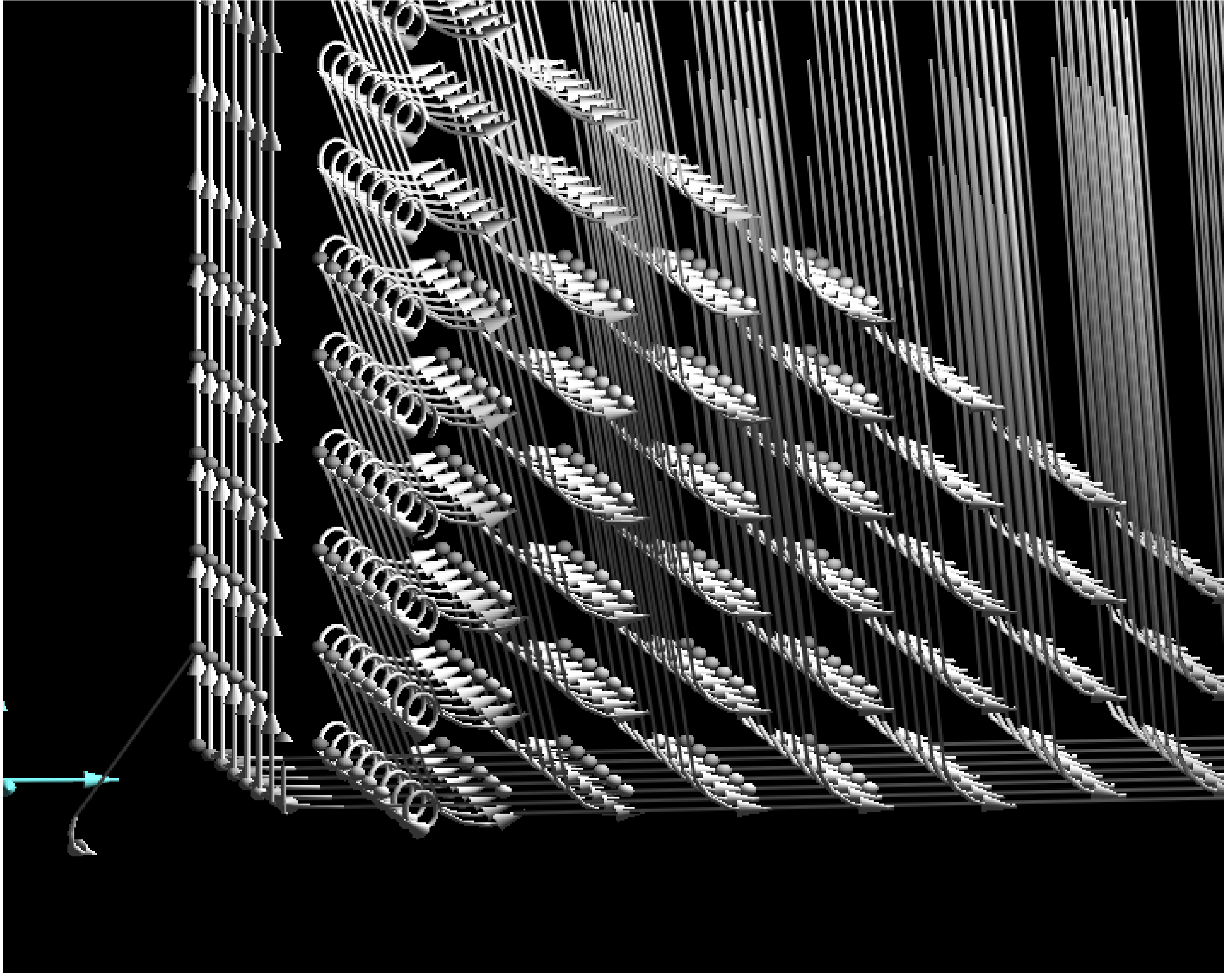
```
int u[21];
int main()
{
 int i,j,k;
 for(i=1;i<=10;i=i+1)
 {
  for (j=1;j<=10;j=j+1)
  {
   for (k=1;k<=10;k=k+1)
   {
     u[i+j+k]=u[21-i-j+k];
   }
  }
 }
}
```



Пример 3.

```
int b[50][50],a[50],x[50],k;
int main()
{
 int i,j,k;
 for(i=1;i<=10;i=i+1)
 {
  for(j=1;j<=10;j=j+1)
  {
   for(k=1;k<=10;k=k+1)
   {
     a[i+k]=b[i][j-1]+b[j][i];
     b[i][j]=a[i+j-2+k]+b[i][j];
```

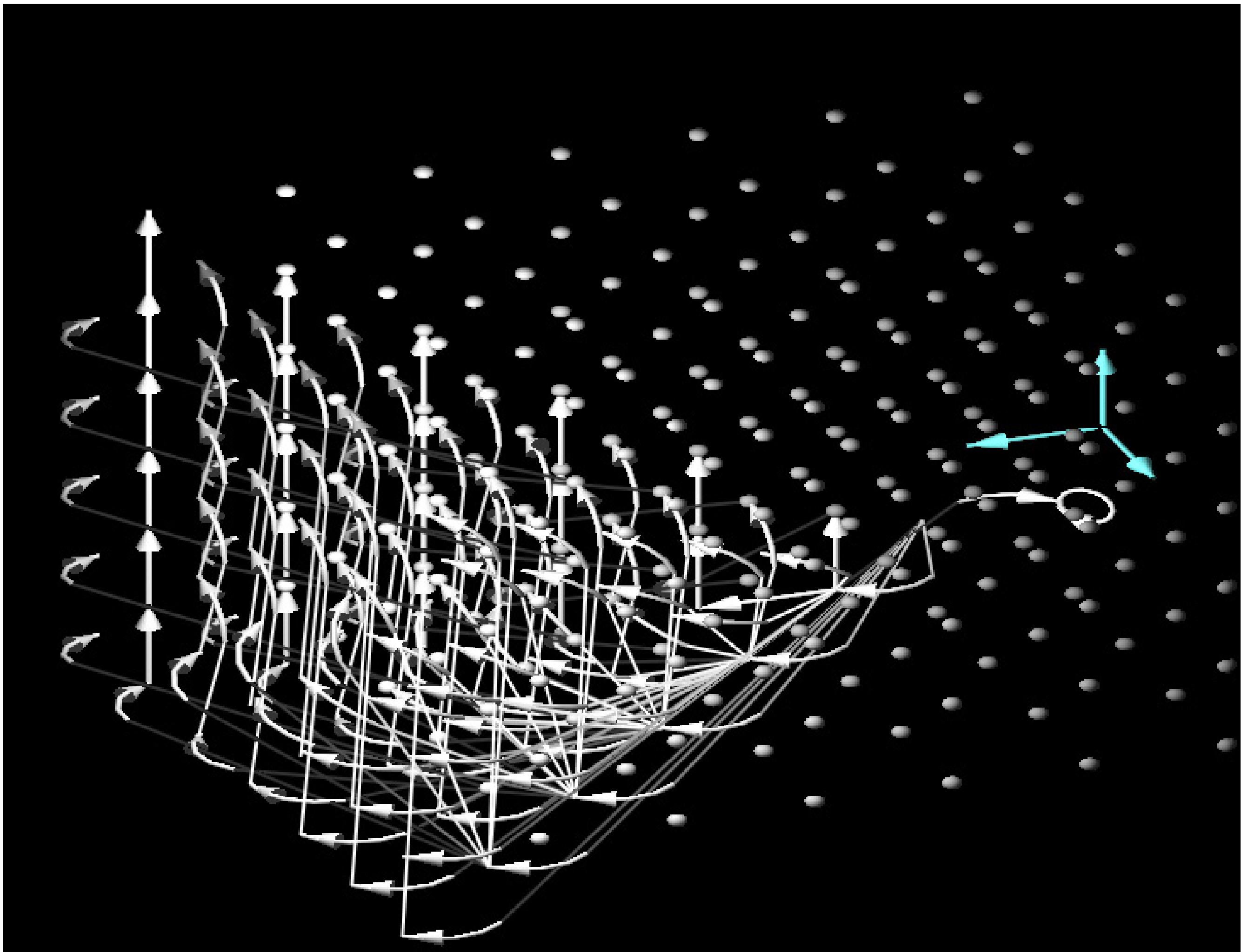
```
   }
  }
 }
}
```



Пример 4.

```
int a[10][10],x[10],b[10];
int main()
{
 int i,j,k;
 x[0]=b[0];
 for(i=1;i<=7;i=i+1)
 {
  x[i]=b[i];
```

```
  for ( j=0 ;j<=(i-1); j=j+1)
  {
   for ( k=0 ;k<=(i-1); k=k+1)
   {
    x[i+k]=x[i]-a[i][i-j]*x[i-j+k];
   }
  }
 }
}
```



Литература

1. Шульженко А.М. Исследование информационных зависимостей программ для анализа распараллеливающих преобразований. Диссертация на соискание ученой степени кандидата технических наук. РГУ, 2006.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Санкт-Петербург «БХВ-Петербург» 2002. – 608 с.
3. Воеводин В. В. Математические модели и методы в параллельных процессах. Москва: Наука. 1986. – 296 с.
4. Воеводин В. В. Информационная структура алгоритмов. М.: МГУ, 1997. – 139 с.
5. Воеводин В. В. Математические основы параллельных вычислений. М.: МГУ, 1991. – 345 с.
6. Feautrier P. Parametric integer programming// Operationnelle/Operations Research, 22(3):243--268, September 1988.
7. Feautrier P. Dataflow analysis of scalar and array references// International Journal of Parallel Programming, 20(1):23--52, February 1991.