МГУПИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ

Кафедра ‘Персональные компьютеры и сети’

В.М.Баканов

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

по выполнению лабораторных работ

Москва 2010

УДК 681.3

Рекомендовано к изданию в качестве учебно-методического пособия редакционно-издательским советом МГУПИ

Рецензент: профессор, к.т.н. Зеленко Г.В.

Баканов В.М.

Параллельные вычисления: учебно-методическое пособие по выполнению лабораторных работ. -М.: МГУПИ, 2010. – 53 с.

Рассматриваются вопросы выявле́ния скрытого параллелизма в алгоритмах путем я́вного (построение ярусно-параллельной формы графа алгоритма) и нея́вного (методика пото́ковых - DATA-FLOW - вычислений), разработки параллельных программ в MPI-парадигме программирования и количественного исследования величины ускорения вычислений при параллелизации от параметров многопроцессорной вычислительной системы и качества параллельных программ.

Пособие имеет практическую направленность и может быть использовано студентами для подготовки к выполнению лабораторных и практических работ, курсовых и дипломных проектов. Создаваемые сетевые приложения работоспособны в многопроцессорной среде архитектуры MPP (*Massively Parallel Processing*); в частности, на Linux-вычислительном кластере кафедры ИТ-4 МГУПИ. Перед проведением работ желательно ознакомиться с конспектом лекций по дисциплине ‘Параллельные вычисления’.

Подготовлено в соответствии Государственным стандартом для учебной дисциплины ‘Параллельные вычисления’.

Содержание

Стр.

Введение..... 4

1 *Лабораторная работа 1.* Представле́ние графа алгоритма в ярусно-параллельной форме ……... 6

2 *Лабораторная работа 2.* Решение задач на симуляторе пото́кового (DATA-FLOW) вычислителя…… 14

3 *Лабораторная работа 3.* Определение параметров коммуникационной сети вычислительного кластера……..... 22

4 *Лабораторная работа 4.* Простые MPI-программы (численное интегрирование)……. 28

5 *Лабораторная работа 5.* Исследование зависимости производительности многопроцессорной вычислительной системы от числа вычислительных узлов и размера обрабатываемых данных …… 40

Список использованной литературы..... 53

Введение

Программная пользовательская компонента технологий параллельных вычислений включают как выбор (а в некоторых случаях – самостоятельную разработку) алгоритма решения задачи, так и рациональную (с точки зрения архитектуры многопроцессорной вычислительной системы - МВС) его программную реализацию (выпуклый пример – классический алгоритм перемножения матриц может быть распараллелен десятком способов, на порядки разня́щихся по времени выполнения задачи с одинаковым размером исходным данных).

Важнейшим этапом здесь является выявление (обычно *скры́того*) параллели́зма в алгоритме (фактически выявление участков кода, независимых по данным – т.е. могущим *выполняться независимо*, а значит, и параллельно). Одним из методов выявления параллелизма является представле́ние алгоритма в т.н. *ярусно-параллельной форме* (ЯПФ); при этом на отдельном ярусе (слое) располагаются операторы, зависимые (по исходным данным для выполнения - операндам) только от результатов операций, находящихся уровнем выше. Операция представления алгоритма в ЯПФ может быть выполнена *программно* или *аппаратно* (с использованием вычислительной системы со специализированной архитектурой; http://burcom.ru).

MPI (*Message Passing Interface,* http://www.mpi-forum.org) является технологией создания параллельно выполняющихся программ, основанной на передаче сообщений между процессами (сами процессы могут выполняться как на одном, так и на различных вычислительных узлах). MPI-технология является типичным примером императивной (основанной на полном управлении программистом последовательностью вычислений, распределения данных и обменов информацией между процессами) технологии создания программ для параллельного выполнения [1,2]. Заметим, что сейчас и в дальнейшем всегда будем говорить ‘процесс’ (но не ‘процессор’), памятуя́, что современные многоядерные процессоры могут выполнять несколько независимых процессов одновременно.

Формально MPI-подход основан на включении в программные модули вызовов функций специальной библиотеки (заголо́вочные и библиотечные файлы для языков C/C++ и Fortran) и загрузчика параллельно исполняемого кода в вычислительные узлы (ВУ). Подобные библиотеки имеются практически для все платформ, поэтому написанные с использованием технологии MPI взаимодействия ветвей параллельного приложения программы независимы

от машинной архитектуры (могут исполняться на однопроцессорных и многопроцессорных с общей или разделяемой памятью ЭВМ), от расположения ветвей (выполнение на одном или разных процессорах) и от API (*Application Program Interface*) конкретной операционной системы (ОС).

Основными отличиями стандарта MPI-2 являются: динамическое порождение процессов, параллельный ввод/вывод, интерфейс для C++, расширенные коллективные операции, возможность одностороннего взаимодействия процессов (см. [3] и http://www.mpi-forum.org). В настоящее время MPI-2 нечасто; существующие реализации поддерживают версии MPI-1.

MPI является (довольно) низкоуровневым инструментом программиста; с помощью MPI созданы рассчитанные на численные методы специализированные библиотеки (например, включающая решение систем линейных уравнений, обращение матриц, ортогональные преобразования, поиск собственных значений и т.п. библиотеки ScaLAPACK, http://www.netlib.org/scalapack и AZTEC, http://www.cs.sandia.gov/CRF/aztec1.html для плотнозапо́лненных и разреженных матриц соответственно). MPI не обеспечивает механизмов задания начального размещения процессов по вычислительным узлам (процессорам); это должен явно задать программист или воспользоваться неким сторо́нним механизмом.

1 Лабораторная работа 1. **Представле́ние графа алгоритма в ярусно-параллельной форме**

1.1 Цель работы

Целью работы является получение навыков в представле́нии произвольных алгоритмов в *ярусно-параллельной форме* с целью выявления групп операторов, мо́гущих быть выполненными *независимо* (фактически *параллельно*).

Работа может быть рекомендована в качестве самостоятельной (домашней).

1.2 Теоретическая часть

Выявле́ние параллелизма в произвольном алгоритме – нетривиальная зада́ча. Дело в том, что параллелизм обычно *является скры́тым* (для нетрени́рованного ума). Однако существуют формальные процедуры, позволяющие выявить скрытый параллелизм в алгоритме; одна из них – представление алгоритма в ярусно-параллельной форме (ЯПФ).

Давно известен и применя́ем метод представления алгоритма в виде гра́фовой структуры. Граф G обычно обозначается $G=(V,E)$, где V – множество вершин (*vertex*), E – множество ребер (*edge*), ребро между вершинами i и j обозначается как $e(i,j)$. В общем случае вершины графа соответствуют некоторым *действиям* программы, а ребра – *отношениям* между этими действиями.

Простейший граф такого рода описывает *информационные зависимости алгоритма* (вершины графа соответствуют отдельным *операциям* алгоритма, наличие ребра между вершинами i,j говорит о необходимости при выполнении операции j наличия аргументов (*операндов*), выработанных операцией i; в случае независимости выполнения операций i и j дуга между вершинами отсутствует). Такой граф называют *графом алгоритма* (*вычислительной моделью 'операторы - операнды'*). Даже при отсутствии условных операторов (что маловероятно) число выполненных операций (а следовательно, и общее число вершин графа и, соответственно, число ребер) зависит от размеров входных данных, т.е. граф алгоритма (ГА) является *параметризованным* по размеру входных данных. Ацикли́чность ГА следует из невозможности определения любой величины в алгоритме через саму себя. ГА также является *ориентированным* (все ребра напра́влены). Различают *детерминированный* ГА (программа не содержит условных операторов) и *недетерминированный* ГА (в проти́вном случае). Для недерминированного ГА не существует взаимно-однозначного соответствия между операциями описывающей его программы и вершинами графа при всех наборах входных параметрах; поэтому чаще всего рассматриваются детерминированные алгоритмы. Не имеющие ни одного входящего или выходящего ребра вершины ГА называются *входными* или *выходными вершинами* соответственно. Построение ГА не является трудоемкой операцией (чего нельзя сказать о процедурах анализа графа) –

любой компилятор (интерпретатор) стр**о**ит (явно или неявно) его при анализе каждого *выражения* языка программирования высокого уровня

На рис.1.1 в общем виде представлен алгоритм вычислений по формуле r=a×b+a/c. Исходными данными для вычисления являются константы a,b,c, результат – r. В общем случае преобразование r←a,b,c требует 3 действий (*операторов*) – a×b, a/c и a×b+a/c. Исходными данными (*операндами*) для первого оператора являются a,b; для второго – a,c; для третьего – результаты вычислений a×b и a/c (см. осуществляющее преобразование r←a,b,c 'облако вычислений', показанное рис.1.1а).

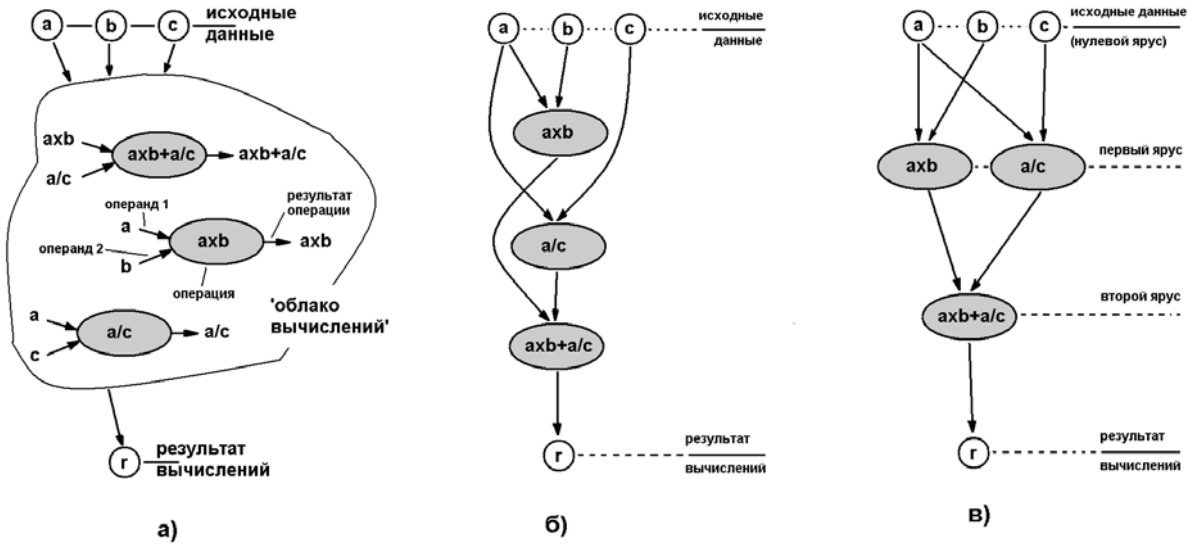

Рисунок 1.1 — Методы преобразова́ния r←a,b,c : а) - представле́ние алгоритма в виде 'о́блака вычислений' (порядок выполнения не определен), б) - последовательного выполнения, в) – я́русно-параллельная форма алгоритма

Для *последовательного вычислителя* порядок выполнения операторов лежит на поверхности – сначала вычисляются a×b и a/c (причем в любой последовательности), далее a×b+a/c (рис.1.1б). Т.к. выполнение a×b и a/c не зависит друг от друга (говорят, что они *ортогона́льны по опера́ндам*), легко получить ЯПФ для этого случая – рис.1.1в); часто из нумерации ярусов исключа́ют самый первый и последний (исходные данные и результаты соответственно, ибо на них не происходит собственно вычислений). В результате становится ясно, что данный алгоритм при параллельной обработке можно вычислить за 2 шага (a×b и a/c одновременно, затем a×b+a/c последовательно) вместо 3-х при последовательной (один за другим a×b, a/c и a×b+a/c); причем на первом ярусе необходима работа двух арифметических процессоров (действия 'умножение' и 'деление'), на втором – одного (действие – 'сложение'). При последовательной обработке общее время выполнения алгоритма будет равно сумме 3-х действий $t_{a\times b} + t_{a/c} + t_{a\times b+a/c}$, при параллельной – $\max(t_{a\times b}, t_{a/c}) + t_{a\times b+a/c}$ (в случае $t_{a\times b} = t_{a/c} = t_{a\times b+a/c}$ получаем выигрыш по скорости в полтора раза).

Фактически при преобразовании графа алгоритма в ЯПФ проводится *анализ внутренней структуры алгоритма* с целью поиска групп операторов,

мо́гущих быть выполненными параллельно. Заметим, что количество ярусов определяет *длину критического пути*.

Вообще говоря, ‘операторами’ можно считать *законченные действия любой сложности по преобразованию данных* – от *единичного выражения (строки́) или группы строк* в языке программирования высокого уровня до *единичной маши́нной (процессорной) инструкции.* Однако между этими указанными крайними случаями есть *существенная разница в числе операндов* (для оператора языка высокого уровня число операндов может достигать десятков/сотен, а для инструкции процессора обычно один/два). Конечно, осуществлять подобные вышеописанным преобразования с операциями, имеющими $1 \div 2$ операнда, много проще, чем при десятках операндов.

Рассмотрим чуть более сложный пример – разреше́ние корней x_1,x_2 полного квадратного уравнения $a \times x^2+b \times x+c=0$ посредством (предложенного индийским математиком Брахмагуптой в VII столетии н.э.) алгоритма в виде алгебраической формулы $x_{1,2}=(-b \pm sqr(b^2-4 \times a \times c))/(2 \times a)$, где a,b,c – постоянные, sqr - операция извлечения квадратного корня. Последовательность вычислений (один из вариантов) нахождения корней приведена́ в табл.1.1 и требует около десятка операций (сложение, вычитание, умножение, деление, изменение знака, вычисление квадратного корня).

Таблица 1.1. — Последовательность вычисления значений корней полного квадратного уравнения

№ оператора	*Действие*	*Примечание*
	Ввод a, b, c	Операции ввода не нумеруются
0	a2 ← 2 × a	a2 – рабочая переменная
1	a4 ← 4 × a	a4 – рабочая переменная
2	b_neg ← neg(b)	b_neg – рабочая переменная; neg – операция изменение знака (‘унарный минус’)
3	bb ← b × b	bb – рабочая переменная
4	ac4 ← a4 × c	ac4 – рабочая переменная
5	p_sqr ← bb-a4	p_sqr – рабочая переменная
6	sq ← sqrt(p_sqr)	sq – рабочая переменная, sqrt – операция вычисления квадратного корня
7	w1 ← b_neg+sq	w1 – рабочая переменная
8	w2 ← b_neg-sq	w2 – рабочая переменная
9	root_1 ← w1/a2	root_1 – первый корень уравнения
10	root_1 ← w2/a2	root_2 – второй корень уравнения

При последовательном вычислении граф алгоритма (рис.1.2) полностью копирует последовательность действий табл.1.1; имеет 3 входных вершины (соответствующие вводу коэффициентов a,b и c), две выходные вершины

(вычисленные корни x_1, x_2 исходного уравнения) и 11 вершин, соответствующих операторам алгоритма.

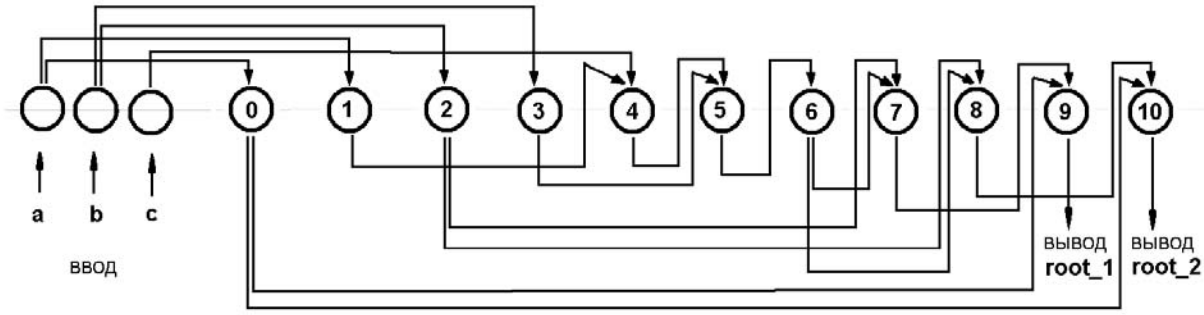

Рисунок 1.2 — Граф алгоритма (зависимость ‘операции – опера́нды’) нахождения корней полного квадратного уравнения для *последовательного выполнения*

Тот же граф предста́влен на рис.1.3 в ЯПФ – в каждом ярусе собраны операторы, требующие для своего выполнения значений (операндов), вычисляемых *только на предыдущих ярусах* (всего на рис.1.3 выделено 6 ярусов); т.о. параллельная обработка данного алгоритма требует последовательного 6-ра́зового выполнения блоков параллельных операций (в каждом из которых запускаются 4,1,1,1,2,2 независимых процесса соответственно, причем ярусы 2,3,4 вынужденно вырождаются в последовательное выполнение).

Анализ графа рис.1.3 позволяет уже сделать некоторые конкретные выводы об альтернативах распараллеливания. Заметим, что ярус 1 перегру́жен операциями (3 умножения и 1 изменение знака), часть из них (кроме операции 2) можно перенести на более нижние ярусы (варианты: операцию 4 на ярус 2, операцию 3 на ярусы 2,3 или 4, операцию 1 на ярусы 2,3,4 или 5); конкретный вариант должен выбираться исходя из дополнительных данных (например, время выполнения конкретных операций, число задействованных вычислительных модулей, минимизация времени обменов данными между мо-

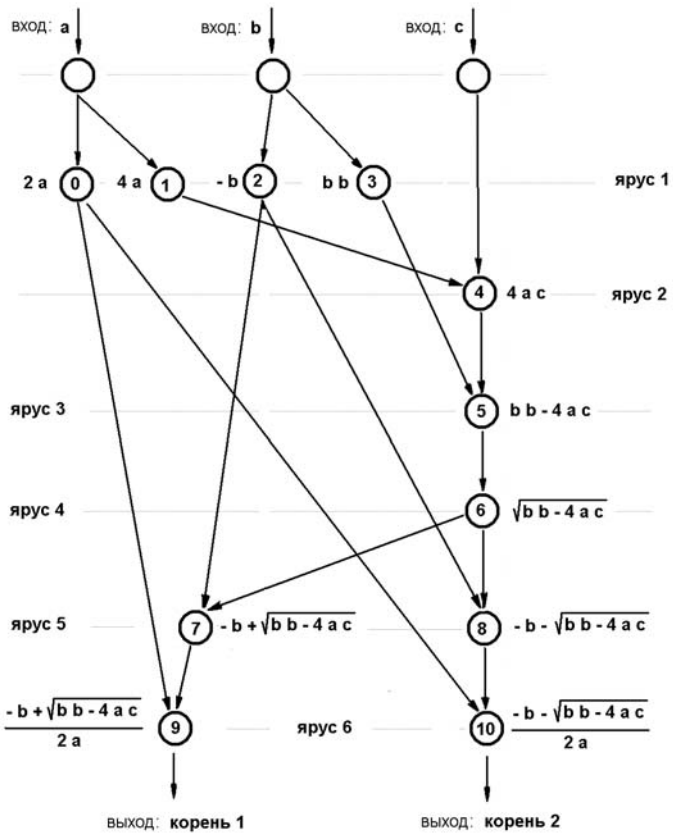

Рисунок 1.3 — Граф алгоритма (зави́симость ‘операции - операнды’) для нахождения корней полного квадратного уравнения с *группировкой операций по я́русам* (в ЯПФ)

дулями). Оптимизация графа вычислений - весьма трудоемкая (сравни́мая с трудоемкости самих вычислений по программе) операция; некоторые постановки проблемы приведены в работе [3].

Выявление этих ярусов представляет собой один из уровней *анализа внутренней структуры алгоритма*. При большом количестве операторов вышеуказанное преобразование ГА в ЯПФ провести непросто, для этого используют специальные программные средства.

Нижеприведена́ *упрощенная* последовательность действий по выявлению мо́гущих выполняться параллельно ярусов графа алгоритма (*вду́мчивый* читатель самостоятельно произведет необходимые уточнения и оптимизацию):

Пункт	*Действие*	*Примечание*
а)	Задать список вершин, зависящих *только от входных данных*; занести его в список *list_1*	Начальные данные для работы алгоритма задаются вручную
б)	Найти вершины, зави́симые по ребрам от вершин, входящих *только* в список *list_1* и предыдущие списки (если они есть); занести их в список *list_2*	Наиболее трудоемкий пункт, требующий просмотра *матрицы смежности* для каждого элемента списка
в)	Если список *list_2* не пуст, скопировать его в *list_1* и идти к пункту б); иначе закончить работу	Цикл по ярусам, пока **о**ные могут быть выявлены

Вариант (с целью достижения наибольшей прозрачности оптимизации не производилось) реализа́ции алгоритма на языке С приведен ниже (исходные данные - квадратная бу́лева матрица смежности MS[ ][ ] размерности N_MS, одномерные целочисленные массивы LIST_1[ ] и LIST_2[ ] длиной N_L1 и N_L2 соответственно):

```
001   do {                              // по ярусам сколько их будет выявлено
002    N_L2=0;
003    for (ii=0; ii<N_L1; ii++) {       // цикл по вершинам в списке LIST_1
004     i_ii=LIST_1[ii];                 // i_ii – номер вершины из списка LIST_1
005     for (j=0; j<N_MS; j++)           // цикл по столбцам MS (j – номер вершины, к которой направлено дуга)
006      if (MS[i_ii][j]) {              // нашли какую-то дугу i_ii→j
007       j1 = j;                        // запомнили вершину, к которой идет дуга от i_ii
008       for (i1=0; i1<N_MS; i1++)      // по строками MS = исходящим вершинам
009        if (MS[i1][j1]) {             // нашли какую-то дугу i1→j1
010          flag=false;
011          for (k=0; k<N_L1; k++)      // цикл по списку LIST_1
012           if (LIST_1[k] == i1)       // если вершина j1 входит в LIST_1…
013            flag=true;                // при flag=true вершина i1 входит в список LIST_1
014          } // конец блока  if (MS[i1][j1])
015        if (flag)                     // …если i1 входит в список LIST_1
016          LIST_2[N_L2++] = j1;        // добавить вершину j1 в список LIST_2
017      } // конец блока if (MS[i_ii][j])
```

```
018     }  // конец блока for (ii=0; ii<N_L1; ii++)
019     for(i=0; i<N_L2; i++)                  // копируем LIST_2 в LIST_1
020      LIST_1[i] = LIST_2[i];
021     N_L1 = N_L2;
022    } while (N_L2);                         // …пока список LIST_2 не пуст
```

Время t_j выполнения операций каждого яруса определяется временем исполнения наиболее длительной операции из расположенных на данном ярусе ($t_j = max(t_{ji})$, где j – номер яруса, i – номер оператора в данном ярусе). При планирования выполнения параллельной программы необходимо учитывать ограниченность по числу процессоров ($P \le P_{max}$), поэтому бывает рационально́ перенести часть (обычно быстро выполня́емых) операторов на нижележащие (менее запо́лненные) ярусы.

$$\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

Рисунок 1.4 — Матрица сме́жности для представле́ния графа алгоритма

При анализе алгоритма и выявлении ярусов параллелизма применяются удобные для ЭВМ методы представления графа в машинной памяти. Одним из (к сожалению, не самых экономных по использованию памяти) представлений графа $G=(V,E)$ является квадратная *ма́трица сме́жности* (нумерация строк и столбцов соответствует нумерации операторов, ‘1’ в (i,j)-той ячейке соответствует наличию ребра $e(i,j)$, ‘0’ – его отсутствию), см. рис.1.4. Классически матрица смежности суть бу́лева, однако при необходимости более тонкого моделирования ее элементы могут быть целыми или даже вещественными; например, с целью учета значения ме́трики (в качестве задержки времени доставки информации, например) канала передачи данных (если отдельные операторы выполняются на различных узлах МВС).

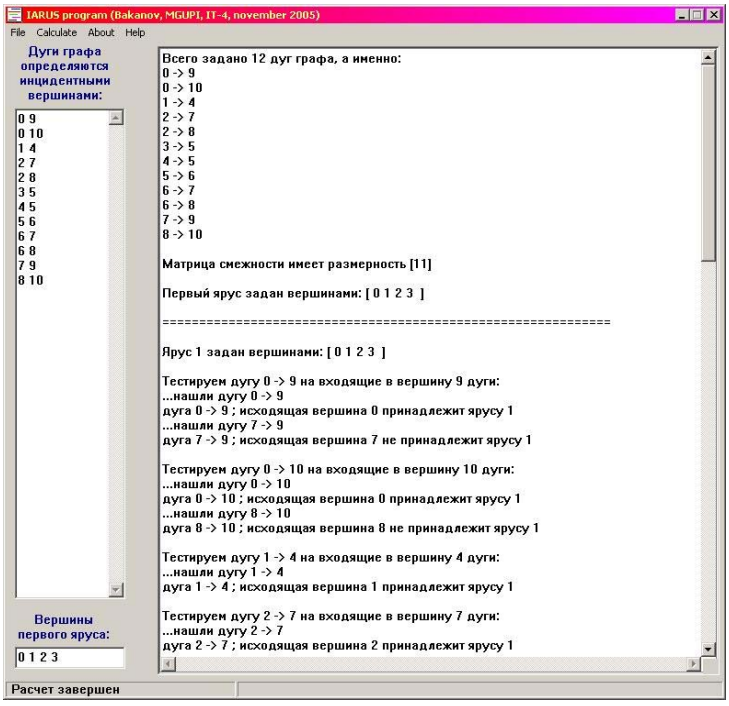

Рисунок 1.5 — Пользовательский интерфейс программы выявле́ния ярусов параллелизма в заданном графе алгоритма

Копия главного окна программы IARUS, служащей для выявления ярусов параллелизма в графе алгоритма, приведена на рис.1.5 (выполнено преобразование в ЯПФ выше-

рассмотренного алгоритма нахождения корней квадратного уравнения). Программа в основном следует вышеприведенным алгоритмам.

Отметим, что в программе IARUS при вводе данных (левый верхний фрейм главного окна на рис.1.5) использован иной способ описания графа алгоритма – метод *инцидéнтных* (т.е. принадлежащих заданному ребру) *вершúн* (каждое ребро описывается двумя параметрами, соответствующими идентификаторам обоих вершин, образующих это ребро). Для начала счета следует ввести (в левом нижнем фрейме) номера вершин, представляющими исходные данные (идентификаторы вершин в виде целых чисел, разделенные одним или несколькими пробелами), протокол расчета выводится в правый фрейм главного окна; файл проекта и протокол расчета могут быть сохранены для модификации и/или анализа.

1.3 Необходимое оборудование

Для проведения работы необходима персональная IBM-совместúмая ЭВМ под управлением Windows, комплект файлов программы IARUS.

1.4 Порядок проведения работы

Студент получает индивидуальное задание (из любой области знаний, сложность характеризуется числом арифметических действий - не более $20 \div 30$), составляет алгоритм, записывает его в последовательном виде в таблицу аналогичную табл.1.1, строит граф алгоритма; далее вручную или с помощью программы IARUS преобразует его в ЯПФ.

1.5 Представление результатов работы

Преподавателю представляется ЯПФ заданного алгоритма в графической форме, оценивается *коэффициент ускорения вычислений* (отношение общего числа операций к числу ярусов) вследствие параллелизáции, максимальное число операций на ярусах и возможность его снижения путем переноса части операций на нижележащие ярусы.

1.6 Вопросы для самопроверки

1. Какие виды графов используются при представлении алгоритмов?
2. Какие зависимости в алгоритме представляет *граф алгоритма*? Каким объектам в алгоритме соответствуют его вершины и дуги?
3. Почему матрица смежности является квадратной? Предложите иной (более экономичный с точки зрения использования памяти ЭВМ) метод сохранения графа алгоритма?
4. Каков размер (в килобайтах) бу́левой матрицы смежности при числе операторов 1’000?
5. Предложите алгоритм нахождения все дуг, принадлежащих заданной вершине, при задании графа методом инцидентных вершин.
6. Дать определение ярусно-параллельной форме алгоритма? Какими свойствами обладают операторы, расположенные на одном ярусе?
7. Каким образом можно ‘разгрузи́ть’ определенный ярус от излишнего количества операторов на нем?

2 Лабораторная работа 2. **Решение задач на симуля́торе пото́кового (DATA-FLOW) вычислителя**

2.1 Цель работы

Целью работы является ознакомле́ние с архитектурой и принципами действия пото́кового (DATA-FLOW) вычислителя, программирование и решения задач на нем.

Работа может быть рекомендована в качестве самостоятельной (домашней) и служить обеспечением научно-исследовательской работы студентов.

2.2 Теоретическая часть

Архитектура большинства современных процессоров базируется на идеях фон Неймана, одной из которых является последовательная вы́борка и выполнение операций (*маши́нных инструкций*). Подобная паради́гма препятствует *глубокому распараллеливанию вычислений*, без чего немыслимо создание суперпроизводительных вычислителей, необходимых в науке и технике [2].

В фон-Неймановских машинах процессом вычислений управляет регистр – счетчик команд (СК), определяющий номер выбираемой из памяти и выполняемой машинной инструкции; путем модификации содержимого СК реализуются условные и безусловные переходы, циклы. Однако именно СК является ’слабым звеном’ классической архитектуры, препятствующим глубокому распараллеливанию программы на уровне машинных инструкций (вследствие явного указания СК в данный момент на одну-единственную инструкцию). В современных процессорах (*Merced, Itanium, Эльбрус*) применяют различные ухищре́ния (EPIC и WLIV-подходы, спекулятивные вычисления), позволяющие выполнять *более одной* (обычно до $5 \div 6$) *инструкции одновременно*.

В начале 1970-х г.г. Джек Деннис (*Jack Dennis*) изложил фундаментальные принципы *управления процессом вычислений пото́ком данных* (DATA-FLOW) в противове́с *программному управлению* (CONTROL-FLOW). Основополагающим понятием при этом становится принцип *готовности к выполнению операций* по условию *гото́вности всех необходимых для выполнения этой операции операндов*. ‘Операнд гото́в’, если соответствующей ячейке памяти неким корректным образом присвоено значение (вычислено или за́слано константой).

В качестве устройства управления вычислениями выступает (вместо СК) коммутатор, получающий информацию об изменении значений операндов на предмет выявле́ния ‘готовых к выполнению’ операций и передающий их имеющимся в системе процессорам (*процессорному полю, о́блаку вычисли́телей*), вновь вычисленные значения определяют готовность следующих операций и т.д. Если в системе имеется достаточное количество процессоров, все готовые в данный момент к выполнению команды будут переданы на исполнение, что и позволяет говорить об *истинно ма́ссовом* (причем реализуе-

мым не программным, а аппаратным – посредством функционирования архитектуры) *параллелизме на уровне машинных инструкций* в DATA-FLOW системах. Между памятью и исполнительными устройствами (коммутаторами, процессорами) циркулируют *токены* (комбинации данных с дополнительными флагами, указывающими на *признаки* данных - готовность к выполнению и др.). Интересно, что в идеале в подобной системе *порядок следования операций несущественен*. Составляющие процессорное поле процессоры не обязательно должны быть универсальными (т.к. значительную долю вычислений составляют арифметические, большее число процессоров может быть рассчитано только на эти действия). DATA-FLOW распараллеливание характеризуется наиболее мелкогранулярным параллелизмом (*fine-grained parallelism*).

Полномасштабная реализация DATA-FLOW вычислителей наталкивается на трудности технологического характера – вследствие работы множества процессоров на единую оперативную память (фактически это аналог архитектуры *компьютеров с общей памятью*, т.е. SMP - *Symmetric Multi Processors*) требуется как огромная производительность шины и самой памяти, так и сложная система диспетчеризации обменов данными. Современные технологии создания процессоров с числом ядер $48 \div 100$ (фирмы *Intel, Tilera, nVidia*) могут рассматриваться мостом к переходу на DATA-FLOW архитектуру.

Несмотря на то, что исторически магистральным направлением архитектуры вычислителей (процессоров) стала классическая фон-Неймановская, работа над не-фон-Неймановскими машинами никогда не угасала. Из разработок DATA-FLOW вычислителей известны *JUMBO* (компьютер с управлением потоком данных университета Ньюкасла на Тайне, Англия), *Манчестерский компьютер потока данных* (Manchester Dataflow Computer), проекты *Monsoon* и *Epsilon* (США), *CSRO* (Австралия) etc. В России до своей кончины в 2005 г. реализацией DATA-FLOW вычислителей занимался В.С.Бурцев.

Максимально упрощенно схема потокового вычислителя приведена на рис.1. Вычислительное устройство представляет собой кольцевую структуру, токены (данные с признаками) хранятся в *ассоциативной памяти* (АП). ‘Готовые к выполнению’ машинные инструкции (с соответствующими признаками) с помощью *входного коммутатора* K_1 передаются *исполняющим устройствам* (процессорам, устройствам ввода/вывода и т.п.). Если все исполняющие устройства в данный момент заняты, готовые инструкции накапливаются в *буфере*

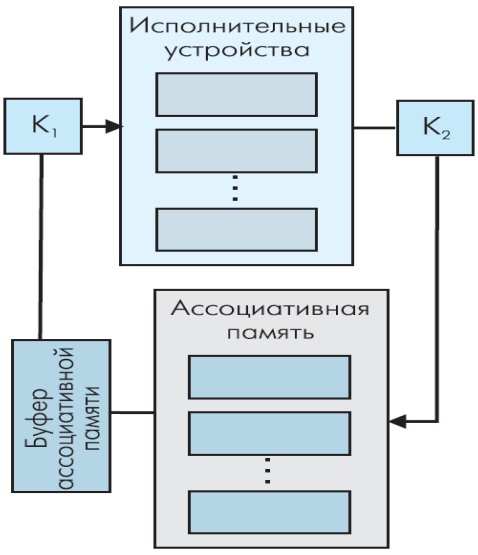

Рисунок 2.1. Упрощенная структурная схема потового вычислителя (журнал ‘ЭЛЕКТРОНИКА: Наука, Технология, Бизнес’, 2002, № 2)

ассоциативной памяти – (БАП); результаты выполнения инструкций через выходной коммутатор K_2 загружа́ются опять же в АП. При угрозе переполнения БАП и АП предусмотрен механизм первоочередного выполнения команд, ведущих к сокращению числа токенов. Правила обработки токенов позволяют реализовать многократное выполнение последовательности инструкций (т.е. организовывать циклы) и выполнять безусловные и условные переходы [2].

В результате пото́ковый вычислитель на *аппаратном уровне* выделяет яруса́ параллелиза́ции (фактически строит ЯПФ, см. работу 1 данного пособия) и т.о. *аппаратно распараллеливает алгоритм*.

На рис.2.2 приведена копия главного окна компьютерной программы-симуля́тора пото́кового вычислителя. Левый нижний фрейм главного окна программы служит для отображения содержимого *памяти инструкций* (ПИ), правый нижний – *памяти данных* (ПД); подобное разделение общей оперативной памяти сделано с целью повышения наглядности, верхний фрейм служит для вывода протокола решения задачи.

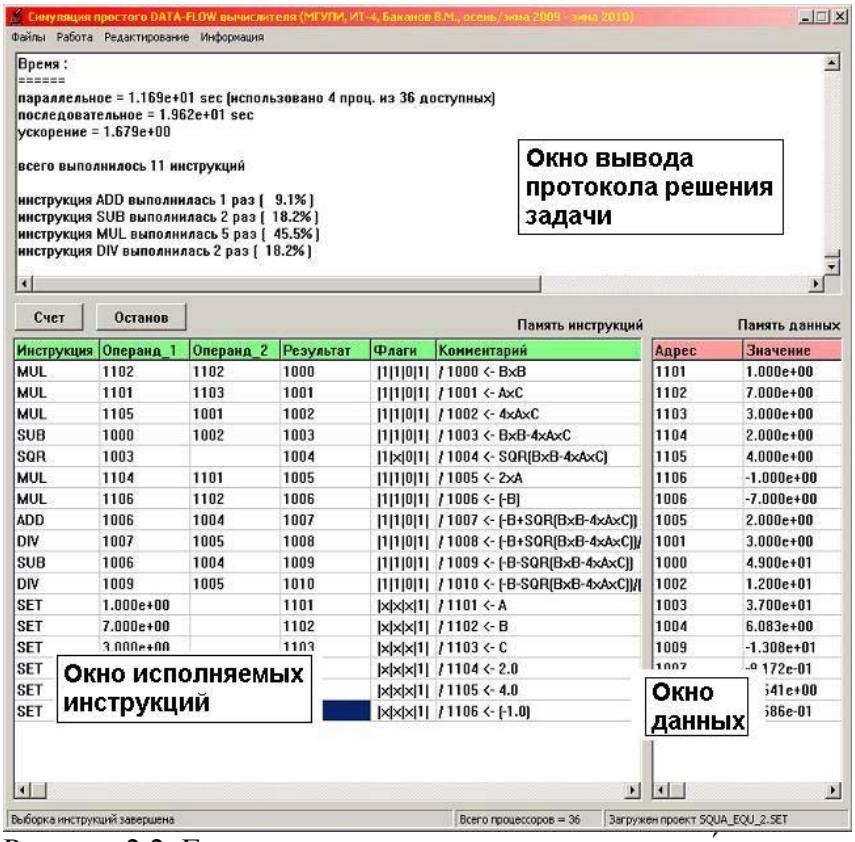

Рисунок 2.2. Главное окно программы – симуля́тора пото́кового (DATA-FLOW) вычислителя

Упрощенный (для учебного процесса) вариант симулятора выполняет 18 инструкций (все над вещественными числами двойной точности), см. табл.2.1. Все инструкции (кроме SET, выполняющейся внепроце́ссорно и служащей для занесе́ния в ПД исходных данных для расчета) используют трехсимвольную мнемо́нику и работают с операндами, заданными адреса́ми-идентификаторами в ПД (идентификатором может служить последовательность любых символов до 15 штук, в строке команды после символа ‘/’ располагается комментарий).

Таблица 2.1. — Список машинных инструкций (операторов) симуля́тора пото́кового (DATA-FLOW) вычислителя

№№	*Инструкция*	*Выполняемое действие*
1	SET вещ._число,	Вещественное число помещается в ПД по адресу

	a_результат	a_результат
2	ADD a_операнд_1, a_операнд_2, a_результат	Значения вещественных чисел по адресам операндов a_операнд_1 и a_операнд_2 складываются и результат помещается по адресу a_результат
3	SUB a_операнд_1, a_операнд_2, a_результат	Из вещественного числа по адресу a_операнд_1 вычитается число по адресу a_операнд_2 результат помещается по адресу a_результат
4	MUL a_операнд_1, a_операнд_2, a_результат	Значения вещественных чисел по адресам операндов a_операнд_1 и a_операнд_2 перемножаются и результат помещается по адресу a_результат
5	DIV a_операнд_1, a_операнд_2, a_результат	Вещественное число по адресу a_операнд_1 делится на число по адресу a_операнд_2 результат помещается по адресу a_результат
6	SQR a_операнд, a_результат	Вычисляется квадратный корень из вещественного числа по адресу a_операнд, результат помещается по адресу a_результат
7	POW a_операнд_1, a_операнд_2, a_результат	Вещественное число по адресу a_операнд_1 возводится в вещественное по адресу a_операнд_2 результат помещается по адресу a_результат
8	SIN a_операнд, a_результат	Вычисляется синус из вещественного числа по адресу a_операнд (в радианах), результат помещается по адресу a_результат
9	COS a_операнд, a_результат	Вычисляется косинус из вещественного числа по адресу a_операнд (в радианах), результат помещается по адресу a_результат
10	TAN a_операнд, a_результат	Вычисляется тангенс из вещественного числа по адресу a_операнд (в радианах), результат помещается по адресу a_результат
11	ASN a_операнд, a_результат	Вычисляется арксинус из вещественного числа по адресу a_операнд, результат помещается по адресу a_результат (в радианах)
12	ACN a_операнд, a_результат	Вычисляется арккосинус из вещественного числа по адресу a_операнд, результат помещается по адресу a_результат (в радианах)
13	ATN a_операнд, a_результат	Вычисляется арктангенс из вещественного числа по адресу a_операнд, результат помещается по адресу a_результат (в радианах)
14	LOG a_операнд, a_результат	Вычисляется натуральный логарифм из вещественного числа по адресу a_операнд, результат помещается по адресу a_результат
15	EXP a_операнд, a_результат	Вычисляется экспонента от вещественного числа по адресу a_операнд, результат помещается по адресу a_результат
16	SNH a_операнд, a_результат	Вычисляется гиперболический синус $((e^{x}-e^{-x})/2)$ от вещественного числа по адресу a_операнд, результат помещается по адресу a_результат
17	CNH a_операнд, a_результат	Вычисляется гиперболический косинус $((e^{x}+e^{-x})/2)$ от вещественного числа по адресу a_операнд, результат помещается по адресу a_результат
18	TNH a_операнд, a_результат	Вычисляется гиперболический тангенс $((e^{x}-e^{-x})/(e^{x}+e^{-x}))$ от вещественного числа по адресу a_операнд, результат

		от вещественного числа по адресу a_операнд, результат помещается по адресу a_результат

Данный симулятор обладает ограничениями:

- Поддерживаются только *линейные алгоритмы* (условные и безусловные переходы не определены).
- Требуется следование концепции '*однократного присваивания*' (любое вновь вычисляемое значение необходимо поместить в *новую ячейку памяти*).

Процесс счета начинается с занесения в ПД данных (с помощью инструкции SET). При окончании выполнения каждой инструкции результат ее выполнения заносится в ПД и в ПИ ищутся операнды, соответствующие этому результату; если все операнды какой-либо инструкции 'готовы' – она передается на выполнение первому свободному процессору. Процесс повторяется до выполнения всех инструкций.

Файл программы имеет фиксированное расширение .SET и загружается в симулятор стандартным способом; ниже приведен пример программы (решение полного квадратного уравнения).

```
MUL   1102,   1102,   1000   / 1000←b×b
MUL   1101,   1103,   1001   / 1001←a×c
MUL   1105,   1001,   1002   / 1002←4×a×c
SUB   1000,   1002,   1003   / 1003←b×b-4×a×c
SQR   1003,           1004   / 1004←SQR(b×b-4×a×c)
MUL   1104,   1101,   1005   / 1005←2×a
MUL   1106,   1102,   1006   / 1006←(-b)
ADD   1006,   1004,   1007   / 1007←(-b+SQR(b×b-4×a×c))
DIV   1007,   1005,   1008   / 1008←(-b+SQR(b×b-4×a×c)) / (2×a) = x1
SUB   1006,   1004,   1009   / 1009←(-b-SQR(b×b-4×a×c))
DIV   1009,   1005,   1010   / 1010←(-b-SQR(b×b-4×a×c)) / (2×a) = x2
SET   1.0,            1101   / 1101←a
SET   7.0,            1102   / 1102←b
SET   3.0,            1103   / 1103←c
SET   2.0,            1104   / 1104←2
SET   4.0,            1105   / 1105←4
SET   -1.0,           1106   / 1106←(-1)
```

При a=1.0, b=7.0, c=3.0 решением является x1=-0.4586, x2=-6.541. Эта программа находится в файле SQUA_EQU_2.SET.

При старте симулятора из файла DATA_FLOW.EXE в него автоматически загружается проект DATA_FLOW.SET и файл настроек DATA_FLOW.INI (в нем, в частности, определяются времена выполнения каждой инструкции в тысячных долях секунды). Управление осуществляется двумя кнопками и посредством выпадающего главного меню. Дополнительная информация может быть получена посредством нажатия F1 и Alt+F1 в среде симулятора.

Нажатием Ctrl+F4 симулятора вызывается текстовый редактор с именем, занесенным в строку File настроечного файла DATA_FLOW.INI ; Shift+F4 позволяет таким же образом редактировать собственно файл DATA_FLOW.INI. Для обновления ПК и файла настроек служит клавиша F5 (это при MODE=0 в DATA_FLOW.INI, при MODE=1 редактор вызывается мода́льно и обновление происходит автоматически по закрытию окна редактора).

При выявле́нии проблем с вычислениями полезен анализ флагов, управляющих вычислениями (5-я слева колонка окна '*Память инструкций*'). В этой колонке индицируется состояние 4-х булевых флагов в форме |a1|a2|b|c| (в случае неиспользования флага в конкретной ситуации выдается 'x').

Установка флагов 'a1' и 'a2' индицирует *признак готовности операндов* (значение по соответствующему адресу в '*Памяти данных*' присво́ено инструкцией SET или вычислено предыдущими инструкциями). Флаг 'b' устанавливается на время исполнения инструкции (и очищается при окончании), флаг 'c' устанавливается после выполнения инструкции.

При 'зависа́нии' процесса вычислений (прекращена́ выборка операций) несложно по состоянию этих флагов определить причину - обычно это инструкция, один (или оба) флага 'a1' и 'a2 готовности операндов которой сбро́шены; в этом случае рациональным будет анализ флага 'c' у инструкции, адрес результата которой равен адресу (адреса́м) соответствующих операндов проблемной инструкции (и, конечно, нали́чие соответствующей строки в '*Памяти данных*').

В базовом варианте симулятора число инструкций в ПИ и пар 'адрес/значение' в ПД равно 1'000 штук каждой, число процессоров – до 100.

Протоколы расчета сохраняются при нажатии пользователем клавиши F2 в файлах с именем загру́женного файла проекта и различными расширениями (форматы данных приводятся для возможного их импорта в программы-анализаторы сторо́нней разработки):

1. *Расшире́ние PRO* - главный файл протокола (представляет собой копию строк, выводимых при решении задачи в *верхний фрейм* главного окна программы-симулятора).
2. *Расшире́ние TPR* - файл анализа времени загрузки процессоров инструкциями программы. Формат каждой из основной части строк этого файла (числовые данные сформатированы по *правой границе поля*):
   - *10 символов:* номер (начиная с 0) процессора, выполняющего данную инструкцию (целое).
   - *10 символов:* номер (начиная с 0) инструкции, выполняемой данным процессором (целое).
   - *10 символов:* время (начиная с момента начала выборки инструкций) *начала выполнения* данной инструкции данным процессором), сек (вещественное).

- *10 символов:* время (начиная с момента начала выборки инструкций) *конца выполнения* данной инструкции данным процессором), сек (вещественное).
- *10 символов:* время выполнения данной инструкции данным процессором, сек (вещественное).
- *До конца строки:* текст инструкции (квадратных скобках); после адреса/идентификатора в круглых скобках выдается значение по данному адресу в ПД (*операнд* или *результат*) + (неизменный) комментарий.

3. *Расширение* TST – файл количества одновременно исполняемых инструкций [(*)] в функции времени; эти данные удобно использовать путем импорта их в MS Excel для построении зависимости числа исполняемых инструкций от времени (числовые данные сформатированы по *правой границе поля*):
   - *10 символов:* время с начала выполнения программы, сек (вещественное).
   - *10 символов:* число процессоров, занятых выполнением инструкций (целое).
   - Начиная с 26-й позиции перечисляются (через 2 пробела) все выполняемые в данные момент инструкции в формате '*Номер_процессора/Номер_инструкции_в_ПИ/Мнемоника инструкции*'.
4. *Расширение* DAT - файл вычисленных программой данных (копия строк *правого фрейма* главного окна). Формат каждой строки этого файла (данные сформатированы *по правой границе поля*):
   - *20 символов:* адрес/идентификатор переменной.
   - *20 символов:* значение по этому адресу (вещественное).

Внимание! При каждом сохранении файлов протокола предыдущие версии файлов *перезаписываются без сохранения*. Пользователь должен позаботиться о сохранении нужной ему информации.

2.3 Необходимое оборудование

Для проведения работы необходима персональная IBM-совместимая ЭВМ под управлением Windows, комплект файлов программы DATA_FLOW.

2.4 Порядок проведения работы

Студент получает индивидуальное задание (из любой области знаний, сложность характеризуется числом арифметических действий – обычно в пределах $20 \div 30$, причем исходные данные задаются конкретными числовыми значениями), составляет алгоритм, записывает его в файл; далее загружа-

[*] При задании количества процессоров (величина Max_Procs в файле DATA_FLOW.INI), заведомо бо́льшим (или равным) максимальному числу выполняемых одновременно инструкций, может быть определен (априори неизвестный) *потенциал распараллеливания* данной программы.

ет в программу DATA_FLOW, отлаживает (исправляет ошибки) и совершает просчет. Преподаватель может ограничить число процессоров в системе определенным числом или же позволить считать без ограничений (при этом может быть определен *потенциал распараллеливания* данного алгоритма).

1.5 Представление результатов работы

Преподавателю представляется отчет, который должен включать:

- работоспособную программу (содержимое файла с расширением SET);
- содержимое памяти данных (содержимое DAT-файла);
- график интенсивности вычислений *I(t)* - зависимость числа одновременно выполняемых инструкций от времени выполнения программы; получается путем импорта данных первых двух столбцов из TST-файла в MS Excel и соответствующей обработки;
- статистические данные выполнения программы (общее число инструкций, процент каждой, время выполнения последовательной и параллельной версий программы и коэффициент ускорения вычислений при параллельном исполнении).

2.6 Вопросы для самопроверки

1. Каковы основные принципы фон-Неймановской архитектуры вычислителей? Какие из них отрицаются пото́ковыми (DATA-FLOW) вычислителями?
2. Что такое счетчик команд? Как с его помощью реализуются условные и безусловные переходы? По какой причине счетчик команд является препятствием к распараллеливанию программы на уровне машинных инструкций?
3. Как понимать ‘гото́вность операции’? Какие условия должны быть соблюдены, чтобы операции считалась ‘готовой’ к выполнению?
4. На основании каких соображений считается, что последовательность команд в DATA-FLOW машине несущественна? Будет ли зависеть последовательность выборки команд на исполнение от последовательности команд в ПИ?
5. По какой причине практически все современные процессоры основаны на классической (фон-Неймановской) архитектуре? Чем обусловлены проблемы технической реализации пото́ковых (DATA-FLOW) вычислителей? Какие события в мире информационных технологий способствуют развитию неклассических (в частности, пото́ковых) архитектур?

3 Лабораторная работа 3. **Определение параметров коммуникационной сети вычислительного кластера**

3.1 Цель работы

Целью работы является замер реальной (с учетом *латéнтности*) производительности коммуникационной сети вычислительного кластера на блокирующих операциях ‘точка-точка’.

3.2 Теоретическая часть

В вычислительных кла́стерах наиболее ‘узким местом’ обычно является сеть, осуществляющая связь между вычислительными узлами (фактически эта сеть является ана́логом общей шины, связывающей различные узлы любой ЭВМ); эта сеть называется *коммуникационной* (в противовес *управляющей*, по которой осуществляется управление вычислительными узлами). Для реализации особо мощных МВС применяют третью сеть для поддержки NFS (*Network File System)*; при двух сетях NFS обычно разделяет *сеть управления*.

Даже весьма быстродействующая сеть обладает свойством *латéнтности* (инерционности, временем ‘разго́на’ до номинальной производительности); латентность особо сильно снижает реальную производительность сети при *частом обмене небольшими по объему* сообщениями (при задержке поступления очередного порции информации вычислительному узлу он вынужденно простаивает, рис.3.1).

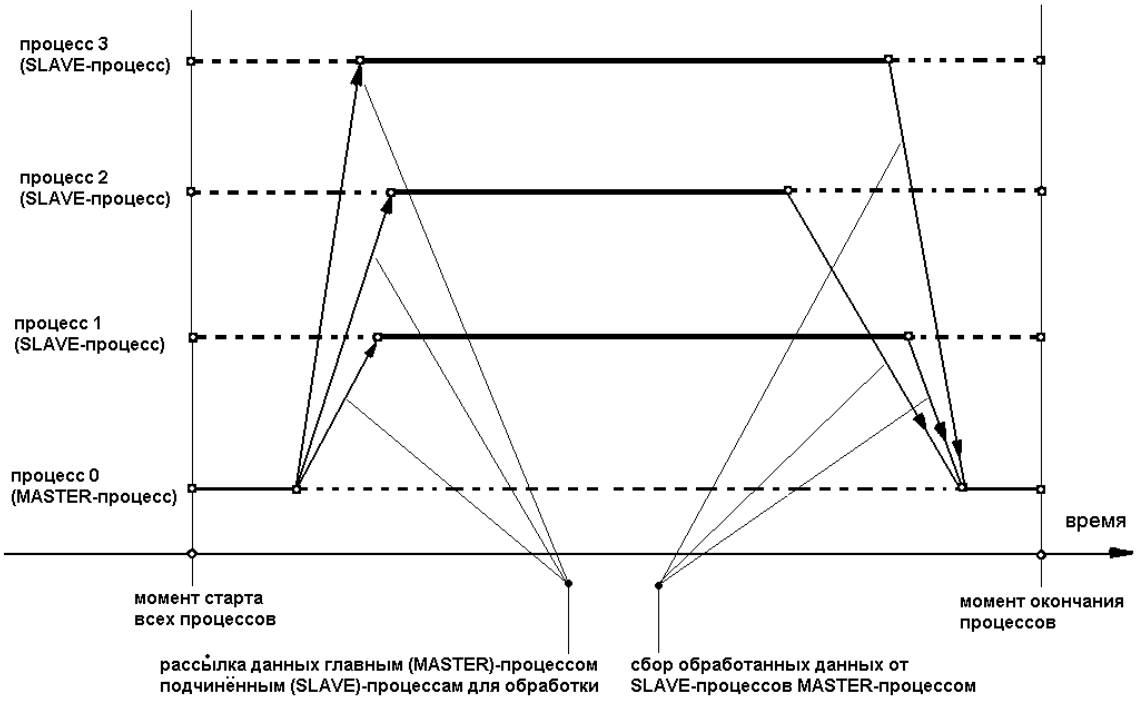

Рисунок 3.1 — Простейший случай жизненного цикла процессов и обмена данными между ними: пунктир - процесс исполняется, но ожидает обмена данными (обработка данных не происходит), сплошные линии - процесс исполняется и обрабатывает данные.

Для 100/1’000 Мбод Ethernet-сетей величина латентности обычно равна нескольким десяткам мксек, для виртуального (из Linux-машин, работающих под VMware в среде Windows) кластера латентность может достигать $300 \div 400$ мксек. Наилучшими значениями латентности (порядка 1 мксек) при огромной пропускной способности обладают (дорогостоящие) сети Infini-Band.

В случае одностороннего обмена сообщениями между двумя узлами (обмен типа ‘точка-точка’) затра́чиваемое на передачу время T (сек) оценивается как:

$$T=X/S+L,$$

где X – длина сообщения (Мбайт),

S – пропускная способность сетевого канала ‘точка-точка’ (мгновенная скорость передачи данных), Мбайт/сек,

L – время разгона операции обмена (не зависит от длины сообщения), сек.

Иногда бывает удобно оперировать латентностью, приведе́нной к скорости (*цена обмена* P, Мбайт):

$$P=L \times S,$$

здесь цена обмена – размер блока данных, которые канал ‘точка-точка’ мог бы передать при нулевой латентности.

При определении реальной (с учетом латентности) пропускно́й способности сети на операциях обмена типа ‘точка-точка’ используют пару простейших блокирующих (блокирующие функции возвращают управление вызывающему процессу только после того, как данные приняты или переданы или скопированы во временный буфер) MPI-предписа́ний MPI_Send/MPI_Recv, причем каждая операция повторяется много раз (с целью статистического усреднения).

Следует иметь в виду, что тестирование коммуникационной сети кластера на операциях ‘точка-точка’ является всего лишь важной, однако всего лишь частью общей процедуры тестирования (более полный набор тестов можно получить с http://parallel.ru/testmpi).

3.3 Необходимое оборудование

Оборудованием является вычислительный кластер под управлением UNIX-совместимой ОС, предустано́вленная поддержка MPI, рабочая консоль программиста для управления прохожде́нием пользовательских задач.

3.4 Порядок проведения работы

Студент подготавливает исходные тексты MPI-программ, компилирует их в исполняемое приложение, запускает на счет, анализирует выходные данные программы и представляет их в графическом виде.

Исходный код простой С-программы PROG_MPI.C тестирования производительности сети приведен ниже:

```
// исходный код программы PROG_MPI.C

#include "mpi.h" /* хидеры для MPI-функций */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

static int message[10000000]; // максимальная длина сообщения

int main (int argc, char *argv[])
{
 double  time_start,time_finish;
 int  i, n, nprocs, myid, len;
 MPI_Request rq;
 MPI_Status status;

 MPI_Init(&argc,&argv); /* инициализация MPI */
 MPI_Comm_size(MPI_COMM_WORLD,&nprocs); /* всего процессов в системе */
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);  /* номер текущего процесса */

 printf("PROGMPI: nprocs=%d myid=%d\n", nprocs, myid);
 fflush( stdout ); // принудительно ‘сбросить’ буфер на диск
 len = atoi( argv[1] ); /* длину сообщения в байтах взять из первого параметра
                           командной строки */
 printf( "length: %d\n", len );

 if (myid == 0) /* если это MASTER-процесс */
 {
  n = 1000; /* число циклов передачи сообщений для усреднения */
  MPI_Send(&n, 4, MPI_CHAR, nprocs-1, 99,MPI_COMM_WORLD);
  time_start=MPI_Wtime(); // запомнить текущий момент времени

  for (i=1; i<=n; i++) // цикл по всем n передачам сообщений
   {
    MPI_Send(message, len, MPI_CHAR, nprocs-1, 99, MPI_COMM_WORLD);
    MPI_Recv(message, len, MPI_CHAR, nprocs-1, 99, MPI_COMM_WORLD,
                                                          &status);
   }
   time_finish=MPI_Wtime();
   printf( "Time %f rate %f speed %f\n", (float)(time_finish-time_start),
                                         (float)(2*n/(time_finish-time_start)),
                                         (float)(2*n*len/(time_finish-time_start)) );
```

```
 } /* конец кода MASTER-процесса */

 else
 if (myid == (nprocs-1)) /* если это последний из процессов */
 {
  MPI_Recv(&n,4, MPI_CHAR, 0, 99, MPI_COMM_WORLD,&status);

  for (i=1; i<= n; i++)
   {
    MPI_Recv(message, len, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    MPI_Send(message, len, MPI_CHAR, 0, 99, MPI_COMM_WORLD);
   }
 } /* конец кода для последнего процесса */

fflush( stdout );
MPI_Finalize(); /* деинициализация MPI */

} // конец программы PROG_MPI
```

Для исполнения этой программы применяется команда (запуск на 5 вычислительных узлах при максимальном времени выполнения 15 мин, последнее число – размер сообщений в байтах):

```
mpirun -np 5 –maxtime 15 prog_mpi 1048576
```

Образец вы́дачи (вычислительный кластер кафедры ИТ-4 МГУПИ, задействованы 2 вычислительных узла):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

PROGMPI: nprocs=2 myid=0
PROGMPI: nprocs=2 myid=1
length: 1048576
length: 1048576
Time 182.223991 rate 10.975503 speed 11508649.257698
```

Экспериментальные данные сво́дятся в табл.3.1, причем принято строить зависимость пропускно́й способности сети (столбец Speed таблицы, Мбайт/сек) от длины сообщения (столбец Len, kбайт) в логарифмической (по основанию 2) шкале абсцисс.

Таблица 3.1. — Данные экспериментов по определению зависимости реальной пропускно́й способности коммуникационной сети кластера от длины передаваемых сообщений

№ №	*length (байт)*	*time (сек)*	*rate (сообщений/сек)*	*speed (байт/сек)*
1	1			
2	2			

3	4			
4	8			
5	16			
…				
17	65’536 *(0,0625 Mb)*			
18	131’072 *(0,125 Mb)*			
19	262’144 *(0,25 Mb)*			
20	524’288 *(0,5 Mb)*			
21	1’048’576 *(1 Mb)*			

Столбец rate показывает частоту обмена сообщениями в секунду. При малых длинах (единицы/десятки байт) сообщений латентность (в мксек) естественным образом вычисляется как $L=10^6/\text{rate}$ (ибо $\lim_{X\to 0}\left(\frac{X}{S}+L\right)=L$).

Данные табл.3.1 могут быть получены с использованием нижерасположенного скрипта (по 6 вычислительных узлов на каждый запуск программы prog_mpi):

```
#!/bin/bash

NP=6 # count of processors

# COMPILATION
mpicc -o prog_mpi -v prog_mpi.c >& prog_mpi.comp

# EXECUTION
for N in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536    \
         131072  262144 524288 1048576 # cycles by size of message (байты)
do
  mpirun -np $NP prog_mpi $N # parallel run
done
```

3.5 Оформление отчета по работе

По данным экспериментов требуется построить (‘от руки’ на ‘миллиметровке’ или средствами MS Excel) зависимость реальной скорости коммуникационной сети от размера сообщений, оценить величину латентности, сделать выводы о размере сообщений, при которых реализуется максимальная пропускная способность сети.

При построении зависимости скорости сети от размера сообщений выбрать кратную степени двойки шкалу по оси абсцисс (в соответствии со столбом length в табл.3.1) и линейную шкалу по оси ординат.

3.6 Вопросы для самопроверки

1. Чем отличается производительность сети при обмене сообщениями различной длины от заявленной изготовителем?
2. Каков физический смысл *латентности* и *цены́ обмена*? В каких единицах они измеряются? Какова *количественно* цена́ обмена при определенной Вами величине латентности?
3. Как зависит реальная производительность сети от размера передаваемых сообщений? При сообщениях какой длины производительность сети достигает максимального уровня?
4. Какова причина повышения латентности при *виртуализации сетей* управления и обмена данными?
5. Определите *экспериментально*, какое количество операций умножения (сложения, деления ‘плавающих’ чисел двойной точности) производит вычислительный узел Вашей МВС за промежуток времени, равный латентности? Оцените количественно *рациональный* размер гра́нулы (*зерна́, блока*) распараллеливания для предоставленного в Ваше распоряжение конкретного оборудования.

4 Лабораторная работа 4. **Простые MPI-программы (численное интегрирование)**

4.1 Цель работы

Целью работы является приобретение практических знаний и навыков в разработке несложных MPI-программ, анализе точности вычисляемых значений.

4.1 Теоретическая часть

Среди задач численного анализа встречается немало задач, распараллеливание которых очевидно. Например, численное интегрирование сводится фактически к (многочисленному) вычислению подинтегральной функции (что естественно доверить отдельным процессам), при этом главная ветвь управляет вычислениями (определяет стратегию распределения точек интегрирования по процессам и ‘собирает’ частичные суммы).

4.2 Необходимое оборудование

Для проведения работы необходим вычислительный кластер под управлением UNIX-совместимой ОС, предустано́вленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

4.2 Порядок проведения работы

Студент подготавливает исходные тексты MPI-программ, компилирует их в исполни́мое приложение, запускает на счет, анализирует выходные данные.

Часть 1 работы. Общеизвестно, что $\frac{\pi}{4} = arctg(1) - arctg(0) = \int_0^1 \frac{dx}{1+x^2}$. Заменяя вычисление интеграла коне́чным суммированием, получаем $\int_0^1 \frac{dx}{1+x^2} \approx \frac{1}{n}\sum_{j=1}^{j=n} \frac{1}{1+x_j^2}$, где $x_j = (j-0{,}5)/n$, n – число участков суммирования при численном интегрировании.

Площадь каждого участка (вертикальной ‘полоски’ - stripe) вычисляется функцией COMPUTE_INTERVAL как произведение ширины ‘полоски’ (width) на значение функции в центре ‘полоски’ (используется ‘метод прямоугольников‘ при интегрировании), далее площади суммируются главным процессом (применяется равномерная сетка).

С целью уясне́ния принципов распределения вычислений рекомендуется проанализировать текст функции compute_interval (здесь j – номер участка интегрирования, myrank – номер данного вычислительного узла, intervals – об-

щее число интервалов численного интегрирования, ntasks – общее число вычислительных узлов, значение локальных сумм накапливается в localsum):

```
localsum=0.0;
for (j=myrank; j<intervals; j+= ntasks) {
 x = (j + 0.5) * width;
 localsum += 4 / (1 + x*x);
}
```

Заметим, что нагрузка каждого процесса вычислением конкретного (единственного) значения столь простой подинтегральной функции было бы крайне нерациональным решением, т.к. время обмена сообщениями (посылка главным процессом значения параметра функции и прием вычисленного значения) сравни́мо со временем вычисления функции и существенно повысило бы время выполнения программы (чрезмерно много коротких пересы́лок); подобный подход является примером излишне тонкозернистого (*fine-grained*) параллелизма, поэтому каждый процесс вычисляет много (общее число, деленное на число процессор без единицы – т.к. нулевой процесс назначается управляющим – MASTER'ом) подинтегральных сумм. В целом следует стремиться к использованию возможно *ме́ньшего количества максимально длинномерных* обменов.

В качестве функции времени совместно с MPI_Wtime применяется стандартная С-функция *локального для вычислительного узла* времени ftime (функция f_time). В программе использованы исключительно (блокирующие) функции обмена 'точка-точка' MPI_Send/MPI_Recv, активно используется барьерная функция MPI_Barrier (при вызове MPI_Barrier(MCW) управление в вызывающую программу не возвращается до тех пор, пока все процессы группы не вызовут MPI_Barrier(MCW), где MCW – коммуникатор). Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован.

```
// исходный текст программы PI_01.C

#include "mpi.h" /* хидер MPI-функций */
#include <stdio.h>
#include <math.h>
#include <sys/timeb.h> // хидер функции  ftime

double f_time(void); /* прототип функции ftime  */
#include "f_time.c"

double compute_interval (int myrank, int ntasks, long intervals);

int main(int argc, char ** argv)
```

```
{
 long intervals=100; // число участков для интегрирования
 int i, myrank,ranksize;
 double pi, di, send[2], recv[2],
          t1,t2, t3,t4, t5,t6,t7, // моменты времени
          pi_prec=4.0*(atanl(1.0)-atan(0.0)); // точное значение π для сравнения
 MPI_Status status;

 t1=f_time(); /* зафиксировать момент времени */
 MPI_Init (&argc, &argv);  /* инициализация MPI-системы */
 t2=f_time();
 MPI_Comm_rank (MPI_COMM_WORLD, &myrank); /* номер текущего процессора */
 MPI_Comm_size (MPI_COMM_WORLD, &ranksize); /* общее число процессоров */

 if (myrank == 0)  /* если это главный ( MASTER ) – процесс */
 {
 intervals = atoi( argv[1] ); /* взять число аргументов командной строки */
 printf ("Calculation of PI by numerical integration with %ld intervals\n", intervals);
 }

 MPI_Barrier(MPI_COMM_WORLD); /* установить барьер вычислений */

 if (myrank == 0) /* если это MASTER-процесс */
 {  /* распределение данных по процессам */
 printf ("Master: Sending # of intervals to MPI-Processes \n");
 t3 = MPI_Wtime();
 for (i=1; i<ranksize; i++)  /* цикл по процессам */
  MPI_Send (&intervals, 1, MPI_LONG, i, 98, MPI_COMM_WORLD);
 } /* конец работы MASTER-процесса */

 else   /* если это подчиненный ( рабочий, SLAVE ) процесс */
 { /* прием данных от MASTER'ра */
 MPI_Recv (&intervals, 1, MPI_LONG, 0, 98, MPI_COMM_WORLD, &status);
 } // конец работы SLAVE-процесса

 /* вычисление данных на заданном интервале */
 t4 = MPI_Wtime(); /* зафиксировать момент времени */
 pi = compute_interval (myrank, ranksize, intervals); /* вызов compute_interval */
 t5 = MPI_Wtime();
 MPI_Barrier (MPI_COMM_WORLD); /* все процессы выполняются? */
 t6 = MPI_Wtime();

 if (myrank == 0) /* если это MASTER-процесс */
{  /* сбор результатов, их суммирование и вывод */
 for (i=1; i<ranksize; i++)
 {
  MPI_Recv (&di, 1, MPI_DOUBLE, i, 99, MPI_COMM_WORLD, &status);
  pi += di;
 } /* конец сбора результатов вычислений со SLAVE-процессов */

 t7 = MPI_Wtime();
 printf ("Master: Has collected sum from MPI-Processes \n");
```

```
 printf ("\nPi estimation: %.12lf (rel.error= %.5f %%)\n",
          pi, 1.0e2*(pi_prec-pi)/pi_prec);
 printf ("%ld tasks used,  execution time: %.3lf sec\n",ranksize, t7 –t3);
 printf("\nStatistics:\n");
 printf("Master: startup: %.0lf msec\n",t2-t1);
 printf("Master: time to send # of intervals:%.3lf sec\n",t4-t3);
 printf("Master: waiting time for sincro after calculation:%.2lf sec\n",t6-t5);
 printf("Master: time to collect: %.3lf sec\n",t7-t6);
 printf("Master: calculation time:%.3lf sec\n",t5-t4);

 MPI_Barrier (MPI_COMM_WORLD); /* все процессы выполняются? */

 for (i=1; i<ranksize; i++) /* цикл по SLAVE-процессам */
 {  /* сбор времени вычислений со всех процессов, кроме MASTER'a */
 MPI_Recv(recv, 2, MPI_DOUBLE, i, 100, MPI_COMM_WORLD, &status);
 printf("process %d: calculation time: %.3lf sec,\twaiting time for sincro.: %.3lf sec\n",
          i,recv[0],recv[1]);
 } // конец цикла по SLAVE-процессам

} // конец работы MASTER-процесса

 else /* это SLAVE-процесс */
{ /* отсылка результатов вычислений назад MASTER-процессу */
 MPI_Send (&pi, 1, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD);
 MPI_Barrier (MPI_COMM_WORLD); /* все процессы отработали? */
 send[0]=t5-t4;
 send[1]=t6-t5;
 MPI_Send (send, 2, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
} // конец SLAVE-процесса

 MPI_Finalize ();
} // конец  функции main

/* ============================================================= */
double compute_interval (int myrank, int ntasks, long intervals)
{ /* вычисление локальных сумм для интегрирования */
 double x, width, localsum=0.0;
 long j;
 width = 1.0 / intervals; /* ширина интервала -  stripe */
 for (j=myrank; j<intervals; j+= ntasks)
 {
  x = (j + 0.5) * width;
  localsum += 4 / (1 + x*x);
 }
 return (localsum * width);  /* площадь прямоугольника  */
} // конец функции compute_interval
// конец программы PI_01.C
```

Функция f_time содержится в файле F_TIME.C (подключается посредством #include “f_time.c”):

```
double f_time() /* возвращается время с 01 января 1970 г. в виде вещественного
                   числа двойной точности с форматом ‘секунды.миллисекунды’ */
{
 struct timeb tp;
 ftime(&tp); /* штатная С-функция */
 return ((double)(tp.time)+1.0e-3*(double)(tp.millitm));
} // конец функции f_time
```

Возможно точное (double при данной разрядной сетке ЭВМ) значение π определяется как 4.0×(atanl(1.0)-atanl(0.0)) и сравнивается с вычисленным (относительная ошибка rel.error выводится в процентах).

Образец вы́дачи программы приведен ниже (запуск на процессорах кластера кафедры ИТ-4 МГУПИ):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

   Calculation of PI by numerical Integration with 10000 intervals
   Master: Sending # of intervals to MPI-Processes
   Master: Has collected sum from MPI-Processes

   Pi estimation: 3.141592654423 (rel.error= -0.00001 %)
   5 tasks used – Execution time: 0.017 sec

   Statistics:
   Master: startup: 4243 msec
   Master: time to send # of intervals: 0.000 sec
   Master: waiting time for sincro after calculation: 0.00 sec
   Master: time to collect: 0.000 sec
   Master: calculation time: 0.017 sec
   process 1: calculation time: 0.016 sec,  waiting time for sincro.: 0.001 sec
   process 2: calculation time: 0.017 sec,  waiting time for sincro.: 0.000 sec
   process 3: calculation time: 0.016 sec,  waiting time for sincro.: 0.001 sec
   process 4: calculation time: 0.015 sec,  waiting time for sincro.: 0.002 sec
```

Индивидуальные задания включает определение числа интервалов интегрирования, необходимое для представления числа π с заданной относительной точностью (линейка 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}), что реализуется методом повторных расчетов при увеличивающемся intervals).

4.3 Дополнительные задания (самостоятельная работа студентов)

- Обеспечить автоматический выбор оптимального числа интервалов вычисления интеграла (методом удвоения начального значения intervals до момента непревышения точности вычислений точности представления double-вещественного числа).
- Оценить возможное ускорение вычислений при увеличении числа процессоров (‘прогнать’ программу на числе процессоров N=2,3,4,5,6,7,8… с

одинаковым числом intervals), при этом оценить точность вычисления числа π.

- Изменить программу вычисления определенного интеграла путем замены функций обмена ‘точка-точка’, коллективными функциями MPI_Bcast (передача ‘один-всем’ из ветви 0, функция передает число интервалов всем параллельным ветвям) и MPI_Reduce (получение данных от всех ветвей и суммирования их в ветви 0).

Часть 2 работы. Для вычисления значения π можно использовать метод ‘стрельбы’ (рис.4.1) – вариант метод Монте-Карло. В применении к данному случаю метод заключается в генерации *равномерно распределенных* на двумерной области $[0 \le x \le 1,\ 0 \le y \le 1]$ точек и определении $\pi = 4 \times \frac{S_{OAC}}{S_{OABC}} \approx 4 \times \frac{score}{darts}$, где S – площади фигур на рис.4.1, score – число попавших внутрь четверти окружности (фигура 0AC) точек (условие $x^2+y^2 \le 1.0$, на рис.4.1 удовлетворяющие этому условию точки обозначены квадратами), darts – общее число точек (‘выстрелов’, throws). Вычисленное таким образом значение π является приближенным, в общем случае точность вычисления иско́мого значения повышается с увеличением числа ‘выстрелов’ и качества датчика случайных чисел; подобные методы используются в случае трудностей точной числовой оценки (например, для вычисления определенных интегралов большой кратности).

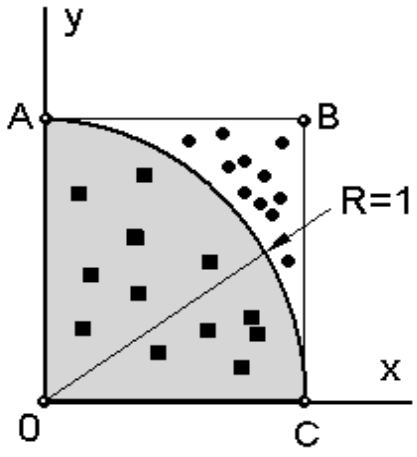

Рисунок 4.1 — Определение величины π методом ‘стрельбы’

Именно функция dboard (‘мишень’) генерирует DARTS пар равномерно распределенных на отрезке $[0 \div 1]$ случайных чисел, проверяет принадлежность нахождения точки (определяемой координатами последней двойки случайных чисел) внутренней области четверти окружности и вычисляет текущее значение π; расчеты проводятся при начальном значении DARTS с последующим увеличением ROUNDS раз (за́даны посредством #define).

Последовательный вариант программ вычисления π методом ‘стрельбы’ приведен ниже (файл pi_ser.c, функция dboard расположена в файле dboard.c и подключается посредством #include ‘dboard.c’):

```
// исходный код последовательной программы PI_SER.C

#include <stdio.h>
#include <math.h>

#include <sys/timeb.h>
double f_time(void);
```

```
void srandom (unsigned seed);
long random(void);
double dboard (int darts);

#include “dboard.c” // включение файла dboard.c
#include “f_time.c”  // включение файла f_time.c

#define DARTS    10000  /* число пробных выстрелов в dartboard */
#define ROUNDS 100     /* число итераций */

int main(int argc, char *argv[])
{
double pi,         /* среднее значение π после darts */
          avepi,   /* среднее значение π  для всех итераций */
          t1, t2,   // моменты времени
          pi_prec=4.0*(atanl(1.0)-atan(0.0)); // точное значение π  для сравнения
int i, n;

t1=f_time(); // фиксируем момент старта вычислений
srandom (5); // инициализация генератора случайных чисел
avepi = 0.0;
for (i=0; i<ROUNDS; i++) // цикл по числу вызовов функции dboard
{
 pi = dboard(DARTS);
 avepi = ((avepi * i) + pi)/(i + 1);
 t2=f_time();
 printf(“%7d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n”,
          (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
} // конец цикла for (i=0; i<ROUNDS; i++)
} // конец функции main
// конец программы PI_SER.C
```

Текст функции DBOARD (подключается посредством #include “dboard.c”):

```
double dboard(int darts)
{
double x_coord,    /* x-координата, промежуток –1 и 1  */
          y_coord,    /* y-координата, между –1 и 1  */
          pi,
          r;              /* случайное число между 0 и 1  */
int score,   /* число ‘выстрелов’ в цикле */
          n;
unsigned long cconst;

cconst = 2 << (31 – 1); /* используется для конвертации целого в диапазоне
                                  0 и 2**31 в вещественное между 0 и 1.0 */
score = 0; // инициализация счетчика

for (n=1; n<=darts; n++)  // циклы по генерации случайных чисел x и y
{
 r = (double)random() / cconst;
```

```
 x_coord = (2.0 * r) – 1.0; // случайная x-координата
 r = (double)random() / cconst;
 y_coord = (2.0 * r) – 1.0; // случайная y-координата
 if (((x_coord*x_coord) + (y_coord*y_coord)) <= 1.0) /* x и y находятся  внутри
                                                       окружности R радиусом 1.0 ? */
   score++;
} // конец цикла for (n=1; n<=darts; n++)
 pi = 4.0 * (double)score / (double)darts;
 return(pi);
} // конец функции dboard
```

При первом варианте распараллеливания (программа pi_02.c) используются только (блокирующие) функции обмена ‘точка-точка’ MPI_Send/MPI_Recv: вычислительные узлы посылают MASTER-узлу частичные (определенные на основе выполненных данным узлом ‘выстрелов’) значения π, главный процесс суммирует их и выдает на печать усредненные значения:

```
// исходный код параллельной программы PI_02.C

#include “mpi.h”
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define DARTS    10000
#define ROUNDS 100
#define MASTER   0   /* номер главного процесса */

void srandom (unsigned seed);
double dboard (int darts);
#include “dboard.c” // включение прототипа функции dboard.c

int main(int argc, char *argv[])
{
double homepi,   /* текущее значение числа π   */
        pi,          /* среднее для π  после darts выстрелов */
        avepi,       /* среднее для π  для всех итераций */
        pirecv,      /* значение π  в рабочих узлах */
        pisum,       /* накопленное значение π  для рабочих узлов */
        t1, t2, // моменты времени
        pi_prec=4.0*(atan(1.0)-atan(0.0)); // точное значение π
int     taskid,      /* номер процесса */
        numtasks,
        source,      /* номер процесса, посылающего сообщения */
        rc,          /* возвращаемый код для MPI-функций */
        i, n;
MPI_Status status;

rc = MPI_Init(&argc,&argv);
rc = MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
```

```
rc = MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

t1=MPI_Wtime(); // запомнили момент начала вычислений

srandom (taskid); // инициализация датчика случайных чисел номером процесса

avepi = 0.0;
for (i=0; i<ROUNDS; i++)
{
 homepi = dboard(DARTS); /* все процессы вычисляют π , используя
                            алгоритм dartboard */

if (taskid != MASTER) // это не MASTER-процесс
{
 rc = MPI_Send(&homepi, 1, MPI_DOUBLE, MASTER, i, MPI_COMM_WORLD);
 if (rc != MPI_SUCCESS) // если ошибка MPI_Send - функции
  printf("%d: Send failure on round %d\n", taskid, i);
}
else // это MASTER-процесс
{
 pisum = 0.0;
 for (n=1; n<numtasks; n++)
 {
  rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE, i,
                MPI_COMM_WORLD, &status);
  if (rc != MPI_SUCCESS) // if error MPI_Recv function
    printf("%d: Receive failure on round %d\n", taskid, i);
  pisum += pirecv;
 }
/* вычисление среднего значения π для данной итерации */
 pi = (pisum + homepi) / numtasks;
/* вычисление среднего значения π для всех итераций */
 avepi = ((avepi * i) + pi) / (i + 1);
 t2=MPI_Wtime(); // фиксируем время конца вычислений
 printf("%9d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n",
        (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
  }
} // конец цикла (i=0; i<ROUNDS; i++)
 MPI_Finalize();
 return 0;
} // конец функции main
// конец программы of PI_02.C
```

Выдача программы (вычислительный кластер кафедры ИТ-4, число 'выстрелов' от 1000 до 10'000) приведена ниже (напомним, что каждое вычисленное значение следует рассматривать как одну из реализаций случайной величины, нормально распределенной вокруг истинного значения π):

```
LAM 7.0.6/MPI 2 C++ - Indiana University

10000 throws aver.value of PI= 3.119600000000 (rel.error= 0.70005 %, time= 0.003 sec)
```

```
20000 throws aver.value of PI= 3.137800000000 (rel.error= 0.12072 %, time= 0.006 sec)
30000 throws aver.value of PI= 3.140000000000 (rel.error= 0.05070 %, time= 0.037 sec)
40000 throws aver.value of PI= 3.144100000000 (rel.error= -0.07981 %, time= 0.040 sec)
50000 throws aver.value of PI= 3.143440000000 (rel.error= -0.05880 %, time= 0.043 sec)
60000 throws aver.value of PI= 3.139333333333 (rel.error= 0.07192 %, time= 0.046 sec)
70000 throws aver.value of PI= 3.141200000000 (rel.error= 0.01250 %, time= 0.048 sec)
80000 throws aver.value of PI= 3.145250000000 (rel.error= -0.11642 %, time= 0.051 sec)
90000 throws aver.value of PI= 3.144933333333 (rel.error= -0.10634 %, time= 0.053 sec)
…
100000 throws aver.value of PI= 3.143760000000 (rel.error= -0.06899 %, time= 0.056 sec)
…
200000 throws aver.value of PI= 3.143840000000 (rel.error= -0.07154 %, time= 0.082 sec)
…
400000 throws aver.value of PI= 3.142310000000 (rel.error= -0.02283 %, time= 0.133 sec)
…
800000 throws aver.value of PI= 3.143230000000 (rel.error= -0.05212 %, time= 0.236 sec)
…
1000000 throws aver.value of PI= 3.142872000000 (rel.error= -0.04072 %, time= 0.293 sec)
```

Видно, что рассчитанные значения π являются весьма приближенными даже при большом числе ‘выстрелов’, причем не во всех случаях увеличение числа ‘выстрелов’ вызывает возрастание точности представления иско́мой величины.

Ниже приведен вариант программы (файл pi_03.c) с использованием коллективных функций (вместо функций семейства ‘точка-точка’); при этом в MASTER-процессе функция MPI_Reduce суммирует величины homepi от всех зада́ч (сумма помещается в pisum; homepi при этом является буфером посы́лки, pisum – приемным буфером):

```
// исходный код программы PI_03.C

#include “mpi.h”
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define DARTS    10000
#define ROUNDS 100
#define MASTER   0

void srandom (unsigned seed);
double dboard (int darts);
#include “dboard.c”

int main(int argc, char *argv[])
{
double homepi, pisum, pi, avepi,
          t1, t2, // time’s moments
          pi_prec=4.0*(atanl(1.0)-atan(0.0));
int       taskid, numtasks, rc, i:
```

```
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

t1=MPI_Wtime();

avepi = 0;
for (i=0; i<ROUNDS; i++)
{
 homepi = dboard(DARTS);
 rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM, MASTER,
                      MPI_COMM_WORLD);
 if (rc != MPI_SUCCESS)
  printf("%d: failure on MPI_Reduce\n", taskid);

if (taskid == MASTER)
{
  pi = pisum/numtasks;
  avepi = ((avepi * i) + pi)/(i + 1);
  t2=MPI_Wtime();
  printf("%9d throws aver.value of PI= %.12lf (rel.error= %.5lf %%, time= %.3lf sec)\n",
         (DARTS * (i+1)), avepi, 1.0e2*(pi_prec-avepi)/pi_prec, t2-t1);
}
} // конец цикла for (i=0; i<ROUNDS; i++)
 MPI_Finalize();
 return 0;
} // конец функции main
// конец программы PI_03
```

При выполнении второй части работы следует скомпилировать варианты PI_SER (на кластере кафедры ИТ-4 МГАПИ команда компиляции и разрешения ссылок icc –o pi_ser pi_ser.c, запуска на исполнение с выдачей данных в файл pi_ser.out – pi_ser > pi_ser.out), PI_02 и PI_03 программы вычисления значения π, 'прогнать' их в режиме выполнения на одном и нескольких процессорах, проанализировать выдачу.

Индивидуальные задания включает определение числа интервалов интегрирования, необходимое для представления числа π с заданной относительной точностью (линейка 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}), что реализуется путем повторных расчетов при увеличении числа 'выстрелов'.

4.5 Дополнительные задания (самостоятельная работа студентов)

- Оценить снижение времени вычисления π параллельной версией программы по отношению к последовательной (файл PI_SER.C) при различных числах 'выстрелов'.
- Оценить возможное ускорение вычислений при увеличении числа процессоров ('прогнать' программу на числе процессоров N=3,4,5,6,7,8… с одинаковым числом 'выстрелов').

4.6 Вопросы для самопроверки

1. Привести примеры априори хорошо распараллеливающихся алгоритмов. В чем заключается условие близкой к идеальной распараллеливаемости?
2. Будет ли результат численного интегрирования (программа PI_01.C) монотонно стремиться к точному значению соответствующего определенного интеграла при неограниченном увеличении числа интервалов интегрирования?
3. Каким образом и от каких параметров датчика случайных чисел при методе ‘стрельбы’ зависит точность вычисления определенного интеграла? Какой оценки числа π следует ожидать при *статистически неравноме́рном распределении* координат точек ‘выстрелов’: а) в случае расположения максимума гистограммы распределения вблизи 0? б) вблизи 1? в) ро́вно в центре отрезка $[0 \div 1]$?

5 Лабораторная работа 5. **Изучение зависимости производительности многопроцессорной вычислительной системы от числа вычислительных узлов и размера обрабатываемых данных**

5.1 Цель работы

Целью работы является уяснение принципов распределения вычислений по вычислительным узлам кластера, приобретение практических знаний и навыков в создании MPI-программ реализации стандартных алгоритмов, исследование зависимости скорости вычислений от числа вычислительных узлов и размера обрабатываемых данных.

Данная работа фактически является исследовательской и может быть предложена для обеспечения научно-исследовательского практикума для студентов $3 \div 4$ курсов.

5.2 Теоретическая часть

Операции с матрицами большой (тысячи/сотни тысяч) размерности являются основой многих численных методов (например, МКЭ – *метода конечных элементов*). Только одна (далеко не самая крупная) квадратная матрица размером $3'000 \times 3'000$ double-чисел требует для размещения 72 Мбайт ОП (предполагаем, что матрица является плотнозаполненной – т.е. число нулевых элементов мало; иначе потребуются специальные методы хранения ненулевых элементов). Ясно, что в этом случае программист обязан распределить по процессорам кластера не только вычислительные процедуры, но и данные (те же матрицы); использование для хранения матриц дисковой памяти катастрофически замедляет программу.

Одна из простейших матричных операций – умножение. Известно, что в случае [C] = [A]×[B] значение элементов результирующей матрицы вычисляется как (*классический* способ):

$$c_{ij} = \sum_{k=1}^{k=NCA} a_{ik} b_{kj},$$

где $i=1 \div NRA$ – строки матриц [A] и [C],
$j=1 \div NCB$ – столбцы матриц [B] и [C],
$k=1 \div NCA$ – столбцы матрицы [A] и строки матрицы [B].

Хотя известны более эффективные алгоритмы (напр., *умножение матриц по Винограду* и *рекурсивный алгоритм Штрассена умножения квадратных матриц*), в данной работе используется именно стандартный (требующий N^3 операций умножения и N^3-N^2 сложений, где N – характерный размер матрицы).

Дальнейшим развитием (в сторону минимизации числа пересылок данных) ленточного алгоритма умножения матриц является алгоритм Фокса (Geoffrey Fox, et al., *Solving Problems on Concurrent Processors*, Englewood Cliffs, NJ, Prentice-Hall, 1998) и алгоритм Кэнно-

на (в этом случае процессам пересылаются не ленты, а *прямоугольные блоки* исходных матриц; усложнением при этом является необходимость дополнительного межпроцессного обмена, ибо на основе прямоугольных блоков [A] и [B] невозможно вычислить такой же блок [C] – необходим циклический сдвиг фронта вычислений).

Алгоритм Фокса иногда именуется *клеточным* (ибо он основан на распределении блоков матриц по ОП процессов по *принципу шахматной доски*), этот алгоритм описан также в работах [1,3]; ограничением его является (непременное) условие $M=Q^2$ и $N \% Q = 0$ (N – порядок перемножаемых матриц, M – число процессов, виртуально распределенных на квадратную сетку размеров $Q \times Q$, причем каждый процесс связан с блоком размером $N/Q \times N/Q$ элементов матриц). Алгоритм Кэннона отличается более разумным распределением блоков исходных матриц по процессорам, при этом упрощается схема обмена данными между последними.

Например, при умножении матриц размером 6×6 по алгоритму Фокса при 9 процессах каждому из них передаются подматрицы размером $N' = N/Q = N/\sqrt{M} = 2$ следующим образом (процессоры объединены квадратной сеткой размерности Q, нумерация везде начинается с 0):

Процесс 0	Процесс 1	Процесс 2
$A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	$A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$	$A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{11} \end{pmatrix}$
Процесс 3	Процесс 4	Процесс 5
$A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	$A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	$A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$
Процесс 6	Процесс 7	Процесс 8
$A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$	$A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{52} & a_{53} \end{pmatrix}$	$A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$

В общем случае действия перемножения матриц ‘по Фоксу’ иллюстрируются следующим алгоритмом:

```
for(step=0; step<Q; step++)
{
```
1. Выбрать подматрицу матрицы [A] в каждой строке процессов (в *r*-той строке находится подматрица A_{ru}, где $u=(r+step) \bmod Q$).
2. Для процессов каждой строки переслать всем иным процессам той же строки выбранную подматрицу.
3. Каждый процесс умножает полученную подматрицу матрицы [A] на находящуюся в ОП процесса подматрицу матрицы [B].
4. Для каждого процесса переслать подматрицу матрицы [B] процессу, находящемуся в решетке выше (из первой строки пересылка идет в последнюю строку).

```
}
```

Тривиальная программа MM_SER.C умножения матриц стандартным способом (используется последовательный вариант вычислений в одном процессе) приведена ниже:

```
// исходный код программы MM_SER.C

#include <stdio.h>

#include <sys/timeb.h> // хидер для функции ftime
double f_time(void); /* прототип для функции f_time(void) */
#include “f_time.c”

#define NRA 3000  /* число строк матрицы A */
#define NCA 3000  /* число столбцов матрицы A */
#define NCB 10      /* число столбцов матрицы B */

int main(int argc, char *argv[])
{
int    i, j, k; /* индексы */
double a[NRA][NCA],        /* двумерный массив для размещения матрицы A */
          b[NCA][NCB],        /* двумерный массив для размещения матрицы B */
          c[NRA][NCB],        /* двумерный массив для результирующей матрицы C */
          t1,t2; // time’s momemts

/* инициализация элементов матриц A, B и C */
 for (i=0; i<NRA; i++)
  for (j=0; j<NCA; j++)
    a[i][j] = i+j; // возможно использование генератора случайных чисел
 for (i=0; i<NCA; i++)
  for (j=0; j<NCB; j++)
    b[i][j] = i*j;
 for(i=0; i<NRA; i++)
  for(j=0; j<NCB; j++)
   c[i][j] = 0.0;

t1=f_time(); // запомнить время начала вычислений

/* собственно гнездо циклов для умножение матриц */
 for(i=0; i<NRA; i++)
  for(j=0; j<NCB; j++)
    for(k=0; k<NCA; k++)
      c[i][j] += a[i][k] * b[k][j];

t2=f_time(); // время конца вычислений

printf (“Multiplay time= %.3lf sec\n\n”, t2-t1);
printf(“Here is the result matrix:\n”);
 for (i=0; i<NRA; i++)
  {
   printf(“\n”);
   for (j=0; j<NCB; j++)
    printf(“%6.2f  “, c[i][j]);
   }
 printf (“\n”);

} // конец программы MM_SER.C
```

Теоретически (без учета затрат времени на обмен данными) задача умножения матриц распараллеливается идеально (выходные данные вычисляются по единому алгоритму на основе исходных и в процессе вычисления не изменяют последних), однако при практическом распараллеливании (разработ-

ке программы) необходимо учитывать как задержки времени при обменах, так и распределение по вычислительным узлам вычислений и блоков данных. Т.о. стратегия распределения блоков матриц по ВУ должна основываться на анализе структуры процесса вычисления (т.н. *тонкой информационной структуре алгоритма*, [1]); в профессиональных пакетах линейной алгебры (ScaLAPACK, AZTEC) распределению содержимого обрабатываемых матриц по вычислительным узлам уделяется серьезное внимание. В данной работе задача распределения содержимого матриц по ВУ не решается, *распараллеливаются только вычисления*.

Лежащий на поверхности способ распараллеливания вычислений заключается в перемножении каждым процессом значений a_{ik} и b_{kj} и суммировании главным процессом этих частичных сумм для получения c_{ij}. Однако этот подход наименее рационален по производительности, ибо каждое умножение (а их число пропорционально N^3) сопровождается минимум 3 обме́нами, причем длительность каждого намного превышает длительность операции умножения. Кроме того, при таком способе велика *избыточность пересылок* (и a_{ik} и b_{kj} участвуют в формировании не одного c_{ij}, а многих).

Более рационально в каждом процессе вычислять произведение *i*-той строки [A] и *j*-того столбца [B], в результате сразу получаем готовое значение c_{ij}; и при этом число обменов (пропорциональное N^2) все еще излишне высоко́. Известно, что в языке C/C++ элементы матрицы располагаются в ОП по строкам (в Fortran'е – по столбцам), поэтому в C пересы́лка строк записывается тривиально, а обмен столбцами потребует некоторых ухищрений.

Еще более эффективно пересылать каждому процессу *серию строк (ленту) матрицы* [A] и *серию столбцов* [B]; этим реализуется крупнозернистый (*coarse-grained*) параллелизм (рис.5.1), а метод умножения матриц именуется *ленточным*. Каждый процесс при этом вычисляет (определяемый условием $c_{ij}, i \in [i1 \ldots i1], i \in [j1 \ldots j2]$) прямоугольный блок матрицы [C]. При программировании на C для пересы́лки процессорам вертикальной ленты [B] необходимо использовать временный (рабочий) массив, при программировании на Fortran'е - то же для горизонтальной ленты [A]. Заметим, что нахождение в ОП процесса горизонтальной ленты [A] дает возможность вычислить не только прямоугольный блок $c_{ij}, i \in [i1 \ldots i1], i \in [j1 \ldots j2]$, но и всю вертикальную ленту $c_{ij}, i \in [1 \ldots NRA], i \in [j1 \ldots j2]$, см. рис.5.1; для этого необходимо дополнительно пересылать только́ новые горизонтальные ленты [A]. При программировании на C для пересы́лки процессорам вертикальной ленты [B] придется использовать временный (рабочий) массив, при программировании на Fortran'е - то же для горизонтальной ленты [A].

Несколько (4 штуки) вариантов распараллеливания классического метода умножения матриц приведены в табл. 5.1; обозначения: КУ – координирующий (MASTER) узел, ВУ – вычислительный (WORKER) узел, N – число трок/столбцов матрицы, P – число ВУ, L – размер обрабатываемых чисел в байтах, L=sizeof(double)).

Таблица 5.1 — Различные методы распараллеливания классического алгоритма умножения матриц

№ №	*Описание алгоритма*	*Определение необходимого количества пересы́лок данных (обме́нов; при этом пересы́лка M чисел в едином пакете считается единичной)*	*Результат (необходимое число пересы́лок)*
1	1. КУ посылает каждому ВУ элементы a_{ik} и b_{kj} 2. ВУ их перемножает и отправляет полученное произведение назад КУ 3. КУ складывает произведения (фактически *частичные суммы*) для получения готового c_{ij}	▫ Для вычисления каждого произведения $a_{ik} \times b_{kj}$ требуется 2 пересы́лки и еще 1 для отправления произведения назад (все размером L) ▫ Для вычисления каждого c_{ij} необходимо сделать вышеописанное N раз ▫ Необходимо вычислить N^2 штук c_{ij}; т.о. всего требуется $3 \times N^3$ пересы́лок данных	$3 \times N^3$ чисел
2	1. КУ посылает каждому ВУ i-тую строку матрицы [A] и k-тый столбец матрицы [B] 2. ВУ перемножает элементы строки [A] на соответствующие элементы [B] 3. ВУ отсыла́ет готовое c_{ij} на КУ	▫ Для вычисления каждого c_{ij} требуется 2 пересы́лки (строка [A] и столбец [B], длиной N*L) плюс 1 для отправления готового c_{ij} (длиной L) назад на КУ ▫ Т.к. необходимо вычислить N^2 штук c_{ij}; всего требуется $3 \times N^2$ пересы́лок данных	$3 \times N^2$ (но все равно пересылаются $3 \times N^3$ чисел)

3	1. КУ посылает каждому ВУ (N/P) строк (т.е. *горизонтальную ленту*) матрицы [A] и (M/P) столбцов (т.е. *вертикальную ленту*) матрицы [B] (M - число столбцов матрицы [B]), если M # N) 2. ВУ перемножает элементы присланных лент строк [A] на соответствующие элементы лент столбцов [B] 3. ВУ отсылает готовый прямоугольный блок (размером [N/P]×[M/P]) элементов c_{ij} на КУ	▫ Для вычисления каждого прямоугольного блока (размером [N/P]×[M/P]) элементов c_{ij} требуется 2 пересы́лки (длиной N^2*L/P) плюс 1 (длиной N^2/P^2) для отправления назад ▫ Таких действий необходимо сделать P раз (по числу КУ) ▫ Всего получаем 3×P пересы́лок	3×P (но все равно пересылаются $3\times N^3$ чисел)
3a	То же самое, но *горизонтальная лента* [A] пересылается один раз, а потом (N/P) раз пересылаются *вертикальные ленты* [B]	▫ Все то же самое, но каждая пересылка ≈ вдвое короче (т.е. экономится время на пересылках данных от КУ к ВУ)	-

Данные табл.5.1 подтверждают положение о том, что практический любой изначально последовательный алгоритм может быть распараллелен *многими вариантами*, причем производительность каждого сильно различается (целесообразно выбирать вариант с максимальным размером гранулы параллелизации и минимумом пересылок между ВУ).

Т.к. обычно число столбцов матрицы [B] много меньше числа строк [A] (часто NCB=1 и матрица [B] фактически является вектором), в приведенной ниже программе MM_MPI.C матрица [B] пересылается всем процессам целиком.

Заметим, что в программе явно указана размерность перемножаемых матриц (идентификаторы NRA, NCA и NCB), но не число вычислительных узлов (оно задается в момент запуска программы, а распределение вычислений по узлам обеспечивается автоматически).

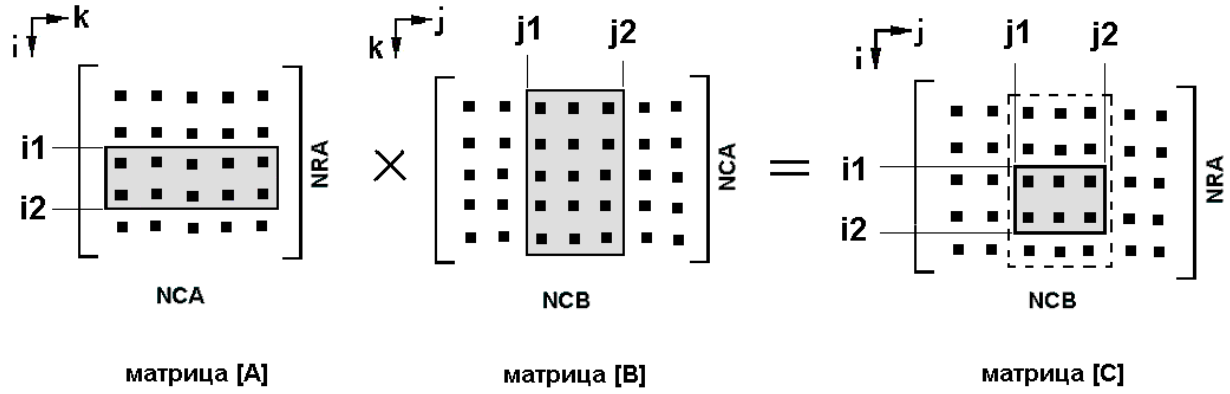

Рисунок 5.1 — Ленточная схема умножения матриц (пересылаемые ленты и вычисленный блок результирующей матрицы выделены серым фоном)

```
// исходный код программы MM_MPI.C

// Ros Leibensperger / Blaise Barney. Converted to MPI: George L.
#include “mpi.h”
#include <stdio.h>

#define NRA 1000 /* число строк матрицы A */
#define NCA 1000 /* число столбцов матрицы A */
#define NCB 1000 /* число столбцов матрицы B */

#define MASTER 0 // номер MASTER-процесса
#define FROM_MASTER  1  /* идентификатор сообщений ОТ MASTER-процесса */
#define FROM_WORKER 2  /* идентификатор сообщений ОТ WORKER-процесса */

#define M_C_W  MPI_COMM_WORLD // чтобы писать поменьше…

int main(int argc, char *argv[])
{
  int numtasks,      /* number of tasks in partition */
      taskid,        /* идентификатор процесса */
      numworkers,    /* число WORKER-процессов */
      source,        /* идент. сообщения источника */
      dest,          /* идент. сообщения приемника */
```

```
      rows,              /* число строка матрицы [A], посылаемых каждому процессу */
        averow, extra, offset,
        i, j, k, rc;  /* индексные переменные */
double a[NRA][NCA],  /* двумерные массивы для хранения матриц */
         b[NCA][NCB],
         c[NRA][NCB],
         t1,t2;
MPI_Status status;

 rc = MPI_Init(&argc,&argv);
 rc|= MPI_Comm_size(M_C_W, &numtasks);
 rc|= MPI_Comm_rank(M_C_W, &taskid);
 if (rc != MPI_SUCCESS)
   printf (“error initializing MPI and obtaining task ID information\n”);
 else
   printf (“task ID = %d\n”, taskid);
 numworkers = numtasks-1;
/*************************** MASTER-процесс ***********************************/
  if (taskid == MASTER)
   {
     printf(“Number of worker tasks = %d\n”,numworkers);
     for (i=0; i<NRA; i++)
       for (j=0; j<NCA; j++)
         a[i][j]= i+j;
     for (i=0; i<NCA; i++)
       for (j=0; j<NCB; j++)
         b[i][j] = i*j;

/* отсылка части матриц WORKER-процессам */
     averow = NRA/numworkers; // среднее число строк/процесс
     extra = NRA%numworkers;  // остаток для последнего процесса
     offset = 0;

t1=MPI_Wtime(); // фиксировать время начала счета

     for (dest=1; dest<=numworkers; dest++)
     {
       if(dest <= extra)
         rows = averow + 1;
       else
         rows = averow;

       rows = (dest <= extra) ? averow+1 : averow;
       printf(“...sending %d rows to task %d\n”, rows, dest);
       MPI_Send(&offset, 1, MPI_INT, dest, FROM_MASTER, M_C_W);
       MPI_Send(&rows,  1, MPI_INT, dest, FROM_MASTER,  M_C_W);
       MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER,
                                                              M_C_W);
       MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, M_C_W);
       offset += rows;
     }
```

```
/* ожидание результатов от WORKER-процессов */
     for (source=1; source<=numworkers; i++) // цикл по WORKER-процессам
     {
       MPI_Recv(&offset, 1, MPI_INT, source, FROM_WORKER, M_C_W, &status);
       MPI_Recv(&rows,  1, MPI_INT, source, FROM_WORKER,  M_C_W, &status);
       MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, FROM_WORKER,
                                                                  M_C_W, &status);
     }

t2=MPI_Wtime();
printf ("Multiply time= %.3lf sec\n\n", t2-t1);

     printf("Here is the result matrix:\n");
     for (i=0; i<NRA; i++)
     {
       printf("\n");
       for (j=0; j<NCB; j++)
         printf("%6.2f  ", c[i][j]);
      }
     printf ("\n");
}

/***************************** WORKER-процессы ***********************************/
  if (taskid > MASTER) // если это не MASTER-процесс
  {
     MPI_Recv(&offset, 1, MPI_INT, MASTER, FROM_MASTER, M_C_W, &status);
     MPI_Recv(&rows,  1, MPI_INT, MASTER, FROM_MASTER,  M_C_W, &status);
     MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, FROM_MASTER,
                                                  M_C_W, &status);
     MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, FROM_MASTER,
                                                  M_C_W, &status);

     for (k=0; k<NCB; k++)
       for (i=0; i<rows; i++)
       {
         c[i][k] = 0.0;
         for (j=0; j<NCA; j++)
           c[i][k] += a[i][j] * b[j][k];
       }

     MPI_Send(&a_offset, 1, MPI_INT, MASTER, FROM_WORKER, M_C_W);
     MPI_Send(&a_rows,  1, MPI_INT, MASTER, FROM_WORKER,  M_C_W);
     MPI_Send(&c, a_rows*NCB, MPI_DOUBLE, MASTER, FROM_WORKER,
                                                          M_C_W);
  }
  MPI_Finalize();
} // конец программы MM_MPI_2.C
```

Стро́ки матрицы [A] распределяются между numworkers процессов по rows штук – каждому процессу dest=1÷numworkers посредством вызова функции MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) пересылаются rows строк (rows=averow, если NRA делится

на numworkers нацело и rows=averow+1 в противном случае, offset – номер первой пересылаемой строки). Матрица [B] пересылается посредством MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, FROM_MASTER, MPI_COMM_WORLD) всем numworkers процессам целиком.

После получения процессами данных выполняется умножение посредством внешнего цикла по *k=1÷NCA* (т.е. по столбцам [A] и строкам [B] и внутреннего цикла по *i=1÷rows* (т.е. переданной части строк [A] и [C]):

```
for (k=0; k<NCB; k++)
 for (i=0; i<rows; i++) // …на узле присутствует только rows строк матрицы [A]
 {
  c[i][k] = 0.0;
   for (j=0; j<NCA; j++)
     c[i][k] += a[i][j] * b[j][k];
 }
```

В конце работы все numworkers процессов возвращают rows*NCB строк [C] главному процессу посредством выполнения MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, FROM_MASTER, MPI_COMM_WORLD); причем предварительно номер первой строки offset передается посредством MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD), а число строк rows – MPI_Send(&rows,1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD). MASTER-процесс принимает эти данные вызовом MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, FROM_WORKER, MPI_COMM_WORLD, &status).

Ниже приводится начало выдачи программы при NRA=1000 и числе вычислительных узлов 12 (видно, что в случае невозможности деления NRA на numworkers нацело равный extra=NRA%numworkers остаток распределяется равномерно по первым 4 вычислительным узлам):

```
Number of worker tasks = 12
task ID = 5
task ID = 11
task ID = 10
task ID = 1
task ID = 0
task ID = 6
task ID = 7
task ID = 3
task ID = 2
task ID = 9
task ID = 8
task ID = 12
task ID = 4
...sending 84 rows to task 1
...sending 84 rows to task 2
...sending 84 rows to task 3
...sending 84 rows to task 4
...sending 83 rows to task 5
```

```
...sending 83 rows to task 6
...sending 83 rows to task 7
...sending 83 rows to task 8
...sending 83 rows to task 9
...sending 83 rows to task 10
...sending 83 rows to task 11
...sending 83 rows to task 12
```

На практике процедура умножения матриц является всего лишь частью разрабатываемой проблемно-ориентированной системы; при этом исходные матрицы [A] и [B] вычисляются на предварительных стадиях расчета и необязательно присутствуют в полном виде в ОП главного процесса; программист должен разработать интерфейсные процедуры, позволяющие заполнять исходные матрицы непосредственно в ОП вычислительных узлов (и соответственно получать значения коэффициентов матрицы-произведения).

5.3 Необходимое оборудование

Для проведения работы необходим вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

5.4 Порядок проведения работы

Студент подготавливает исходные тексты MPI-программ, компилирует их в исполни́мое приложение, запускает на счет, проводит анализ полученных данных по заданию преподавателя.

В качестве параллельной программы умножения матриц используется аналогичная ранее приведенной MM_MPI.C, но с переменным размером квадратных перемножаемых матриц (в файле с именем mm_mpi_nnnn.c находится программа перемножения матриц размером nnnn×nnnn). При проведении эксперимента используются триады значений размера матриц N3, N2 и N1, причем N3=2×N2=4×N1; каждая триада запускается на общем числе процессоров от 1 до 24, при этом ‘считающих’ (WORKER) узлов P=1÷23. Т.о. всего опытов 23×3=69 (размер N1 выбирается таким образом, чтобы общее время счета не превышало 60÷90 мин).

Напр., для случая N1=800 скрипт выполнения эксперимента может быть таким:

```
#!/bin/bash

NP_MAX=24 # max count processors in system

for N in 0800 1600 3200 # cycles to matrice dimensions
do
  mpicc -o mm_mpi_$N -v mm_mpi/mm_mpi_$N.c  >& mm_mpi_$N.comp # compilation
  for NP in $(seq 2 $NP_MAX) # cycles to processors number
  do
```

```
  mpirun -np $NP mm_mpi_$N # parallel run
  done
done
```

Результаты отдельных опытов удобно заносить в табл.4.2; в каждой ячейке записывается два числа – время счета и коэффициент ускорения счета на P узлах $k_{y.p}$ (по отношению ко времени выполнения программы единственным ‘считающим’ узлом, т.е. $k_{y.p}=t_p/t_1$).

Таблица 4.2 — Данные опытов по определению зависимости производительности от числа вычислительных узлов и размерности обрабатываемых данных

Число ‘считающих’ узлов	*Время счета при размере матриц 600×600/ускорение, сек/раз*	*Время счета при размере матриц 1200×1200/ускорение, сек/раз*	*Время счета при размере матриц 2400×2400/ускорение, сек/раз*
1	(*принимается за 1,0* при вычислении ускорения для этого столбца)	…то же…	…то же…
2			
…			
22			
23			

5.5 Обработка результатов эксперимента

Полученные и внесённые в табл.4.2 данные полезно представить графически в форме, аналогичной рис.5.2 (графики построить вручную на ‘миллиметровке’ или использовать возможности MS Excel).

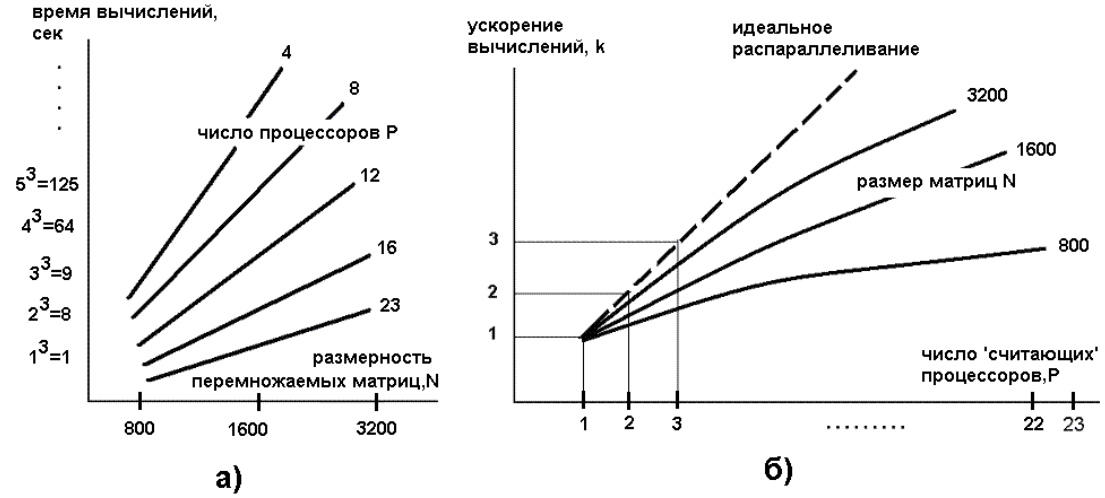

Рисунок 5.2 — а) - экспериментальная зависимость времени вычислений от размера обрабатываемых данных и числа вычислительных узлов, б) - зависимость ускорения вычислений от числа вычислительных узлов и размеров обрабатываемых данных

5.6 Вопросы для самопроверки

1. В каком случае время умножения матриц будет больше – в случае выполнения MM_SER или MM_MPI_0 при числе процессов P=2 (включая MASTER-процесс)?
2. Какими соображениями следует пользоваться при разработке стратегии распределения вычислений и блоков данных между процессами при распараллеливании алгоритмов?
3. Каким образом целесообразно распределить блоки исходных и вычисленной матриц в случае процедуры нахождения матричного произведения? При умножении/делении матрицы на скаляр? При транспонировании матрицы?
4. Оценить (в единицах умножения элементов двумерного массива ‘плавающих’ чисел двойной точности с последующим сложением их частичных сумм) размер *гранулы* (зерна, блока) параллелизма для случая параллельного умножения матриц ленточным способом. В каком случае *коэффициент ускорения вычислений* больше – при большей или меньшей размерности умножаемых матриц (предположение *проверить экспериментально*)?
5. Предложить и обосновать более эффективную (относительно вышеразобранных) стратегию распределения больших матриц по вычислительным узлам (обязательно учесть некратность числа строк и столбцов матрицы числу процессоров).
6. Известно, что ускорение вычислений растет не пропорционально числу вычислительных узлов (а медленнее), для теоретической оценки этого существует закон Амдаля $S \le \dfrac{1}{f + \left(\dfrac{1-f}{p}\right)}$, связывающий потенциальное ускорение вычислений S при распараллеливании на p процессорах с долей f операций, выполняемых *априори последовательно*. Используя возможности MS Excel по подбору параметра (меню Сервис → Подбор параметра…) и данные экспериментов (рис.5.2б), определите значение параметра f. Будет ли его величина одинакова для различных размеров умножаемых матриц? Почему? (провести мысленный эксперимент).

Список использованной литературы

1. Богачев К.Ю. Основы параллельного программирования. –М.: Бином, Лаборатория знаний, 2010. –344 с.
2. Барский А.Б. Параллельные информационные технологии. –М.: Бином, Лаборатория знаний, Интернет-университет информационных технологий, 2007. –503 с.
3. Гергель В.П. Теория и практика параллельных вычислений. –М.: Бином, Лаборатория знаний, Интернет-университет информационных технологий, 2007. –423 с.

Учебное издание.

Баканов Валерий Михайлович

Параллельные вычисления

Учебно-методическое пособие
по выполнению лабораторных работ

ЛР № _____ от __ _____марта 2010 г.

Подписано в печать хх.03.2010 г.
Формат 60 × 84. 1/16.
Объем 3,3 п.л. Тираж 100 экз. Заказ хх.

Московский государственный
университет приборостроения и информатики.
107996, Москва, ул.Стромынка, 20.