

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**Методические указания
по выполнению лабораторных работ по дисциплине
«Высокопроизводительные вычисления»**

для магистров направления
«Прикладная математика и информатика»

профиль
«Математическое моделирование»

Ставрополь, 2026

Высокопроизводительные вычисления: Методические указания по выполнению лабораторных работ. - Ставрополь: СКФУ, 2021. - 113 с.

Пособие подготовлено в соответствии с Федеральным государственным образовательным стандартом высшего образования. В учебном пособии изложены методические рекомендации по выполнению 12 лабораторных работ.

Пособие предназначено для изучения дисциплины «Высокопроизводительные вычисления» для магистров направления подготовки «Прикладная математика и информатика».

ОГЛАВЛЕНИЕ

Введение.....	4
1 Анализ параллельных вычислений	6
2 Расчет коммуникационной трудоемкости параллельных алгоритмов.....	13
3 Принципы разработки параллельных алгоритмов	28
4 Среда разработки приложений Microsoft Visual Studio	39
5 Создание проекта параллельного программирования	54
6 Программирование на MPI	65
7 Параллельные методы умножения матрицы на вектор	69
8 Программная реализация параллельных методы матричного умножения.....	75
9 Программная реализация параллельных методов решения систем линейных уравнений.....	83
10 Программная реализация параллельных методов сортировки	89
11 Программная реализация параллельных методов на графах	94
12 Программная реализация параллельных методов решения дифференциальных уравнений в частных производных	100
Литература	113

ВВЕДЕНИЕ

Потребность решения сложных прикладных задач с большим объемом вычислений и принципиальная ограниченность максимального быстродействия "классических" - по схеме фон Неймана - ЭВМ привели к появлению многопроцессорных вычислительных систем. Использование таких средств вычислительной техники позволяет существенно увеличивать производительность ЭВМ при любом существующем уровне развития компьютерного оборудования. При этом, однако, необходимо "параллельное" обобщение традиционной - последовательной - технологии решения задач на ЭВМ. Так, численные методы в случае использования суперкомпьютеров должны проектироваться как системы параллельных и взаимодействующих между собой процессов, допускающих исполнение на независимых процессорах. Применяемые алгоритмические языки и системное программное обеспечение должны обеспечивать создание параллельных программ, организовывать синхронизацию и взаимоисключение асинхронных процессов и т.п.

Цель курса состоит в изложении математических моделей и методов высокопроизводительных вычислений для суперкомпьютерных вычислительных средств с использованием технологий параллельного программирования.

Задачи курса:

- знакомство с основными областями применения, архитектурой, принципами построения и классификацией суперкомпьютерных вычислительных средств;
- анализ и изучение подходов к формализации высокопроизводительных вычислений;
- изучение принципов разработки параллельных алгоритмов и технологий параллельного программирования;
- обобщение основ численных методов решения прикладных математических задач с использованием высокопроизводительной вычислительной техники.

Освоение дисциплины «Высокопроизводительные вычисления» позволит будущему магистру по направлению подготовки «Прикладная математика и информатика» полноценно осуществлять свою профессиональную деятельность.

Аппаратура и материалы

Практикум выполняется в штатном компьютерном классе, оборудованном IBM-совместимыми персональными компьютерами под управлением операционной системы Windows. Необходимым является наличие инструментальной среды Visual Studio и высокопроизводительного вычислительного кластера.

Указания по технике безопасности

Указания по технике безопасности в штатном компьютерном классе разработаны и утверждены администрацией и доводятся каждому студенту

под роспись. Дополнительных требований нет.

1 АНАЛИЗ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Цель: Углубить и закрепить знания, полученные на лекциях по анализу параллельных вычислений.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Для иллюстрации введенных понятий на лекции рассмотрим учебный пример решения задачи вычисления частных сумм для последовательности числовых значений. Кроме того, данные показатели будут использоваться для характеристики эффективности всех рассматриваемых в лекциях параллельных алгоритмов при решении ряда типовых задач вычислительной математики.

1. Вычисление частных сумм последовательности числовых значений

Рассмотрим для демонстрации ряда проблем, возникающих при разработке параллельных методов вычислений, сравнительно простую задачу нахождения частных сумм последовательности числовых значений

$$S_k = \sum_{i=1}^k x_i, \quad 1 \leq k \leq n,$$

где n есть количество суммируемых значений.

Изучение возможных параллельных методов решения данной задачи начнем с еще более простого варианта ее постановки – с задачи вычисления общей суммы имеющегося набора значений (в таком виде задача суммирования является частным случаем общей задачи редукции)

$$S = \sum_{i=1}^n x_i.$$

1.1 Последовательный алгоритм суммирования

Традиционный алгоритм для решения этой задачи состоит в последовательном суммировании элементов числового набора

$$\begin{aligned} S &= 0, \\ S &= S + x_1, \dots \end{aligned}$$

Вычислительная схема данного алгоритма может быть представлена следующим образом (рисунок 1):

$$G_1 = (V_1, R_1),$$

где $V_1 = \{v_{01}, \dots, v_{0n}, v_{11}, \dots, v_{1n}\}$ есть множество операций (вершины $v_{01}, \dots, v_{0n}$ обозначают операции ввода, каждая вершина v_{1i} , $1 \leq i \leq n$, соответствует прибавлению значения x_i к накапливаемой сумме S), а

$$R_1 = \{(v_{0i}, v_{1i}), (v_{1i}, v_{1i+1}), 1 \leq i \leq n-1\}$$

есть множество дуг, определяющих информационные зависимости операций.

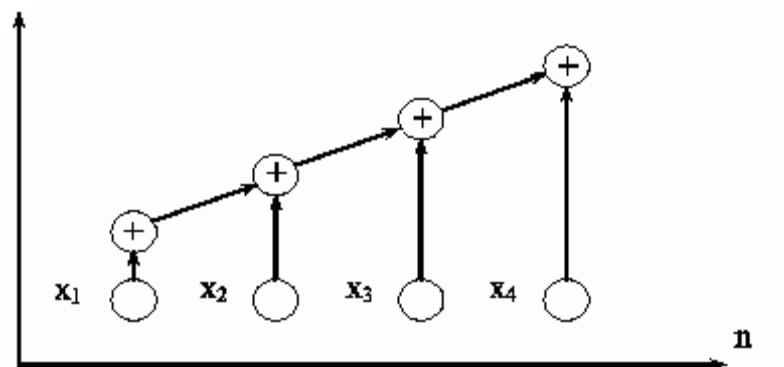


Рисунок 1 - Последовательная вычислительная схема алгоритма суммирования

Как можно заметить, данный "стандартный" алгоритм суммирования допускает только строго последовательное исполнение и не может быть распараллелен.

1.2 Каскадная схема суммирования

Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования состоит в следующем (рисунок 2):

- на первой итерации каскадной схемы все исходные данные разбиваются на пары, и для каждой пары вычисляется сумма их значений;
- далее все полученные суммы также разбиваются на пары, и снова выполняется суммирование значений пар и т.д.

Данная вычислительная схема может быть определена как граф (пусть $n=2^k$) $G_2(V_2, R_2)$,

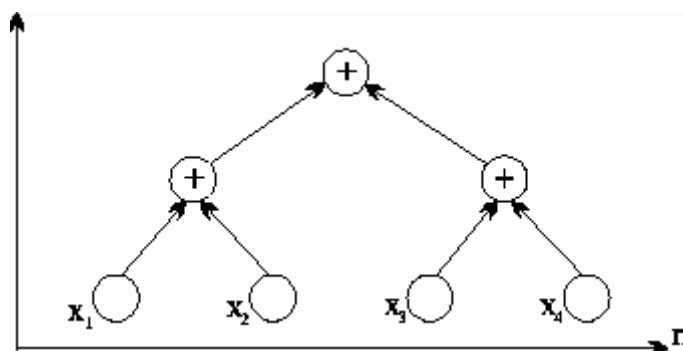


Рисунок 2 - Каскадная схема алгоритма суммирования

где $V_2 = \{(v_{i1}, \dots, v_{li}), 0 \leq i \leq k, 1 \leq l_i \leq 2^{i-1}n\}$ есть вершины графа ($v_{01}, \dots, v_{0n}$) - операции ввода, ($v_{11}, \dots, v_{1n/2}$) - операции суммирования первой итерации и т.д.), а множество дуг графа определяется соотношениями:

$$R_2 = \{(v_{i-1, 2j-1} v_{ij}), (v_{i-1, 2j} v_{ij}), 1 \leq i \leq k, 1 \leq j \leq 2^{i-1}n\}.$$

Как нетрудно оценить, количество итераций каскадной схемы оказывается равным величине

$$k = \log_2 n,$$

а общее количество операций суммирования

$$K_{\text{посл}} = n/2 + n/4 + \dots + 1 = n - 1$$

совпадает с количеством операций последовательного варианта алгоритма суммирования. При параллельном исполнении отдельных итераций каскадной схемы общее количество параллельных операций суммирования является равным

$$K_{\text{пар}} = \log_2 n.$$

Поскольку считается, что время выполнения любых вычислительных операций является одинаковым и единичным, то $T_1 = K_{\text{посл}}$, $T_p = K_{\text{пар}}$, поэтому показатели ускорения и эффективности каскадной схемы алгоритма суммирования можно оценить как

$$S_p = T_1 / T_p = (n - 1) / \log_2 n,$$

$$E_p = T_1 / p T_p = (n - 1) / (p \log_2 n) = (n - 1) / ((n/2) \log_2 n),$$

где $p = n/2$ есть необходимое для выполнения каскадной схемы количество процессоров.

Анализируя полученные характеристики, можно отметить, что время параллельного выполнения каскадной схемы совпадает с оценкой для параллельного выполнения. Однако при этом эффективность использования процессоров уменьшается при увеличении количества суммируемых значений

$$\lim E_p \rightarrow 0 \text{ при } n \rightarrow \infty$$

1.3 Модифицированная каскадная схема

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы. Для упрощения построения оценок можно предположить $n = 2^k$, $k = 2^s$. Тогда в новом варианте каскадной схемы все вычисления производятся в два последовательно выполняемых этапа суммирования (рисунок 3):

- на первом этапе вычислений все суммируемые значения подразделяются на $(n/\log_2 n)$ групп, в каждой из которых содержится $\log_2 n$ элементов; далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования; вычисления в каждой группе могут выполняться независимо друг от друга (т.е. параллельно – для этого необходимо наличие не менее $(n/\log_2 n)$ процессоров);
- на втором этапе для полученных $(n/\log_2 n)$ сумм отдельных групп применяется обычная каскадная схема.

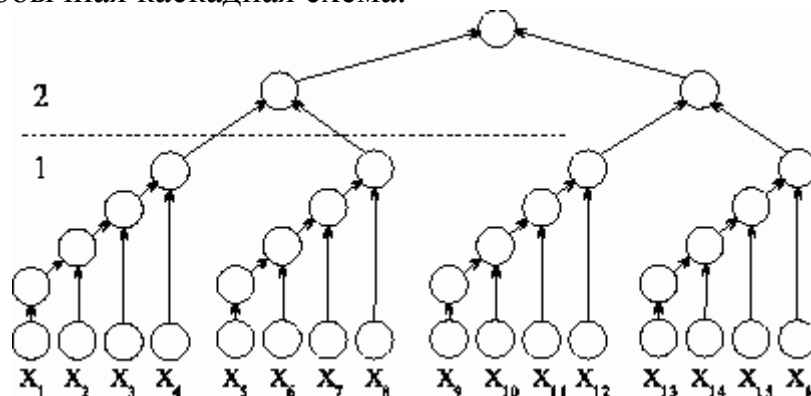


Рисунок 3 - Модифицированная каскадная схема суммирования

Тогда для выполнения первого этапа требуется $\log_2 n$ параллельных операций при использовании $p_1 = (n/\log_2 n)$ процессоров. Для выполнения второго этапа необходимо

$$\log_2(n/\log_2 n) \leq \log_2 n$$

параллельных операций для $p_2 = (n/\log_2 n)/2$ процессоров. Как результат, данный способ суммирования характеризуется следующими показателями:

$$T_p = 2\log_2 n, \quad p = (n/\log_2 n).$$

С учетом полученных оценок показатели ускорения и эффективности модифицированной каскадной схемы определяются соотношениями:

$$S_p = T_1/T_p = (n-1)/2\log_2 n,$$

$$E_p = T_1/pT_p = (n-1)/(2(n/\log_2 n)\log_2 n) = (n-1)/2n.$$

Сравнивая данные оценки с показателями обычной каскадной схемы, можно отметить, что ускорение для предложенного параллельного алгоритма уменьшилось в 2 раза, однако для эффективности нового метода суммирования можно получить асимптотически ненулевую оценку снизу

$$E_p = (n-1)/2n \geq 0,25; \quad \lim_{n \rightarrow \infty} E_p \rightarrow 0,5 \text{ при } n \rightarrow \infty.$$

Можно отметить также, что данные значения показателей достигаются при количестве процессоров. Кроме того, необходимо подчеркнуть, что, в отличие от обычной каскадной схемы, модифицированный каскадный алгоритм является стоимостно-оптимальным, поскольку стоимость вычислений в этом случае

$$C_p = pT_p = (n/\log_2 n)(2\log_2 n)$$

является пропорциональной времени выполнения последовательного алгоритма.

2 Вычисление всех частных сумм

Вернемся к исходной задаче вычисления всех частных сумм последовательности значений и проведем анализ возможных способов последовательной и параллельной организации вычислений. Вычисление всех частных сумм на скалярном компьютере может быть получено при помощи обычного последовательного алгоритма суммирования при том же количестве операций (!)

$$T_1 = n.$$

При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемым результатам; достижение эффективного распараллеливания требует привлечения новых подходов (может быть, даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения задач. Так, для рассматриваемой задачи нахождения всех частных сумм алгоритм, обеспечивающий получение результатов за $\log_2 n$ параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (рисунок 4):

- перед началом вычислений создается копия s вектора суммируемых значений ($S = x$);
- далее на каждой итерации суммирования i , $1 \leq i \leq \log_2 n$, формируется вспомогательный вектор q путем сдвига вправо вектора S на 2^{i-1} пози-

ций (освобождающиеся при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов S и Q.

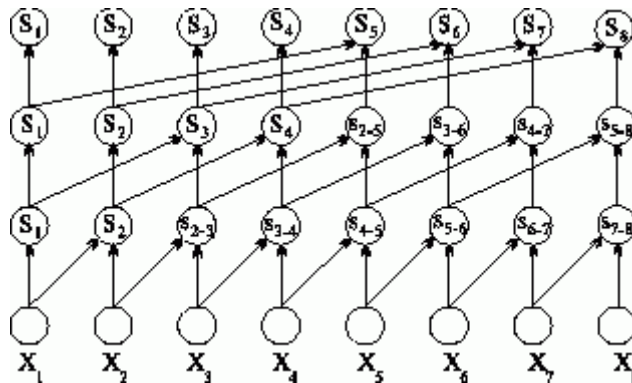


Рисунок 4 - Схема параллельного алгоритма вычисления всех частных сумм

(величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности)

Всего параллельный алгоритм выполняется за $\log_2 n$ параллельных операций сложения. На каждой итерации алгоритма параллельно выполняются p скалярных операций сложения и, таким образом, общее количество скалярных операций определяется величиной

$$K_{\text{пар}} = n \log_2 n$$

(параллельный алгоритм содержит большее (!) количество операций по сравнению с последовательным способом суммирования). Необходимое количество процессоров определяется количеством суммируемых значений ($p=n$).

С учетом полученных соотношений показатели ускорения и эффективности параллельного алгоритма вычисления всех частных сумм оцениваются следующим образом:

$$S_p = T_1 / T_p = n / \log_2 n,$$

$$E_p = T_1 / p T_p = n / (p \log_2 n) = n / (n \log_2 n) = 1 / \log_2 n.$$

Как следует из построенных оценок, эффективность алгоритма также уменьшается при увеличении числа суммируемых значений, и при необходимости повышения величины этого показателя может оказаться полезной модификация алгоритма, как и в случае с обычной каскадной схемой.

ЗАДАНИЕ

1. Разработайте модель и выполните оценку показателей ускорения и эффективности параллельных вычислений:

1.1. для задачи скалярного произведения двух векторов

$$y = \sum_{i=1}^N a_i b_i;$$

1.2. для задачи поиска максимального значения для заданного набора числовых данных

- 1.3. для задачи поиска минимального значения для заданного набора числовых данных

$$y_{\min} = \min_{1 \leq i \leq N} a_i, \quad y_{\max} = \max_{1 \leq i \leq N} a_i;$$

- 1.4. для задачи нахождения среднего значения для заданного набора числовых данных

$$y = \frac{1}{N} \sum_{i=1}^N a_i.$$

- 1.5. для задачи сортировки вектора значений по убыванию.
1.6. для задачи нахождения среднеквадратического отклонения выборки случайных значений.
1.7. для задачи нахождения дисперсии выборки случайных значений.
1.8. для задачи сортировки вектора значений по возрастанию.
1.9. Для задачи матричного умножения
1.10. Для задачи умножения вектора на матрицу.
1.11. Для задачи сложения двух векторов.
1.12. Для задачи сложения двух матриц.
1.13. Для задачи нахождения обратной матрицы.
1.14. Для задачи нахождения определителя матрицы.
1.15. Для задачи нахождения средневзвешенного значения вектора значений.
1.16. Для задачи вычисления ранга матрицы.
2. Выполните в соответствии с законом Амдаля оценку максимально достижимого ускорения для задач п. 1.
3. Выполните оценку ускорения масштабирования для задач п.1.
4. Выполните построение функций изоэффективности для задач п.1.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как определяется модель "операции — операнды"?
2. Как определяется расписание для распределения вычислений между процессорами?
3. Как определяется время выполнения параллельного алгоритма?
4. Какое расписание является оптимальным?
5. Как определить минимально возможное время решения задачи?
6. Что понимается под паракомпьютером и для чего может оказаться полезным данное понятие?
7. Какие оценки следует использовать в качестве характеристики времени последовательного решения задачи?
8. Как определить минимально возможное время параллельного решения задачи по графу "операнды – операции"?
9. Какие зависимости могут быть получены для времени параллельного решения задачи при увеличении или уменьшении числа используемых процессоров?

10. При каком числе процессоров могут быть получены времена выполнения параллельного алгоритма, сопоставимые по порядку с оценками минимально возможного времени решения задачи?
11. Как определяются понятия ускорения и эффективности?
12. Возможно ли достижение сверхлинейного ускорения?
13. В чем состоит противоречивость показателей ускорения и эффективности?
14. Как определяется понятие стоимости вычислений?
15. В чем состоит понятие стоимостно-оптимального алгоритма?
16. В чем заключается проблема распараллеливания последовательного алгоритма суммирования числовых значений?
17. В чем состоит каскадная схема суммирования? С какой целью рассматривается модифицированный вариант данной схемы?
18. В чем состоит различие показателей ускорения и эффективности для рассматриваемых вариантов каскадной схемы суммирования?
19. В чем состоит параллельный алгоритм вычисления всех частных сумм последовательности числовых значений?
20. Как формулируется закон Амдаля? Какой аспект параллельных вычислений позволяет учесть данный закон?
21. Какие предположения используются для обоснования закона Густавсона – Барсиса?
22. Как определяется функция изоэффективности?
23. Какой алгоритм является масштабируемым? Приведите примеры методов с разным уровнем масштабируемости.

2 РАСЧЕТ КОММУНИКАЦИОННОЙ ТРУДОЕМКОСТИ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ

Цель: Углубить и закрепить знания, полученные на лекциях по коммуникационной трудоемкости параллельных алгоритмов.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Алгоритмы маршрутизации определяют путь передачи данных от процессора – источника сообщения до процессора, к которому сообщение должно быть доставлено. Среди возможных способов решения данной задачи различают:

- оптимальные, определяющие всегда наикратчайшие пути передачи данных, и неоптимальные алгоритмы маршрутизации;
- детерминированные и адаптивные методы выбора маршрутов (адаптивные алгоритмы определяют пути передачи данных в зависимости от существующей загрузки коммуникационных каналов).

К числу наиболее распространенных оптимальных алгоритмов относится класс методов покоординатной маршрутизации (dimension-ordered routing), в которых поиск путей передачи данных осуществляется поочередно для каждой размерности топологии сети коммуникации. Так, для двумерной решетки такой подход приводит к маршрутизации, при которой передача данных сначала выполняется по одному направлению (например, по горизонтали до достижения вертикали, на которой располагается процессор назначения), а затем данные передаются вдоль другого направления (данная схема известна под названием алгоритма ХУ-маршрутизации).

Для гиперкуба покоординатная схема маршрутизации может состоять, например, в циклической передаче данных процессору, определяемому первой различающейся битовой позицией в номерах процессоров — того, на котором сообщение располагается в данный момент времени, и того, на который оно должно быть передано.

Методы передачи данных

Время передачи данных между процессорами определяет коммуникационную составляющую (communication latency) длительности выполнения параллельного алгоритма в многопроцессорной вычислительной системе. Основным набор параметров, описывающих время передачи данных, состоит из следующего ряда величин:

- время начальной подготовки (t_n) характеризует длительность подготовки сообщения для передачи, поиска маршрута в сети и т. п.;
- время передачи служебных данных (t_c) между двумя соседними процессорами (т.е. для процессоров, между которыми имеется физический канал передачи данных). К служебным данным может относиться заголовок сообщения, блок данных для обнаружения ошибок передачи и т. п.;

- время передачи одного слова данных по одному каналу передачи данных (t_k). Длительность подобной передачи определяется полосой пропускания коммуникационных каналов в сети.

К числу наиболее распространенных методов передачи данных относятся два основных способа коммуникации). Первый из них ориентирован на передачу сообщений (метод передачи сообщений или МПС) как неделимых (атомарных) блоков информации (store-and-forward routing или SFR). При таком подходе процессор, содержащий сообщение для передачи, готовит весь объем данных для передачи, определяет процессор, которому следует направить данные, и запускает операцию пересылки данных. Процессор, которому направлено сообщение, в первую очередь осуществляет прием полностью всех пересылаемых данных и только затем приступает к пересылке принятого сообщения далее по маршруту. Время пересылки данных $t_{пд}$ для метода передачи сообщения размером m байт по маршруту длиной l определяется выражением:

$$t_{пд} = t_k + (mt_k + t_c)l. \quad (1)$$

При достаточно длинных сообщениях временем передачи служебных данных можно пренебречь и выражение для времени передачи данных может быть записано в более простом виде:

$$t_{пд} = t_k + mt_k l. \quad (2)$$

Второй способ коммуникации основывается на представлении пересылаемых сообщений в виде блоков информации меньшего размера – пакетов, в результате чего передача данных может быть сведена к передаче пакетов (метод передачи пакетов или МПП). При таком методе коммуникации (cut-through routing или CTR) принимающий процессор может осуществлять пересылку данных по дальнейшему маршруту непосредственно сразу после приема очередного пакета, не дожидаясь завершения приема данных всего сообщения. Время пересылки данных при использовании метода передачи пакетов определяется выражением:

$$t_{пд} = t_k + mt_k + t_c l. \quad (3)$$

Сравнивая полученные выражения, можно заметить, что в большинстве случаев метод передачи пакетов приводит к более быстрой пересылке данных; кроме того, данный подход снижает потребность в памяти для хранения пересылаемых данных при организации приема-передачи сообщений, а для передачи пакетов могут использоваться одновременно разные коммуникационные каналы. С другой стороны, реализация пакетного метода требует разработки более сложного аппаратного и программного обеспечения сети, может увеличить накладные расходы (время подготовки и время передачи служебных данных). Кроме того, при передаче пакетов возможно возникновение конфликтных ситуаций (дедлоков).

Анализ трудоемкости основных операций передачи данных

При всем разнообразии выполняемых операций передачи данных при параллельных способах решения сложных научно-технических задач, опре-

деленные процедуры взаимодействия процессоров сети могут быть отнесены к числу основных коммуникационных действий, либо наиболее широко распространенных в практике параллельных вычислений, либо тех, к которым могут быть сведены многие другие процессы приема-передачи сообщений. Важно отметить также, что в рамках подобного базового набора для большинства операций коммуникации существуют процедуры, обратные по действию исходным операциям (так, например, операции передачи данных от одного процессора всем имеющимся процессорам сети соответствует операция приема в одном процессоре сообщений от всех остальных процессоров). Как результат, рассмотрение коммуникационных процедур целесообразно выполнять попарно, поскольку во многих случаях алгоритмы выполнения прямой и обратной операций могут быть получены исходя из одинаковых предпосылок.

Рассмотрение основных операций передачи данных в этой лекции будет осуществляться на примере таких топологий сети, как кольцо, двумерная решетка и гиперкуб. Для двумерной решетки будет предполагаться также, что между граничными процессорами в строках и столбцах решетки имеются каналы передачи данных (т.е. топология сети представляет собой тор). Как и ранее, величина m будет означать размер сообщения в словах, значение p определяет количество процессоров в сети, а переменная N задает размерность топологии гиперкуба.

Передача данных между двумя процессорами сети

Трудоемкость данной коммуникационной операции может быть получена путем подстановки длины максимального пути (диаметра сети) в выражения для времени передачи данных при разных методах коммуникации – см. табл. 1.

Таблица 1. Время передачи данных между двумя процессорами

Топология	Передача сообщений	Передача пакетов
Кольцо	$t_n + mt_k \lfloor p/2 \rfloor$	$t_n + mt_k + t_c \lfloor p/2 \rfloor$
Решетка-тор	$t_n + 2mt_k \lfloor \sqrt{p}/2 \rfloor$	$t_n + mt_k + 2t_c \lfloor \sqrt{p}/2 \rfloor$
Гиперкуб	$t_n + mt_k \log_2 p$	$t_n + mt_k + t_c \log_2 p$

Передача данных от одного процессора всем остальным процессорам сети

Операция передачи данных (одного и того же сообщения) от одного процессора всем остальным процессорам сети (one-to-all broadcast или single-node broadcast) является одним из наиболее часто выполняемых коммуникационных действий. Двойственная ей операция – прием на одном процессоре сообщений от всех остальных процессоров сети (single-node accumulation). Подобные операции используются, в частности, при реализации матрично-векторного умножения, решении систем линейных уравнений методом Гаусса, решении задачи поиска кратчайших путей и др.

Простейший способ реализации операции рассылки состоит в ее выполнении как последовательности попарных взаимодействий процессоров сети. Однако при таком подходе большая часть пересылок является избыточной и возможно применение более эффективных алгоритмов коммуникации. Изложение материала будет проводиться сначала для метода передачи сообщений, затем – для пакетного способа передачи данных.

Передача сообщений. Для **кольцевой топологии** процессор – источник рассылки может инициировать передачу данных сразу двум своим соседям, которые, в свою очередь, приняв сообщение, организуют пересылку далее по кольцу. Трудоемкость выполнения операции рассылки в этом случае будет определяться соотношением:

$$t_{n\partial} = (t_n + mt_k) \lceil p/2 \rceil. \quad (4)$$

Для топологии типа **решетка-тор** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети. Так, рассылка может быть выполнена в виде двухэтапной процедуры. На первом этапе организуется передача сообщения всем процессорам сети, располагающимся на той же горизонтали решетки, что и процессор – инициатор передачи. На втором этапе процессоры, получившие копию данных на первом этапе, рассылают сообщения по своим соответствующим вертикалям. Оценка длительности операции рассылки в соответствии с описанным алгоритмом определяется соотношением:

$$t_{n\partial} = 2(t_n + mt_k) \lceil \sqrt{p}/2 \rceil. \quad (5)$$

Для гиперкуба рассылка может быть выполнена в ходе N-этапной процедуры передачи данных. На первом этапе процессор-источник сообщения передает данные одному из своих соседей (например, по первой размерности) – в результате после первого этапа есть два процессора, имеющих копию пересылаемых данных (данный результат можно интерпретировать также как разбиение исходного гиперкуба на два таких одинаковых по размеру гиперкуба размерности N-1, что каждый из них имеет копию исходного сообщения). На втором этапе два процессора, задействованные на первом этапе, пересылают сообщение своим соседям по второй размерности и т.д. В результате такой рассылки время операции оценивается при помощи выражения:

$$t_{n\partial} = (t_n + mt_k) \log_2 p. \quad (6)$$

Сравнивая полученные выражения для длительности выполнения операции рассылки, можно отметить, что наилучшие показатели имеет топология типа гиперкуб; более того, можно показать, что данный результат является наилучшим для выбранного способа коммуникации с помощью передачи сообщений.

Передача пакетов. Для топологии типа **кольцо** алгоритм рассылки может быть получен путем логического представления кольцевой структуры сети в виде гиперкуба. В результате на этапе рассылки процессор – источник сообщения передает данные процессору, находящемуся на расстоянии $p/2$ от исходного процессора. Далее, на втором этапе оба процессора, уже имеющие

рассылаемые данные после первого этапа, передают сообщения процессорам, находящимся на расстоянии $p/4$, и т.д. Трудоемкость выполнения операции рассылки при таком методе передачи данных определяется соотношением:

$$t_{nd} = \sum_{i=1}^{\log_2 p} (t_n + mt_k + t_c p / 2^i) = (t_n + mt_k) \log_2 p + t_c (p - 1) \quad (7)$$

(как и ранее, при достаточно больших сообщениях временем передачи служебных данных можно пренебречь).

Для топологии типа **решетка-тор** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети, в соответствии с тем же способом обобщения, что и в случае использования метода передачи сообщений. Получаемый в результате такого обобщения алгоритм рассылки характеризуется следующим соотношением для оценки времени выполнения:

$$t_{nd} = (t_n + mt_k) \log_2 p + 2t_c (\sqrt{p} - 1). \quad (8)$$

Для **гиперкуба** алгоритм рассылки (и, соответственно, временные оценки длительности выполнения) при передаче пакетов не отличается от варианта для метода передачи сообщений.

Передача данных от всех процессоров всем процессорам сети

Операция передачи данных от всех процессоров всем процессорам сети (all-to-all broadcast или multinode broadcast) является естественным обобщением одиночной операции рассылки, двойственная ей операция – прием сообщений на каждом процессоре от всех процессоров сети (multinode accumulation). Подобные операции широко используются, например, при реализации матричных вычислений.

Возможный способ реализации операции множественной рассылки состоит в выполнении соответствующего набора операций одиночной рассылки. Однако такой подход не является оптимальным для многих топологий сети, поскольку часть необходимых операций одиночной рассылки потенциально может быть выполнена параллельно.

Передача сообщений. Для **кольцевой топологии** каждый процессор может инициировать рассылку своего сообщения одновременно (в каком-либо выбранном направлении по кольцу). В любой момент каждый процессор выполняет прием и передачу данных, завершение операции множественной рассылки произойдет через $p-1$ цикл передачи данных. Длительность выполнения операции рассылки оценивается соотношением:

$$t_{nd} = (t_n + mt_k)(p - 1). \quad (9)$$

Для топологии типа **решетка-тор** множественная рассылка сообщений может быть выполнена при помощи алгоритма, получаемого обобщением способа передачи данных для кольцевой структуры сети. Схема обобщения состоит в следующем. На первом этапе организуется передача сообщений отдельно по всем процессорам сети, располагающимся на одних и тех же

горизонтальных решетки (в результате на каждом процессоре одной и той же горизонтали формируются укрупненные сообщения размера $m\sqrt{p}$, объединяющие все сообщения горизонтали). Время выполнения этапа:

$$t'_{n\partial} = (t_n + mt_k)(\sqrt{p} - 1).$$

На втором этапе рассылка данных выполняется по процессорам сети, образующим вертикали решетки. Длительность этого этапа:

$$t''_{n\partial} = (t_n + m\sqrt{p}t_k)(\sqrt{p} - 1).$$

Общая длительность операции рассылки определяется соотношением:

$$t_{n\partial} = 2t_n(\sqrt{p} - 1) + mt_k(p - 1). \quad (10)$$

Для гиперкуба алгоритм множественной рассылки сообщений может быть получен путем обобщения ранее описанного способа передачи данных для топологии типа решетки на размерность гиперкуба N . В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе i , $1 \leq i \leq N$, выполнения алгоритма функционируют все процессоры сети, которые обмениваются своими данными со своими соседями по i -ой размерности и формируют объединенные сообщения. Время операции рассылки может быть получено при помощи выражения:

$$t_{n\partial} = \sum_{i=1}^{\log_2 p} (t_n + 2^{i-1}mt_k) = t_n \log_2 p + mt_k(p - 1). \quad (11)$$

Передача пакетов. Применение более эффективного для **кольцевой структуры** и топологии типа **решетка-тор** метода передачи данных не приводит к какому-либо улучшению времени выполнения операции множественной рассылки, поскольку обобщение алгоритмов выполнения операции одиночной рассылки на случай множественной рассылки приводит к перегрузке каналов передачи данных (т.е. к существованию ситуаций, когда в один и тот же момент для передачи по одной и той же линии имеется несколько ожидающих пересылки пакетов данных). Перегрузка каналов приводит к задержкам при пересылках данных, что и не позволяет проявиться всем преимуществам метода передачи пакетов.

Широко распространенным примером операции множественной рассылки является задача редукции (reduction), которая определяется в общем виде как процедура выполнения той или иной обработки данных, получаемых на каждом процессоре в ходе множественной рассылки (в качестве примера такой задачи может быть рассмотрена проблема вычисления суммы значений, находящихся на разных процессорах, и рассылки полученной суммы по всем процессорам сети). Способы решения задачи редукции могут состоять в следующем:

- непосредственный подход заключается в выполнении операции множественной рассылки и последующей затем обработке данных на каждом процессоре в отдельности;
- более эффективный алгоритм может быть получен в результате применения операции одиночного приема данных на отдельном процессоре,

выполнения на этом процессоре действий по обработке данных и рассылки полученного результата обработки всем процессорам сети;

- наилучший же способ решения задачи редукции состоит в совмещении процедуры множественной рассылки и действий по обработке данных, когда каждый процессор сразу же после приема очередного сообщения реализует требуемую обработку полученных данных (например, выполняет сложение полученного значения с имеющейся на процессоре частичной суммой). Время решения задачи редукции при таком алгоритме реализации в случае, например, когда размер пересылаемых данных имеет единичную длину ($m=1$) и топология сети имеет структуру гиперкуба, определяется выражением:

$$t_{n\partial} = (t_n + t_k) \log_2 p. \quad (12)$$

Другим типовым примером использования операции множественной рассылки является задача нахождения частных сумм последовательности значений S_i

$$S_k = \sum_{i=1}^k x_i, \quad 1 \leq k \leq p \quad (13)$$

(будем предполагать, что количество значений совпадает с количеством процессоров, значение x_i располагается на i -м процессоре и результат S_k должен получаться на процессоре с номером k).

Алгоритм решения данной задачи также может быть получен при помощи конкретизации общего способа выполнения множественной операции рассылки, когда процессор выполняет суммирование полученного значения (но только в том случае, если процессор – отправитель значения имеет меньший номер, чем процессор-получатель).

Обобщенная передача данных от одного процессора всем остальным процессорам сети

Общий случай передачи данных от одного процессора всем остальным процессорам сети состоит в том, что все рассылаемые сообщения являются различными (one-to-all personalized communication или single-node scatter). Двойственная операция передачи для данного типа взаимодействия процессоров – обобщенный прием сообщений (single-node gather) на одном процессоре от всех остальных процессоров сети (отличие данной операции от ранее рассмотренной процедуры сборки данных на одном процессоре состоит в том, что обобщенная операция сборки не предполагает какого-либо взаимодействия сообщений (например, редукции) в процессе передачи данных).

Трудоемкость операции обобщенной рассылки сопоставима со сложностью выполнения процедуры множественной передачи данных. Процессор – инициатор рассылки посылает каждому процессору сети сообщение размера m , и, тем самым, нижняя оценка длительности выполнения операции характеризуется величиной $mt_k(p-1)$.

Проведем более подробный анализ трудоемкости обобщенной рассылки для случая топологии типа **гиперкуб**. Возможный способ выполнения операции состоит в следующем. Процессор – инициатор рассылки передает половину своих сообщений одному из своих соседей (например, по первой размерности) – в результате исходный гиперкуб становится разделенным на два гиперкуба половинного размера, в каждом из которых содержится ровно половина исходных данных. Далее, действия по рассылке сообщений могут быть повторены, и общее количество повторений определяется исходной размерностью гиперкуба. Длительность операции обобщенной рассылки может быть охарактеризована соотношением:

$$t_{nd} = t_n \log_2 p + m t_k (p - 1) \quad (14)$$

(как и отмечалась выше, трудоемкость операции совпадает с длительностью выполнения процедуры множественной рассылки).

Обобщенная передача данных от всех процессоров всем процессорам сети

Обобщенная передача данных от всех процессоров всем процессорам сети (total exchange) представляет собой наиболее общий случай коммуникационных действий. Необходимость выполнения подобных операций возникает в параллельных алгоритмах быстрого преобразования Фурье, транспонирования матриц и др.

Выполним краткую характеристику возможных способов выполнения обобщенной множественной рассылки для разных методов передачи данных.

Передача сообщений. Общая схема алгоритма для **кольцевой топологии** состоит в следующем. Каждый процессор производит передачу всех своих исходных сообщений своему соседу (в каком-либо выбранном направлении по кольцу). Далее процессоры осуществляют прием направленных к ним данных, затем среди принятой информации выбирают свои сообщения, после чего выполняют дальнейшую рассылку оставшейся части данных. Длительность выполнения подобного набора передач данных оценивается при помощи выражения:

$$t_{nd} = (t_n + \frac{1}{2} m p t_k) (p - 1). \quad (15)$$

Способ получения алгоритма рассылки данных для топологии типа **решетка-тор** является тем же самым, что и в случае рассмотрения других коммуникационных операций. На первом этапе организуется передача сообщений отдельно по всем процессорам сети, располагающимся на одних и тех же горизонталях решетки (каждому процессору по горизонтали передаются только те исходные сообщения, что должны быть направлены процессорам соответствующей вертикали решетки). После завершения этапа на каждом процессоре собираются p сообщений, предназначенных для рассылки по одной из вертикалей решетки. На втором этапе рассылка данных вы-

полняется по процессорам сети, образующим вертикали решетки. Общая длительность всех операций рассылок определяется соотношением:

$$t_{n\partial} = (2t_n + mpt_{\kappa})(\sqrt{p} - 1). \quad (16)$$

Для **гиперкуба** алгоритм обобщенной множественной рассылки сообщений может быть получен путем обобщения способа выполнения операции для топологии типа решетка на размерность гиперкуба N . В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе i , $1 \leq i \leq N$, выполнения алгоритма функционируют все процессоры сети, которые обмениваются своими данными со своими соседями по i -й размерности и формируют объединенные сообщения. При организации взаимодействия двух соседей канал связи между ними рассматривается как связующий элемент двух равных по размеру подгиперкубов исходного гиперкуба, и каждый процессор пары посылает другому процессору только те сообщения, что предназначены для процессоров соседнего подгиперкуба. Время операции рассылки может быть получено при помощи выражения:

$$t_{n\partial} = (t_n + \frac{1}{2}mpt_{\kappa}) \log_2 p \quad (17)$$

(кроме затрат на пересылку, каждый процессор выполняет $mpt_{\kappa} \log_2 p$ операций по сортировке своих сообщений перед обменом информацией со своими соседями).

Передача пакетов. Как и в случае множественной рассылки, применение метода передачи пакетов не приводит к улучшению временных характеристик для операции обобщенной множественной рассылки. Рассмотрим как пример более подробно выполнение данной коммуникационной операции для сети с топологией типа **гиперкуб**. В этом случае рассылка может быть выполнена за $p-1$ итерацию. На каждой итерации все процессоры разбиваются на взаимодействующие пары процессоров, причем это разбиение на пары может быть выполнено таким образом, чтобы передаваемые между разными парами сообщения не использовали одни и те же пути передачи данных. Как результат, общая длительность операции обобщенной рассылки может быть определена в соответствии с выражением:

$$t_{n\partial} = (t_n + mt_{\kappa})(p - 1) + \frac{1}{2}t_{cp} \log_2 p. \quad (18)$$

Циклический сдвиг

Частный случай обобщенной множественной рассылки есть процедура перестановки (permutation), представляющая собой операцию перераспределения информации между процессорами сети, в которой каждый процессор передает сообщение определенному неким способом другому процессору сети. Конкретный вариант перестановки – циклический q -сдвиг (circular q -shift), при котором каждый процессор i , $1 \leq i \leq N$, передает данные процессору с номером $(i + q) \bmod p$. Подобная операция сдвига используется, например, при организации матричных вычислений.

Поскольку выполнение циклического сдвига для кольцевой топологии может быть обеспечено при помощи простых алгоритмов передачи данных, рассмотрим возможные способы выполнения данной коммуникационной операции только для топологий решетка-тор и гиперкуб при разных методах передачи данных.

Передача сообщений. Общая схема алгоритма циклического сдвига для топологии типа **решетка-тор** состоит в следующем. Пусть процессоры перенумерованы по строкам решетки от 0 до $p-1$. На первом этапе организуется циклический сдвиг с шагом $q \bmod \sqrt{p}$ по каждой строке в отдельности (если при реализации такого сдвига сообщения передаются через правые границы строк, то после выполнения каждой такой передачи необходимо осуществить компенсационный сдвиг вверх на 1 для процессоров первого столбца решетки). На втором этапе реализуется циклический сдвиг вверх с шагом $\lfloor q/\sqrt{p} \rfloor$ для каждого столбца решетки. Общая длительность всех операций рассылки определяется соотношением:

$$t_{nd} = (t_n + mt_k)(2\lfloor \sqrt{p}/2 \rfloor + 1). \quad (19)$$

Для гиперкуба алгоритм циклического сдвига может быть получен путем логического представления топологии гиперкуба в виде кольцевой структуры. Для получения такого представления установим взаимно-однозначное соответствие между вершинами кольца и гиперкуба. Для наглядности на рис. 1 приводится вид гиперкуба для размерности $N=3$ с указанием для каждого процессора гиперкуба соответствующей вершины кольца. Положительным свойством выбора такого соответствия является тот факт, что для любых двух вершин в кольце, расстояние между которыми равно $l=2^i$ для некоторого значения i , путь между соответствующими вершинами в гиперкубе содержит только две линии связи (за исключением случая $i=0$, когда путь в гиперкубе имеет единичную длину).

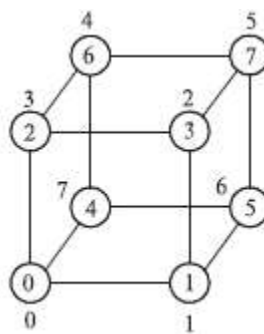


Рис. 1 - Схема отображения гиперкуба на кольцо (в кружках приведены номера процессоров гиперкуба)

Представим величину сдвига q в виде двоичного кода. Количество ненулевых позиций кода определяет количество этапов в схеме реализации операции циклического сдвига. На каждом этапе выполняется операция сдвига с величиной шага, задаваемой наиболее старшей ненулевой позицией

значения q (например, при исходной величине сдвига $q=5=101_2$ на первом этапе выполняется сдвиг с шагом 4, на втором этапе шаг сдвига равен 1). Выполнение каждого этапа (кроме сдвига с шагом 1) состоит в передаче данных по пути, включающему две линии связи. Как результат, верхняя оценка для длительности выполнения операции циклического сдвига определяется соотношением:

$$t_{n\partial} = (t_n + mt_k)(2\log_2 p - 1). \quad (20)$$

Передача пакетов. Использование пересылки пакетов может повысить эффективность выполнения операции циклического сдвига для топологии гиперкуб. Реализация всех необходимых коммуникационных действий в этом случае может быть обеспечена путем отправления каждым процессором всех пересылаемых данных непосредственно процессорам назначения. Применение метода покоординатной маршрутизации позволит избежать коллизий при использовании линий передачи данных (в каждый момент времени для каждого канала будет существовать не более одного готового для отправки сообщения). Длина наибольшего пути при такой рассылке данных определяется как $\log_2 p - \gamma(q)$, где $\gamma(q)$ есть наибольшее целое значение j такое, что 2^j есть делитель величины сдвига q . Тогда длительность операции циклического сдвига может быть охарактеризована при помощи выражения

$$t_{n\partial} = t_n + mt_k + t_c(\log_2 p - \gamma(q)) \quad \begin{matrix} (2 \\ 1) \end{matrix}$$

Методы логического представления топологии коммуникационной среды

Ряд алгоритмов передачи данных допускает более простое изложение при использовании вполне определенных топологий сети межпроцессорных соединений. Кроме того, многие методы коммуникации могут быть получены при помощи того или иного логического представления исследуемой топологии. Как результат, важным моментом при организации параллельных вычислений является возможность логического представления разнообразных топологий на основе конкретных (физических) межпроцессорных структур.

Способы логического представления (отображения) топологий характеризуются следующими тремя основными характеристиками:

- уплотнение дуг (congestion), выражаемое как максимальное количество дуг логической топологии, которые отображаются в одну линию передачи физической топологии;
- удлинение дуг (dilation), определяемое как путь максимальной длины физической топологии, на который отображается дуга логической топологии;
- увеличение вершин (expansion), вычисляемое как отношение количества вершин в логической и физической топологиях.

Для рассматриваемых топологий ограничимся изложением вопросов отображения топологий кольца и решетки на гиперкуб. Предлагаемые ниже

подходы для логического представления топологий характеризуются единичными показателями уплотнения и удлинения дуг.

Представление кольцевой топологии в виде гиперкуба

Установление соответствия между кольцевой топологией и гиперкубом может быть выполнено при помощи двоичного рефлексивного кода Грея $G(i, N)$ (binary reflected Gray code), определяемого в соответствии с выражениями:

$$G(0, 1) = 0, \quad G(1, 1) = 1, \\ G(i, s + 1) = \begin{cases} G(i, s), & i < 2^s, \\ 2^s + G(2^{s+1} - 1 - i, s), & i \geq 2^s, \end{cases} \quad (22)$$

где i задает номер значения в коде Грея, а N есть длина этого кода. Для иллюстрации подхода в табл. 1 показывается отображение кольцевой топологии на гиперкуб для сети из $p=8$ процессоров.

Важное свойство кода Грея: соседние значения $G(i, N)$ и $G(i+1, N)$ имеют только одну различающуюся битовую позицию. Как результат, соседние вершины в кольцевой топологии отображаются на соседние процессоры в гиперкубе.

Таблица 1 - Отображение кольцевой топологии на гиперкуб при помощи кода Грея

Код Грея для $N=1$	Код Грея для $N=2$	Код Грея для $N=3$	Номера процессоров	
			гиперкуба	кольца
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

Отображение топологии решетки на гиперкуб

Отображение топологии решетки на гиперкуб может быть выполнено в рамках подхода, использованного для кольцевой структуры сети.

Тогда для отображения решетки $2^r \times 2^s$ на гиперкуб размерности $N=r+s$ можно принять правило, что элементу решетки с координатами (i, j) соответствует процессор гиперкуба с номером:

$$G(i, r) \parallel G(j, s),$$

где операция $\parallel$ означает конкатенацию кодов Грея.

Оценка трудоемкости операций передачи данных для кластерных систем

Для кластерных вычислительных систем одним из широко применяемых способов построения коммуникационной среды является использование концентраторов (hub) или коммутаторов (switch) для объединения процессорных узлов кластера в единую вычислительную сеть. В этих случаях топология сети кластера представляет собой полный граф, в котором, однако, имеются определенные ограничения на одновременность выполнения коммуникационных операций. Так, при использовании концентраторов передача данных в каждый текущий момент может выполняться только между двумя процессорными узлами; коммутаторы могут обеспечивать взаимодействие нескольких непересекающихся пар процессоров.

Другое часто применяемое решение при создании кластеров состоит в использовании метода передачи пакетов (часто реализуемого на основе стека протоколов TCP/IP) в качестве основного способа выполнения коммуникационных операций.

Если выбрать для дальнейшего анализа кластеры данного распространенного типа (топология в виде полного графа, пакетный способ передачи сообщений), то трудоемкость операции коммуникации между двумя процессорными узлами может быть оценена в соответствии с выражением (модель А)

$$t_{n\partial}(m) = t_n + m \cdot t_k + t_c; \quad (23)$$

оценка подобного вида следует из соотношений для метода передачи пакетов при единичной длине пути передачи данных, т.е. при $l=1$. Отмечая возможность подобного подхода, вместе с этим можно заметить, что в рамках рассматриваемой модели время подготовки данных t_n предполагается постоянным (не зависящим от объема передаваемых данных), время передачи служебных данных t_c не зависит от количества передаваемых пакетов и т.п. Эти предположения не в полной мере соответствуют действительности, и временные оценки, получаемые в результате использования модели, могут не обладать необходимой точностью.

С учетом приведенных замечаний, схема построения временных оценок может быть уточнена; в рамках новой расширенной модели трудоемкость передачи данных между двумя процессорами определяется в соответствии со следующими выражениями (модель В):

$$t_{n\partial} = \begin{cases} t_{\text{нач}_0} + m \cdot t_{\text{нач}_1} + (m + V_c) \cdot t_k, & n = 1 \\ t_{\text{нач}_0} + (V_{\text{max}} - V_c) \cdot t_{\text{нач}_1} + (m + V_c \cdot n) \cdot t_k, & n > 1 \end{cases} \quad (24)$$

где $n = \lceil m / (V_{\text{max}} - V_c) \rceil$ есть количество пакетов, на которое разбивается передаваемое сообщение, величина V_{max} определяет максимальный размер пакета, который может быть доставлен в сети (по умолчанию для операционной системы MS Windows в сети Fast Ethernet $V_{\text{max}}=1500$ байт), а V_c есть объем служебных данных в каждом из пересылаемых пакетов (для протокола TCP/IP,

ОС Windows 2000 и сети Fast Ethernet $v_c=78$ байт). Поясним также, что в приведенных соотношениях константа $t_{нач0}$ характеризует аппаратную составляющую латентности и зависит от параметров используемого сетевого оборудования, значение $t_{нач1}$ задает время подготовки одного байта данных для передачи по сети. Как результат, величина латентности

$$t_n = t_{нач0} + \nu \cdot t_{нач1} \quad (25)$$

увеличивается линейно в зависимости от объема передаваемых данных. При этом предполагается, что подготовка данных для передачи второго и всех последующих пакетов может быть совмещена с пересылкой по сети предшествующих пакетов и латентность, тем самым, не может превышать величины:

$$t_n = t_{нач0} + (V_{max} - V_c) \cdot t_{нач1}. \quad (26)$$

Помимо латентности, в предлагаемых выражениях для оценки трудоемкости коммуникационной операции можно уточнить также правило вычисления времени передачи данных

$$(m + V_c \cdot n) \cdot t_k, \quad (27)$$

что позволяет теперь учитывать эффект увеличения объема передаваемых данных при росте числа пересылаемых пакетов за счет добавления служебной информации (заголовков пакетов).

Завершая анализ проблемы построения теоретических оценок трудоемкости коммуникационных операций, следует отметить, что для практического применения перечисленных моделей необходимо выполнить оценку значений параметров используемых соотношений. В этом отношении полезным может оказаться использование и более простых способов вычисления временных затрат на передачу данных – одной из известных схем подобного вида является подход, в котором трудоемкость операции коммуникации между двумя процессорными узлами кластера оценивается в соответствии с выражением:

$$t_{nd}(m) = t_n + mt_k \quad (28)$$

это модель с, предложенная Хокни (the Hockney model).

ЗАДАНИЕ

1. Разработайте алгоритмы выполнения основных операций передачи данных для топологии сети в виде 3-мерной решетки.
2. Разработайте алгоритмы выполнения основных операций передачи данных для топологии сети в виде двоичного дерева.
3. Примените модель В из теоретической части для оценки временной сложности операций передачи данных. Сравните получаемые показатели.
4. Примените модель С из теоретической части для оценки временной сложности операций передачи данных. Сравните получаемые показатели.

5. Разработайте алгоритмы логического представления двоичного дерева для различных физических топологий сети.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные характеристики используются для оценки топологии сети передачи данных? Приведите значения характеристик для конкретных типов коммуникационных структур (полный граф, линейка, решетка и др.).
2. Какие основные методы применяются при маршрутизации передаваемых данных по сети?
3. В чем состоят основные методы передачи данных? Приведите для этих методов аналитические оценки времени выполнения.
4. Какие операции передачи данных могут быть выделены в качестве основных?
5. В чем состоят алгоритмы выполнения передачи данных от одного процессора всем процессорам сети для топологий кольца, решетки и гиперкуба? Приведите оценки временной трудоемкости для этих алгоритмов.
6. В чем состоят алгоритмы выполнения передачи данных от всех процессоров всем процессорам сети для топологий кольца, решетки и гиперкуба? Приведите оценки временной трудоемкости для этих алгоритмов.
7. В чем состоят возможные алгоритмы выполнения операции редукции? Какой из алгоритмов является наилучшим по времени выполнения?
8. В чем состоит алгоритм выполнения операции циклического сдвига?
9. В чем состоит полезность использования логических топологий? Приведите примеры алгоритмов логического представления структуры коммуникационной сети.
10. В чем состоит различие моделей для оценки времени выполнения операций передачи данных в кластерных вычислительных системах? Какая модель является более точной? Какая модель может быть использована для предварительного анализа временной трудоемкости коммуникационных операций?

3 ПРИНЦИПЫ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ

Цель: Углубить и закрепить знания, полученные на лекциях по методике разработки параллельных алгоритмов.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Известно, что перенос последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи. Для того чтобы построить эффективный параллельный алгоритм, строго говоря, следует провести анализ графа алгоритма и решить задачу отображения. Решение такой задачи оптимизации на графах требует значительных усилий и высокой квалификации.

На практике разработку параллельного алгоритма обычно осуществляет специалист, работающий в некоторой предметной области, не всегда владеющий методами дискретной оптимизации. С другой стороны, строгое решение этой задачи требуется далеко не всегда. Обычно ограничения, связанные с типовым набором доступных архитектур, все равно вынуждают исследователя находить некоторое приемлемое для него решение, руководствуясь не вполне строгими, но проверенными на практике приемами и правилами.

В частности, если в конкретной задаче элементы некоторого массива исходных данных могут обрабатываться независимо друг от друга, то эти правила обычно очевидны и позволяют строить весьма эффективные параллельные алгоритмы. В этом случае задача переработки может свестись к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы в этом случае зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы). По мере увеличения числа (а значит уменьшения размеров) фрагментов данных, объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Иногда используется следующий простой способ построения эффективной параллельной программы, совмещенный с этапом ее отладки. Размеры фрагментов массива исходных данных уменьшают (соответственно увеличивают число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что

накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров.

Совокупность методов и приёмов распараллеливания, не требующих строгого решения задачи отображения графа алгоритма на граф вычислительной системы, будем называть *инженерным* подходом. В настоящем разделе в рамках этого подхода рассматриваются некоторые правила и приемы построения параллельных алгоритмов, выработанные на основе опыта и здравого смысла. Успешность применения этих методов в значительной степени будет зависеть от соответствия структуры построенного параллельного алгоритма типу его внутреннего параллелизма. Поэтому начнем с рассмотрения классификации алгоритмов по этому признаку.

Способность алгоритма к распараллеливанию потенциально связана с одним из двух (или одновременно с обоими) внутренних свойств, которые характеризуются как *параллелизм задач* (message passing) и *параллелизм данных* (data parallel). Если алгоритм основан на параллелизме задач, вычислительная задача разбивается на несколько, относительно самостоятельных подзадач, каждая из которых загружается в "свой" процессор. Каждая подзадача реализуется независимо, но использует общие данные и/или обменивается результатами своей работы с другими подзадачами. Для реализации такого алгоритма на многопроцессорной системе необходимо выявлять независимые подзадачи, которые могут выполняться параллельно. Часто это оказывается далеко не очевидной и весьма трудной задачей. Методика решения этой задачи будет рассмотрена в следующем разделе.

При наличии в алгоритме свойства параллелизма данных, одна операция может выполняться сразу над всеми элементами массива данных. В этом случае различные фрагменты массива могут обрабатываться независимо на разных процессорах. Для алгоритмов этого типа распределение данных между процессорами обычно осуществляется до выполнения задачи на ЭВМ. Построение алгоритма, обладающего свойством параллелизма данных, и подбор подходящей архитектуры компьютера для него могут выполняться с использованием достаточно простых методик, не требующих применения сложного математического аппарата.

Для того чтобы в полной мере использовать структурные свойства алгоритма, необходимо прежде всего выявить, к какому типу он относится. Ниже приводится общая классификация алгоритмов, с точки зрения типа параллелизма.

Алгоритмы, использующие параллелизм данных (Data Parallelism). Этот тип параллелизма характерен для численных алгоритмов обработки, имеющих дело с большими массивами, представляемыми, например, в виде векторов и матриц. Простейшим примером такой задачи является, например, процедура перемножения двух матриц.

Алгоритмы с распределением данных (Data Partitioning). Это разновидность параллелизма данных, при котором пространство данных может быть разделено на непересекающиеся области, с каждой из которых связаны

независимые процессы, оперирующие каждый со своими данными. Требуется лишь редкий обмен между этими процессами.

Релаксационные алгоритмы (Relaxed Algorithm). Алгоритм может быть представлен в виде независимых процессов без синхронизации связи между ними, но процессоры должны иметь доступ к общим данным.

Алгоритмы с синхронизацией итераций (Synchronous Iteration). Многие из стандартных численных итерационных параллельных алгоритмов требуют синхронизации в конце каждой итерации, заключающейся в том, что разрешение на начало следующей итерации дается после того, как все процессоры завершили предыдущую итерацию.

Самовоспроизводящиеся задачи (Replicated Workers). Для задач этого класса создается и поддерживается центральный пул (хранилище) похожих вычислительных задач. Параллельно реализуемые процессы осуществляют выбор задач из пула, выполнение требуемых вычислений и добавление новых задач к пулу. Вычисления заканчиваются, когда пул пуст. Эта технология характерна для исследований графа или дерева.

Конвейерные вычисления (Pipelined Computation). Этот тип вычислений характерен для процессов, которые могут быть представлены в виде некоторой регулярной структуры, например, в виде кольца или двумерной сети. Каждый процесс, находящийся в узле этой структуры, реализует определенную фазу вычислений.

Нетрудно заметить, что некоторые алгоритмы из приведенного списка обладают явно выраженными свойствами параллелизма задач или параллелизма по данным. Вместе с тем ряд алгоритмов в той или иной мере обладают *обоими* указанными свойствами. Это следует учитывать при выборе способа и схемы декомпозиции задачи на подзадачи. Далее рассматривается основанная на разумных предположениях схема этапов построения параллельного алгоритма, основанного на декомпозиции данных.

Технология подготовки параллельных приложений в виде следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.
2. Определение для сформированного набора подзадач информационных взаимодействий.
3. Масштабирование подзадач, определение количества процессоров.
4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

После выполнения указанных этапов и оценки качества параллельного алгоритма (ускорения, эффективности, масштабируемости) может оказаться необходимым повторение некоторых (или всех) этапов. Если в результате ряда попыток желаемые показатели качества не достигаются, следует проанализировать и, возможно, изменить математическую постановку задачи с целью построения новой вычислительной схемы.

Следует заметить, что указанная последовательность этапов носит

условный характер. Часто, приступая к разработке параллельного алгоритма, пользователь ориентируется на конкретную вычислительную систему, в частности, может быть известно возможное число доступных процессоров. Ясно, что на этапе декомпозиции по данным следует использовать эту информацию для выбора числа областей, определяющих число подзадач.

Если точное число процессоров неизвестно, но заданы границы доступного решающего поля, можно начать с масштабирования базового набора задач, а затем выполнить декомпозицию и выявление связей по информации. Другими словами, в приведенной общей схеме необходимым является лишь содержание этапов, в то время как сами этапы могут выполняться в любой последовательности, притом любой из них может оказаться как начальным, так и завершающим. На рис. 1 показана возможная схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений.

Если базовые подзадачи определены, установление информационных зависимостей между ними обычно не вызывает больших затруднений. При проведении анализа информационных зависимостей между подзадачами следует различать:

- локальные (на соседних процессорах) и глобальные (в которых принимают участие все процессоры) схемы передачи данных;
- структурные (соответствующие типовым топологиям коммуникаций) и произвольные способы взаимодействия;
- статические (задаваемые на этапе проектирования) или динамические (определяемые в ходе выполняемых вычислений);
- синхронные (следующая операция выполняется после выполнения предыдущей операции всеми процессорами) и асинхронные способы взаимодействия (процессы могут не дожидаться полного завершения действий по передаче данных)

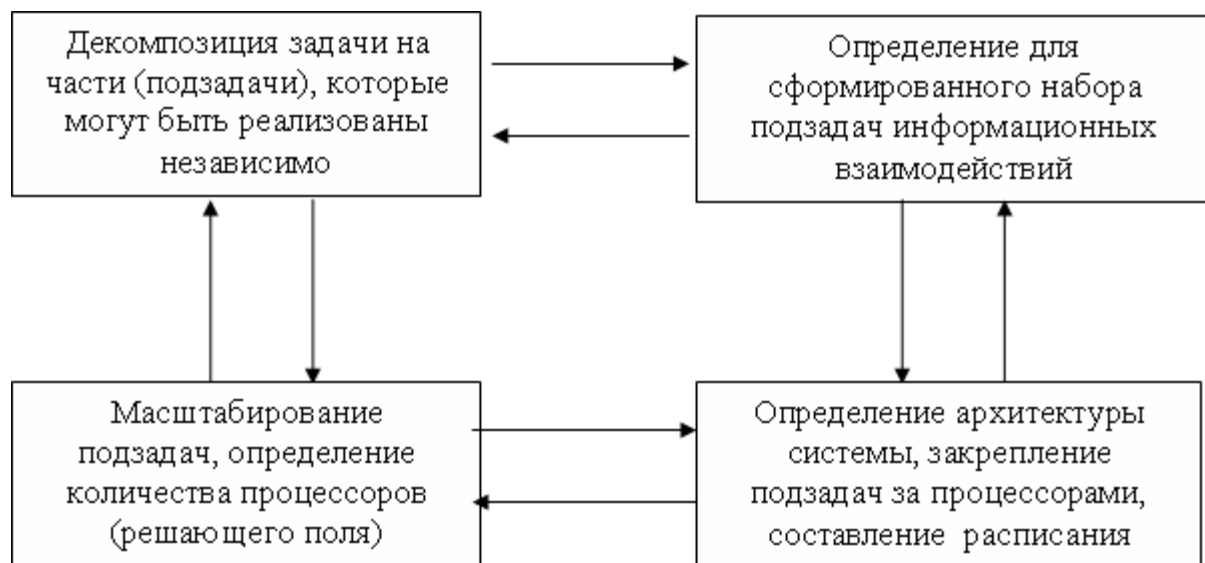


Рисунок 1 - Общая схема взаимосвязи этапов разработки параллельных алгоритмов

Если количество подзадач (областей данных) отличается от числа процессоров, то необходимо выполнить *масштабирование* параллельного алгоритма. Для сокращения количества подзадач укрупняют области исходных данных, притом в первую очередь объединяют области, для которых соответствующие подзадачи обладают высокой степенью информационной взаимозависимости. Если число подзадач меньше числа доступных процессоров, выполняют декомпозицию. Масштабирование облегчается, если правила агрегации и декомпозиции параметрически зависят от числа процессоров.

Распределение подзадач между процессорами очевидно, если количество областей данных совпадает с числом имеющихся процессоров, а топология сети передачи данных – полный граф (все процессоры связаны между собой). Если это не так, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных. Требование минимизации информационных обменов между процессорами может вступить в противоречие с условием равномерной загрузки. Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений изменяется в ходе решения задачи. При этом необходимо перераспределение базовых подзадач между процессорами (динамическая балансировка) в ходе выполнения программы.

Центральной проблемой, как уже неоднократно указывалось выше, является выделение базовых подзадач на этапе декомпозиции. Эта проблема имеет много аспектов, в следующем разделе кратко рассматриваются лишь некоторые важнейшие.

Описанная выше схема этапов может использоваться также и для построения параллельного алгоритма, который характеризуется параллелизмом задач. При этом содержание этапов может существенно отличаться. В частности, центральной проблемой в этом случае является выявление взаимно независимых операторов, которые могут выполняться параллельно и независимо. Эти вопросы будут обсуждаться в следующей лекции.

Способ разделения вычислений на независимые части зависит от того, насколько полно решаемая задача обладает свойством декомпозируемости по данным, определяемого местом алгоритма в приведенной классификации. Если задача допускает реализацию в классе алгоритмов *с распределением данных (Data Partitioning)*, распараллеливание на подзадачи существенно облегчается. В данном случае одна операция или совокупность операций выполняются над всеми элементами массива данных, а задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. При этом обычно предъявляются требования обеспечить:

- примерно равный объем вычислений в выделяемых подзадачах;
 - минимальный информационный обмен данными между процессорами.
- Рассмотрим выполнение этих требований при различных условиях. Простейший и наиболее благоприятный с точки зрения организации

параллельных вычислений случай, когда вся область исходных данных задачи может быть разделена на непересекающиеся области любых размеров, а вычисления в каждой области могут вестись независимо. Ясно, что в этом случае задача декомпозиции чрезвычайно проста: необходимо всю область разбить на подобласти, число которых равно числу доступных процессоров, а размеры подобластей подобрать так, чтобы обеспечить их равномерную загрузженность, с учетом производительности каждого.

С точки зрения организации вычислений обычно более удобной является декомпозиция на области, с границами в виде прямых линий и плоскостей. На рисунке 2 приведены примеры наиболее широко используемых регулярных структур базовых подзадач, при декомпозиции по данным.

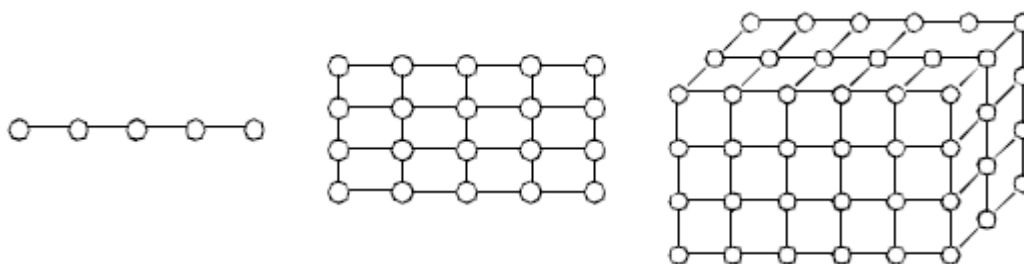


Рисунок 2 - Регулярные одно-, двух- и трехмерные структуры базовых подзадач после декомпозиции данных

Для большинства практических задач при декомпозиции по данным вычисления в каждой области не могут быть полностью независимыми. В частности, после каждой итерации (проведения вычислений во всех точках области) возникает потребность обмена результатами вычислений на границах соседних областей. Это, например, характерно для большинства сеточных методов, в которых для вычисления значения функции в некотором узле используются ее значения в нескольких соседних узлах. В этом случае выполнение указанных выше требований: сбалансированность загрузки процессоров и минимизация информационных обменов, зависит не только от размеров подобластей, но также и от их формы.

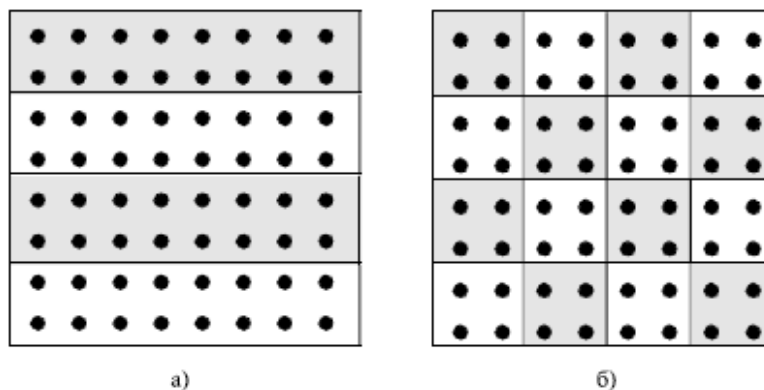


Рисунок 3 - Разделение данных на области:
а — ленточная схема, б — блочная схема

Например, в случае двумерной задачи наиболее часто используется один из двух типов декомпозиции: область может быть разделена на отдельные строки (или последовательные группы строк) – так называемая ленточная схема разделения данных, либо на прямоугольные наборы элементов – блочная схема разделения данных (см. рисунок 3). Возникает естественный вопрос: какая из этих схем декомпозиции «лучше»?

Выбор одной из указанных схем декомпозиции диктуется требованием минимизации пересылок данных между процессорами. Рассмотрим эту задачу для двумерного случая. Будем полагать, что области заданы в виде прямоугольников или квадратов, ширина полос данных в окрестности границ, которыми должны обмениваться процессоры, не зависит от направления границ фрагментов, а объем передаваемых данных определяется длиной сопряженных границ фрагментов. Приведем простой пример декомпозиции двумерной области, имеющей размеры $H \times L$, $H \geq L$. При декомпозиции области данных на четыре подобласти (процессора) общий объем передаваемых данных при ленточном разделении данных (вдоль стороны L) пропорционален $3L$, а при блочном (на равные прямоугольники) – $H + L$. Объем максимального межпроцессорного обмена данными между парами процессоров, обрабатывающих соседние области, составит соответственно L и $H/2$. Нетрудно заметить одинаковый общий и максимальный межпроцессорный обмены имеют место при $H=2L$. Если $H < 2L$, выгоднее блочная декомпозиция, при $H > 2L$ – ленточная. Ясно, что при другом числе процессоров (подобластей) результаты могут оказаться иными.

Если оказалось, что выгоднее блочная декомпозиция, то следующий важный вопрос – выбор размеров блоков. С точки зрения минимизации отношения длины граничных областей к их площади (пропорционального отношению объема межпроцессорного обмена к объему вычислений в данной подобласти) представляется, что форму подобластей следует взять в виде квадратов или прямоугольников близким к квадратам. Однако при этом возникает еще одна проблема.

При разбиении исходной области обработки данных на квадраты одинаковых размеров для фрагментов, расположенных на границах декомпозируемой области, длина границ, сопряженных с соседними фрагментами, а, следовательно и объем передаваемых данных, будет меньше. Указанное различие во времени передачи данных может оказывать существенное влияние на эффективность использования процессоров, если скорость передачи данных низкая. Неэффективность использования процессоров более заметна, когда число областей, на которые разбивается изображение, невелико.

Повышение эффективности использования процессоров может быть достигнуто увеличением размеров областей, находящихся на границах и в углах изображения. В следующем разделе этот вопрос будет детально рассмотрен для случая блочной декомпозиции.

Блочная декомпозиция с учетом локализации подобластей

Известно, что весьма широкий класс задач реализуется в классе алго-

ритмов с *распределением данных (Data Partitioning)*, в которых пространство данных может быть разделено на непересекающиеся области, а вычисления могут осуществляться независимо и требуется лишь редкий обмен между этими процессами. В частности, при моделировании на основе метода конечных элементов, секционной свертке изображений и др. после каждой итерации осуществляется обмен данными, полученными на границах соседних областей. Ясно, что в случае, когда размеры областей, на которые разбивается вся область значений, одинаковы, объемы пересылаемых данных будут различаться в зависимости от места расположения области.

Решим задачу такого разбиения исходной области на квадратные блоки с учетом локализации подобластей, при котором время работы всех процессоров с учетом пересылок максимально сбалансировано. Для простоты рассмотрим случай, когда исходная область квадратная: $X \times X$, где X – число отсчетов одной стороны области. Будем полагать, что для заданных: вычислительного алгоритма и вычислительной системы, известны константы: τ_p – время расчета при обработке одного отсчета (точки) области и τ_n – среднее время, затрачиваемое на передачу информации, необходимой для одной точки области.

Обычно, с точки зрения удобства организации вычислений, всю область данных разбивают на прямоугольные фрагменты. Пусть x – сторона фрагмента, численно равная числу точек. Потребуем, чтобы величина x удовлетворяла неравенству

$$\Delta_{don} \leq (4 \cdot x \cdot \tau_n) / (x^2 \tau_p) \leq (4 \cdot \delta) / x, \quad (1)$$

где $\delta = \tau_n / \tau_p$ – отношение отрезков времени, необходимых для пересылки данных к времени обработки в расчете на одну точку области, а Δ_{don} – допустимая величина отношения времени пересылок к времени обработки внутренней области, задаваемая из условия эффективной загрузки процессоров.

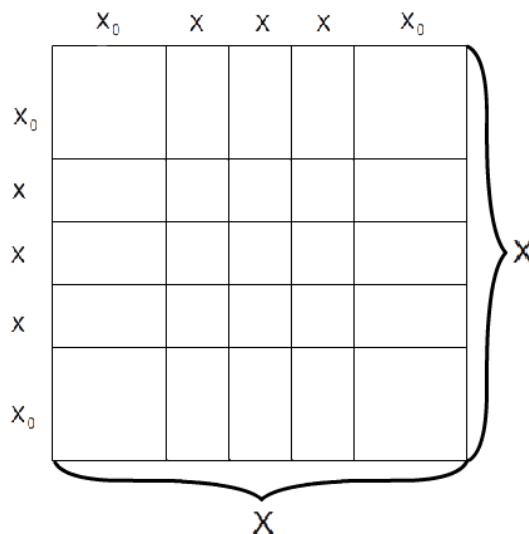


Рис. 4 - Разбиение квадратного изображения на фрагменты

Ясно, что неравенство (1) выполняется также для областей, расположенных на границах исходной области, т.к. они имеют меньшую длину сопряженных границ. Области, находящиеся в углах изображения размером $x_0 \times x_0$, будем называть угловыми, области $x_0 \times x$ ($x \times x_0$) – граничными, а области $x \times x$ – внутренними. Задача заключается в том, чтобы найти x_0 и x такие, чтобы время обработки всех областей с учетом затрат на пересылку было одинаковым (рис. 6.4).

Для простоты полагаем, что ширина полос данных на границах областей, которыми они должны обмениваться, равна одному отсчету, поэтому объем передаваемых данных пропорционален длине границ. Тогда суммарное время обработки с учетом пересылок:

а) для внутренней области:

$$T_{вн} = x^2 \cdot \tau_p + 4 \cdot x \cdot \tau_n, \quad (2)$$

б) для граничной:

$$T_{гр} = x \cdot x_0 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n + x \cdot \tau_n, \quad (3)$$

в) для угловой:

$$T_{угл} = x_0^2 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n. \quad (4)$$

Положим

$$x_0 = k \cdot x, \quad (5)$$

где $1 < k < 1,5$ – коэффициент увеличения угловой (и соответственно граничной) области, который необходимо выбрать из условия балансировки процессоров.

При балансировке процессоров, обрабатывающих внутренние и граничные области, вычислительные затраты на обработку угловых областей существенно возрастают. Поэтому потребуем, чтобы выполнялось равенство

$$T_{угл} = T_{вн}.$$

или

$$k^2 \cdot x + 2 \cdot k \cdot \delta = x + 4 \cdot \delta. \quad (6)$$

Отбрасывая из решений (7) отрицательные значения k , получаем

$$k = \frac{\delta}{x} + \sqrt{\frac{\delta}{x} + 1 + \frac{4 \cdot \delta}{x}}. \quad (7)$$

С учетом неравенства (1) в соответствии с (7) можно записать условие для допустимых значений k :

$$k \leq -\frac{\Delta_{дон}}{4} + \sqrt{\left(\frac{\Delta_{дон}}{4}\right)^2 + 1 + 4 \cdot \Delta_{дон}}. \quad (8)$$

Остается подобрать удовлетворяющее условию (8) наибольшее значение k , при котором целое число n (число полос, на которые разбивается область решений) удовлетворяет равенству

$$(n - 2)x + 2kx = X. \quad (9)$$

Заметим, что обычно отношение δ/x невелико, при этом для значений k , удовлетворяющих (8), время обработки граничных областей не превышает времени обработки угловых и внутренних областей.

Соотношения (8), (9) могут использоваться для выбора начального разбиения исходной области на фрагменты. В действительности эффективность загрузки процессоров будет зависеть от многих других факторов, которые не учитывались в нашей упрощенной модели (например, латентность при передаче данных, а также тот факт, что исходная область может быть не квадратной, а X не обязано делиться без остатка на величину x , и др.).

Для более полного учета влияния всех факторов, которые не принимались во внимание в указанной упрощенной постановке, может использоваться технология итерационного планирования распределения ресурсов. В данном случае ее применение не вызовет дополнительных усложнений по сравнению с описанным в указанной работе вариантом, поскольку задача выбора k однопараметрическая.

Ранее мы уже подчеркивали, что при переносе последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи, а именно: допускает ли задача распараллеливание по данным (*параллелизм данных*) или имеет место лишь *параллелизм задач*. Как уже отмечалось выше, переработка последовательной программы существенно облегчается, если задача допускает распараллеливание по данным. В этом случае задача переработки может свестись к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы будет зависеть от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы). По мере увеличения числа (а значит уменьшения размеров) фрагментов данных объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Можно рекомендовать следующий простой способ построения эффективной программы, основанной на свойстве параллелизма данных. Размеры фрагментов массива исходных данных следует уменьшать (соответственно увеличивать число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров.

Если задача не допускает распараллеливания по данным, т.е. возможен лишь *параллелизм задач*, трудности существенно возрастают. Подход к про-

граммированию, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Для каждой подзадачи пишется своя собственная программа. Чем больше подзадач, тем большее число процессоров можно использовать и тем большего ускорения можно ожидать (если удастся обеспечить равномерную загрузку процессоров и минимизировать обмен данными между ними).

Для построения эффективного кода в данном случае программист должен провести анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсопотребляющих частей. Для этого могут использоваться различные формальные модели (пространственно-временные диаграммы, модели в виде ориентированных графов, сетей Петри и др.).

ЗАДАНИЕ

Разработайте схему параллельных вычислений, используя рассмотренную в лекции методику проектирования и разработки параллельных методов для задачи практического занятия № 1.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. В чем состоят исходные предположения для возможности применения рассмотренной в лекции методики разработки параллельных алгоритмов?
2. Каковы основные этапы проектирования и разработки методов параллельных вычислений?
3. Как определяется модель "подзадачи – сообщения"?
4. Как определяется модель "процессы – каналы"?
5. Какие основные требования должны быть обеспечены при разработке параллельных алгоритмов?
6. В чем состоят основные действия на этапе выделения подзадач?
7. Каковы основные действия на этапе определения информационных зависимостей?
8. В чем состоят основные действия на этапе масштабирования имеющегося набора подзадач?
9. В чем состоят основные действия на этапе распределения подзадач по процессорам вычислительной системы?
10. Как происходит динамическое управление распределением вычислительной нагрузки при помощи схемы "менеджер – исполнитель"?
11. Какой метод параллельных вычислений был разработан для решения гравитационной задачи N тел?
12. Какой способ выполнения операции обобщенного сбора данных является более эффективным?

4 СРЕДА РАЗРАБОТКИ ПРИЛОЖЕНИЙ MICROSOFT VISUAL STUDIO

Цель занятия: изучить интерфейс интегрированной среды разработки приложений на языке высокого уровня Microsoft Visual Studio.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Для написания программ в практических разделах данного учебного пособия будет использоваться компилятор языка C++, а программирование будет вестись в среде **Microsoft Visual Studio 2010**. Предполагается, что на компьютере установлена эта интегрированная среда.

Microsoft Visual Studio 2010 доступна в следующих вариантах:

Express – бесплатная среда разработки, включающая только базовый набор возможностей и библиотек.

Professional – поставка, ориентированная на профессиональное создание программного обеспечения, и командную разработку, при которой созданием программы одновременно занимаются несколько человек.

Premium – издание, включающее дополнительные инструменты для работы и исходным кодом программ и создания баз данных.

Ultimate – наиболее полное издание Visual Studio, включающие все доступные инструменты для написания, тестирования, отладки и анализа программ, а также дополнительные инструменты для работы с базами данных и проектирования архитектуры ПО.

Отличительной особенностью среды **Microsoft Visual Studio 2010** является то, что она поддерживает работу с несколькими языками программирования и программными платформами. Поэтому, перед тем, как начать создание программы на языке C, необходимо выполнить несколько подготовительных шагов по созданию проекта и выбора и настройки компилятора языка C для трансляции исходного кода

После запуска **Microsoft Visual Studio 2010** появляется следующая стартовая страница, которая показана на рисунке 1.

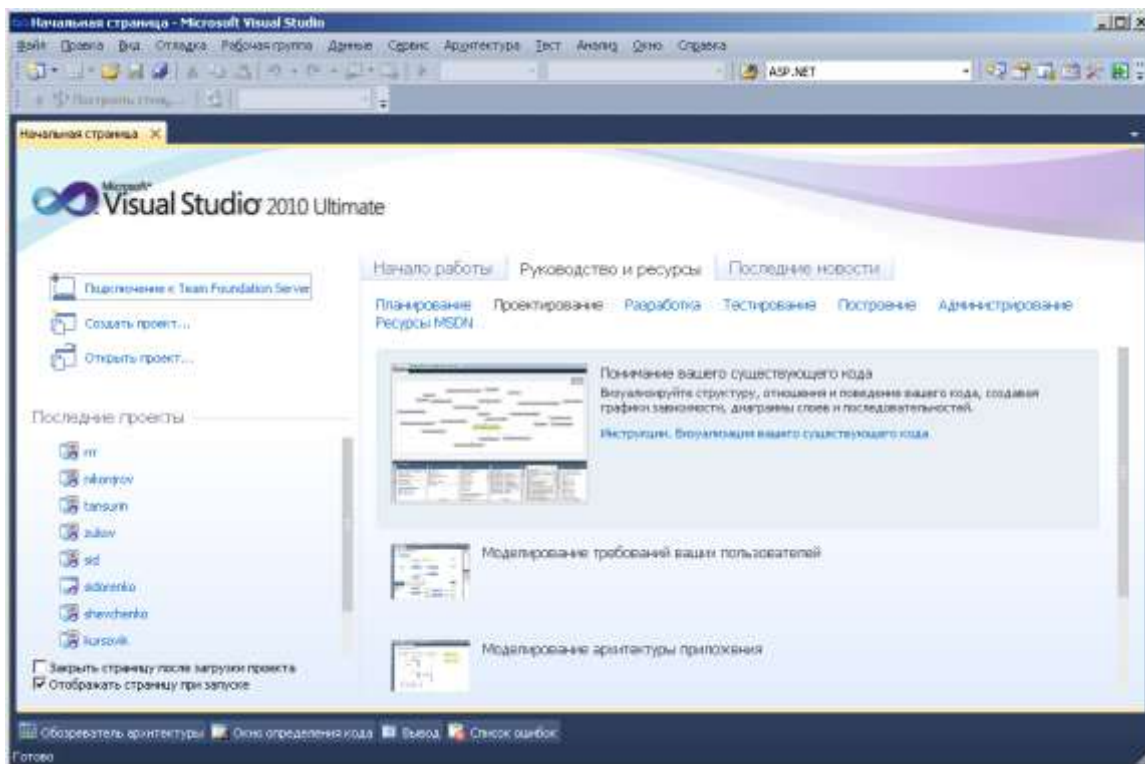


Рисунок 1 - Стартовая страница Visual Studio 2010

Следующим шагом является создание нового проекта. Для этого в меню **Файл** необходимо выбрать **Создать – Проект** (или комбинацию клавиш **Ctrl + Shift + N**). Результат выбора пунктов меню для создания нового проекта показан на рисунке 2.

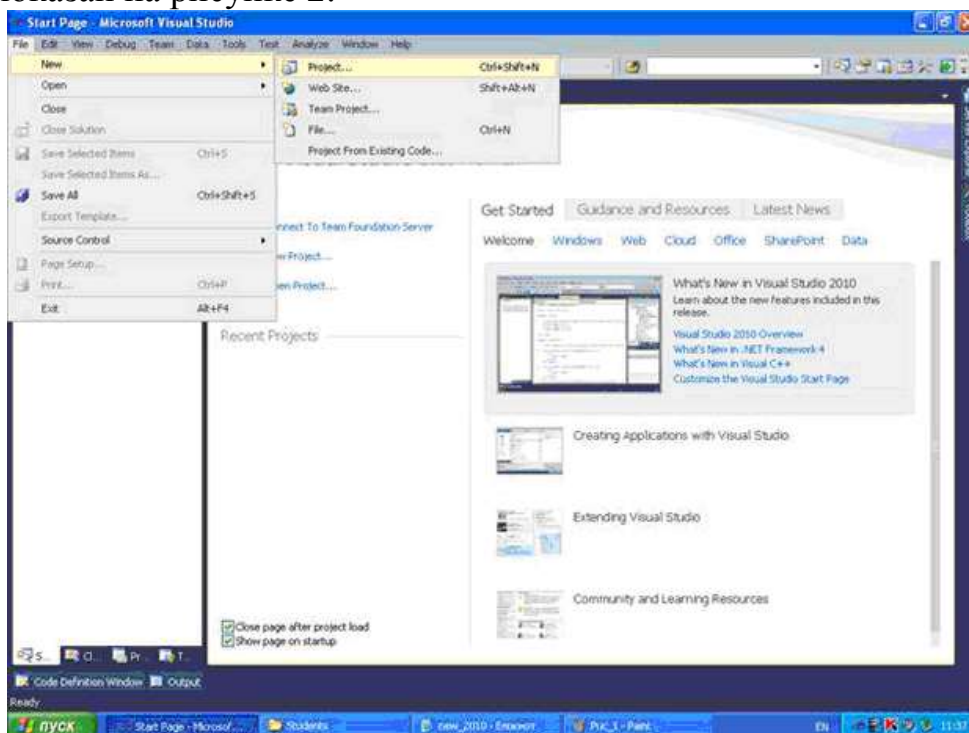


Рисунок 2 - Окно с выбором нового проекта

Среда Visual Studio отобразит окно **New Project**, в котором необходимо

выбрать тип создаваемого проекта. *Проект* (project) используется в Visual Studio для логической группировки нескольких файлов, содержащих исходный код, на одном из поддерживаемых языков программирования, а также любых вспомогательных файлов. Обычно после сборки проекта (которая включает компиляцию всех входящих в проект файлов исходного кода) создается один исполняемый модуль.

В окне **New Project** следует развернуть узел Visual C++, обратиться к пункту Win32 и на центральной панели выбрать Win32 Console Application. Выбор этой опции показан на рисунке 3.

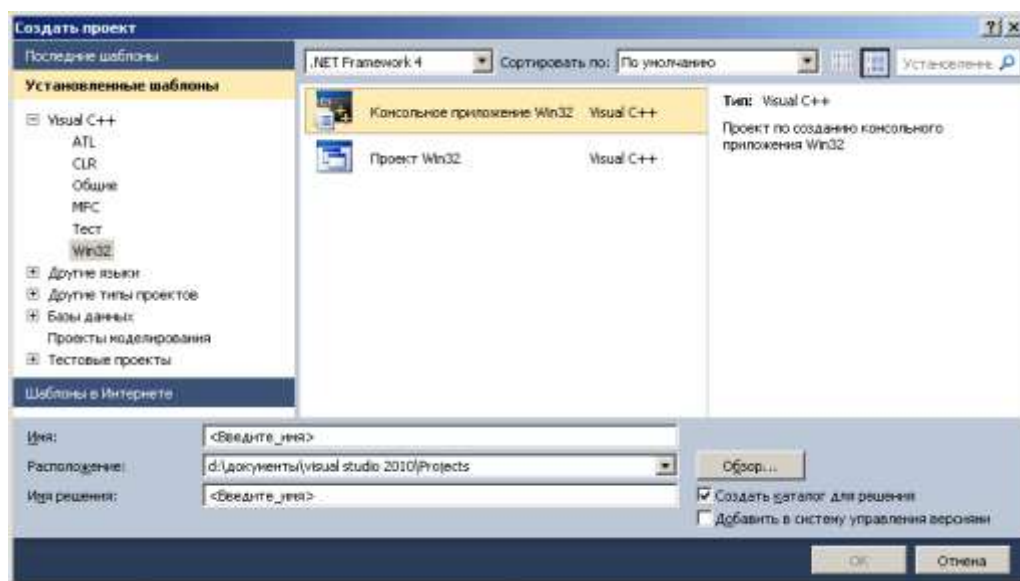


Рисунок 3 - Выбор типа проекта

Затем в поле редактора **Name** (где по умолчанию имеется <Enter_name>) следует ввести имя проекта, например, **hello**. В поле **Location** можно указать путь размещения проекта, или выбрать путь размещения проекта с помощью клавиши (кнопки) Browse. По умолчанию проект сохраняется в специальной папке Projects. Пример выбора имени проекта показано на рисунке 4.

Одновременно с созданием проекта Visual Studio создает решение. *Решение* (solution) – это способ объединения нескольких проектов для организации более удобной работы с ними.

После нажатия кнопки **OK** откроется окно **Win32 Application Wizard** (мастер создания приложений для операционных систем Windows), показанное на рисунке 5.

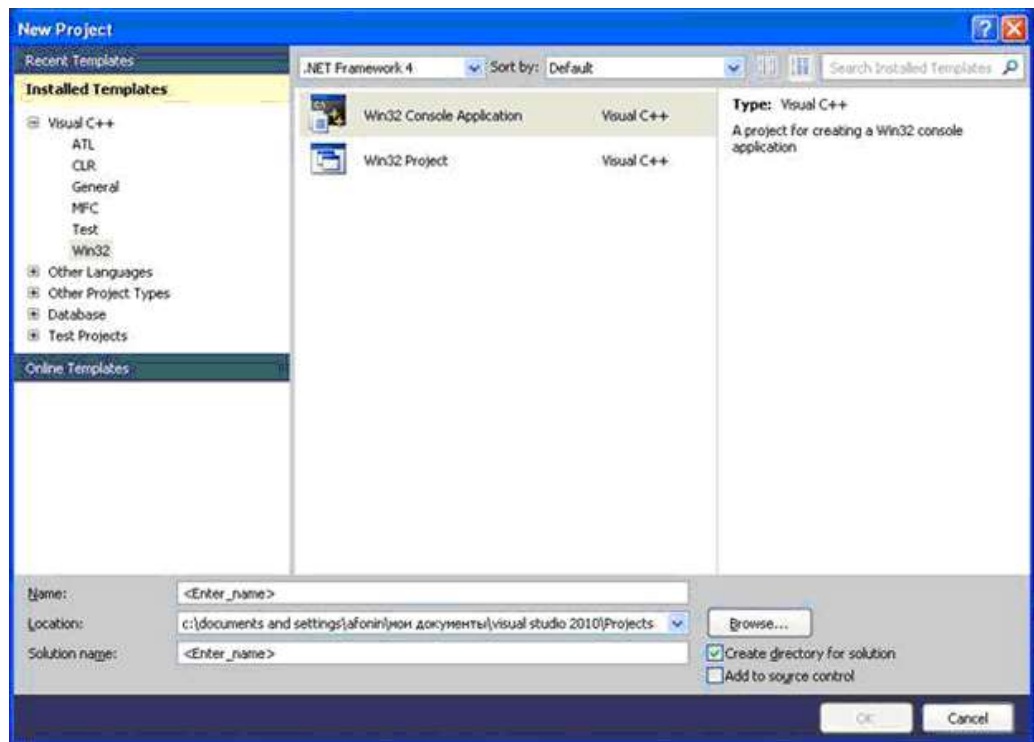


Рисунок 4 - Пример задания имени проекта

Выбор имени проекта может быть достаточно произвольным: допустимо использовать числовое значение, допустимо имя задавать через буквы русского алфавита.

В дальнейшем будем использовать имя, набранное с помощью букв латинского алфавита и, может быть, с добавлением цифр.

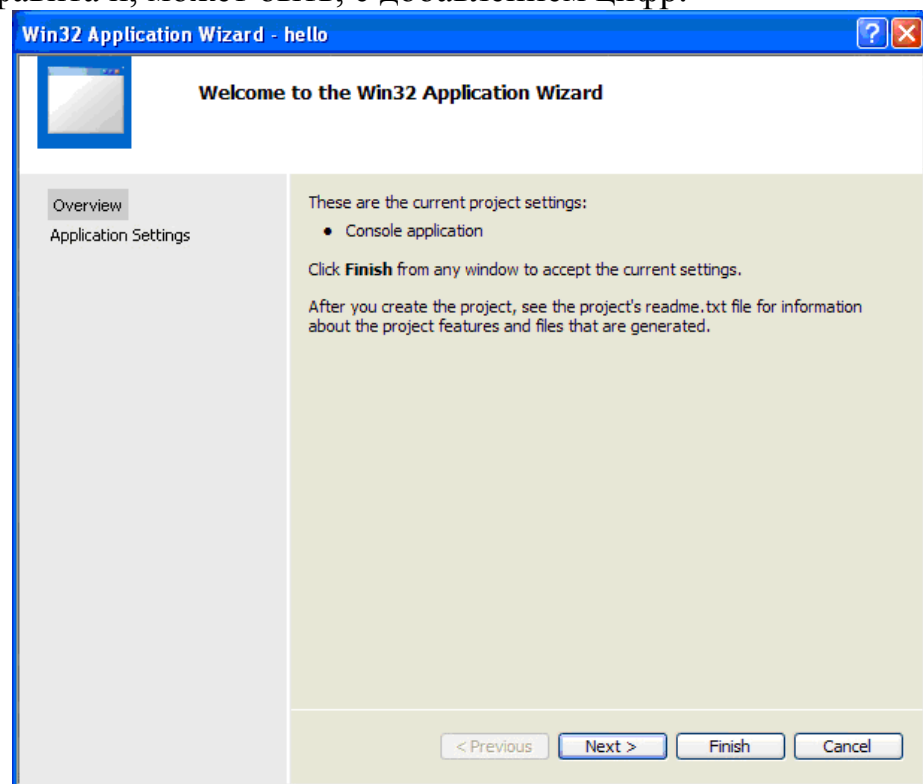


Рисунок 5 - Мастер создания приложения

На первой странице представлена информация о создаваемом проекте, на второй можно сделать первичные настройки проекта. После обращения к странице **Application Settings**, или после нажатия кнопки **Next** получим окно, показанное на рисунке 6.

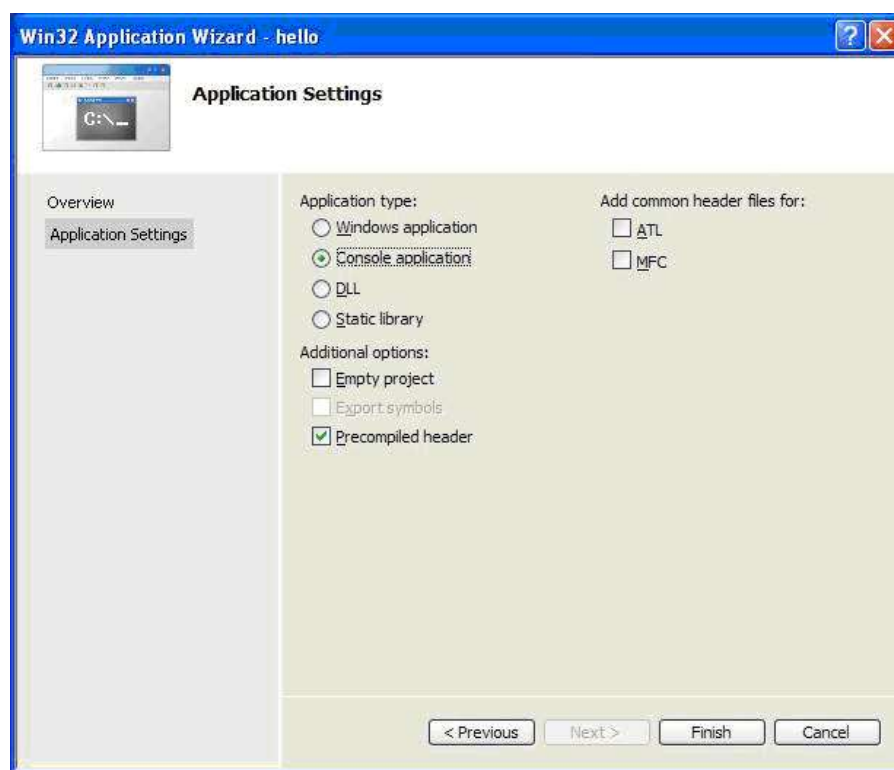


Рисунок 6 - Страница мастера настройки проекта по умолчанию

В дополнительных опциях (**Additional options**) следует поставить галочку в поле **Empty project** (пустой проект) и снять (убрать) галочку в поле **Precompiled header**. Получим экранную форму, показанную на рисунке 7.

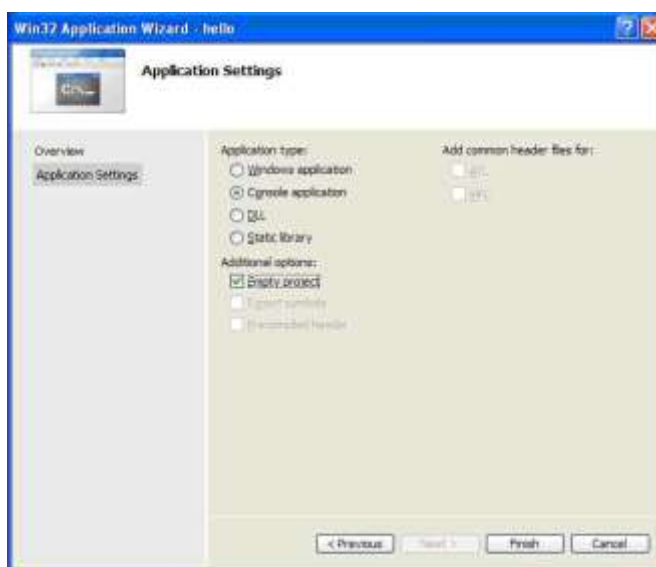


Рисунок 7 - Выполненная настройка мастера приложений
Здесь и далее будут создавать проекты по приведенной схеме, т.е. про-

екты в консольном приложении, которые должны создаваться целиком программистом (за счет выбора **Empty project**). После нажатия кнопки **Finish**, получим экранную форму, показанную на рисунке 2.8, где приведена последовательность действий добавления файла для создания исходного кода к проекту. Стандартный путь для этого: подвести курсор мыши к папке **Source Files** из узла **hello** в левой части открытого проекта приложения, выбрать **Add** и **New Item** (новый элемент).

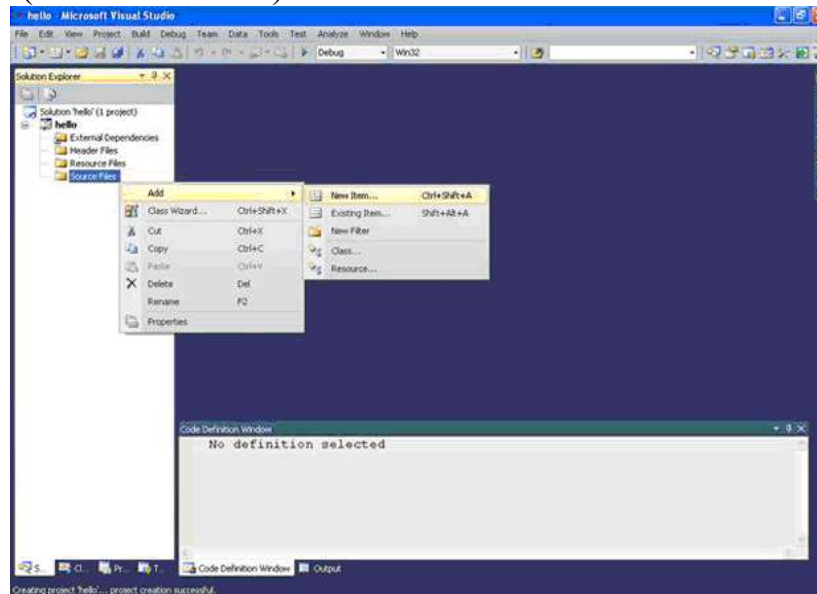


Рисунок 8 - Меню добавления нового элемента к проекту

После выбора (нажатия) **New Item** получим окно, показанное на рисунке 8, где через пункт меню **Code** узла **Visual C++** выполнено обращение к центральной части панели, в которой осуществляется выбор типа файлов. В данном случае требуется обратиться к закладке **C++ File (.cpp)**.

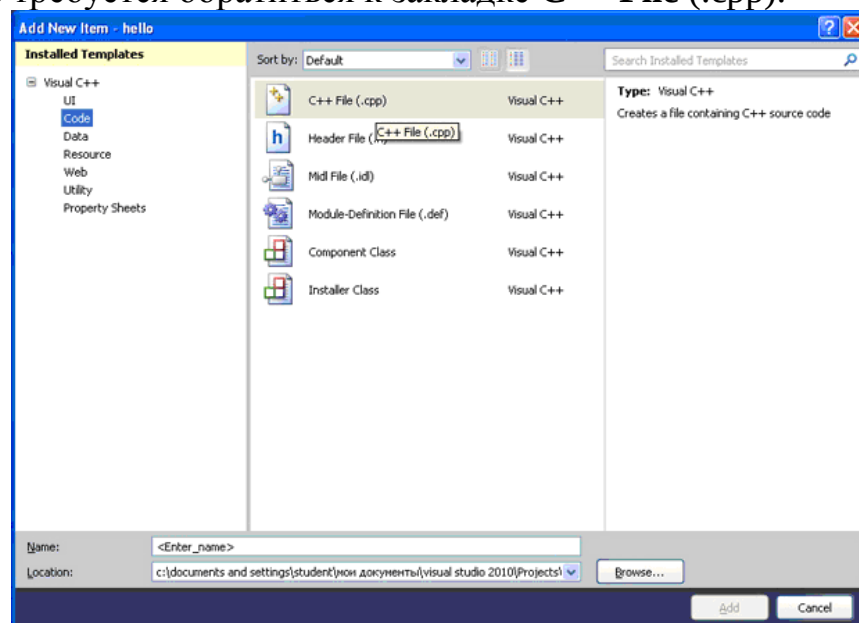


Рисунок 9 - Окно выбора типа файла для подключения к проекту

Теперь в поле редактора **Name** (в нижней части окна) следует задать

имя нового файла и указать расширение **".c"**. Например, **main.c**. Имя файла может быть достаточно произвольным, но имеется негласное соглашение, что имя файла должно отражать его назначение и логически описывать исходный код, который в нем содержится. В проекте, состоящем из нескольких файлов, имеет смысл выделить файл, содержащий главную функцию программы, с которой она начнет выполняться. В данном пособии такому файлу мы будем задавать имя **main.c**, где расширение **.c** указывает на то, что этот файл содержит исходный код на языке **C**, и он будет транслироваться соответствующим компилятором. Программам на языке **C** принято давать расширение **.c**. После задания имени файла в поле редактора **Name**, получим форму, показанную на рисунке 10.

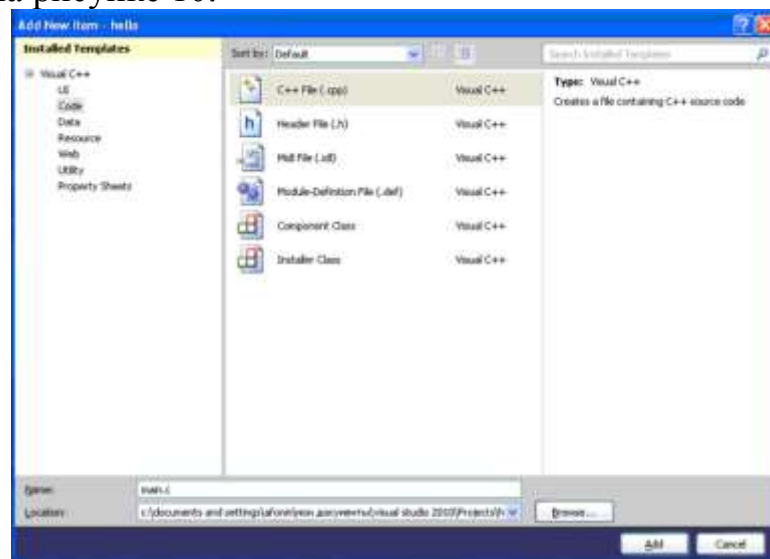


Рисунок 10 - Задание имени файла, подключаемому к проекту

Затем следует нажать кнопку **Add**. Вид среды Visual Studio после добавления первого файла к проекту показан на рисунке 11. Добавленный файл отображается в дереве **Solution Explorer** под узлом Source Files (файлы с исходным кодом), и для него автоматически открывается редактор.

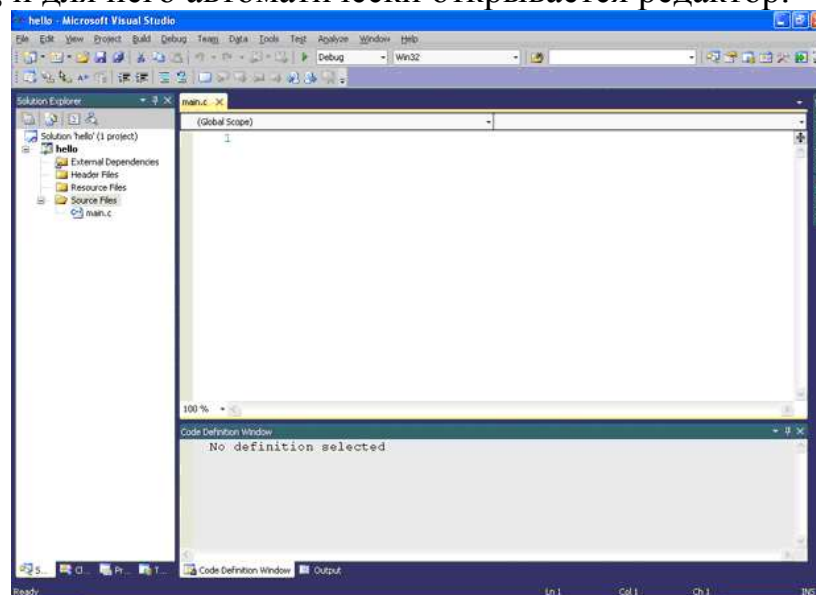


Рисунок 11 - Подключение файла проекта

На рисунке 11 в левой панели в папке **Solution Explorer** отображаются файлы, включенные в проект в папках. Приведем описание.

Папка **Source Files** предназначена для файлов с исходным кодом. В этой папке отображаются файлы с расширением **.c**.

Папка **Header Files** сожержит заголовочные файлы с расширением **.h**.

Папка **Resource Files** содержит файлы ресурсов, например изображения и т. д.

Папка **External Dependencies** отображает файлы, не добавленные явно в проект, но использующиеся в файлах исходного кода, например включенные при помощи директивы `#include`. Обычно в папке **External Dependencies** присутствуют заголовочные файлы стандартной библиотеки, использующиеся в проекте.

Следующий шаг состоит в настройке проекта. Для этого в меню **Project** главного меню следует выбрать **hello Properties** (или с помощью последовательного нажатия клавиш **Alt+F7**). Пример обращения к этому пункту меню показан на рисунке 12.

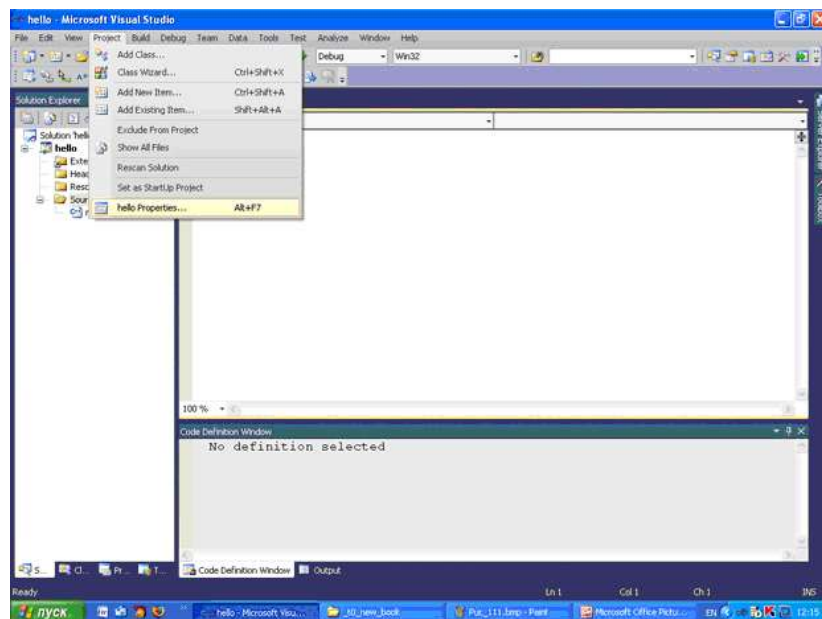


Рисунок 12 - Обращение к странице свойств проекта

После того как произойдет открытие окна свойств проекта, следует обратиться (с левой стороны) к **Configuration Properties**. Появится ниспадающий список, который показан на рисунке 13. Выполнить обращение к узлу **General**, и через него в левой панели выбрать **Character Set**, где установить свойство **Use Multi-byte Character Set**. Настройка **Character Set** (набор символов) позволяет выбрать, какая кодировка символов – ANSI или UNICODE – будет использована при компиляции программы. Для совместимости со стандартом C89 мы выбираем **Use Multi-Byte Character Set**. Это позволяет использовать многие привычные функции, например, функции по выводу

информации на консоль.

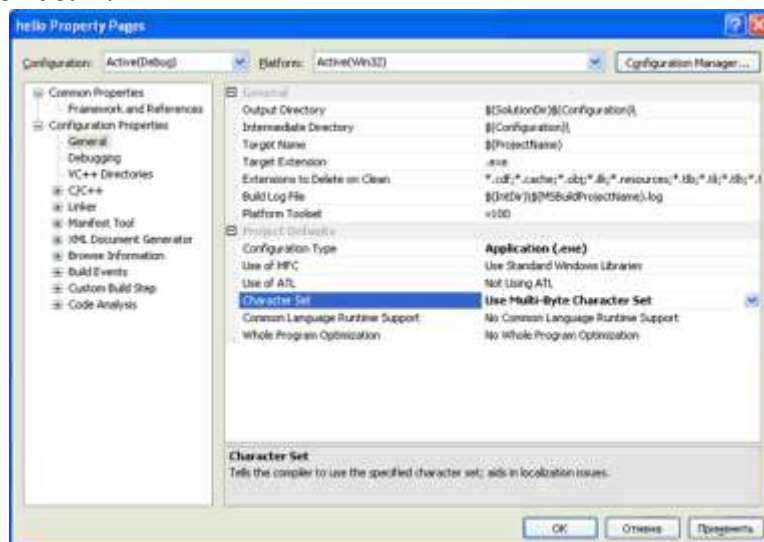


Рисунок 13 - Меню списка свойств проекта

После сделанного выбора, показанного на рисунке 2.13, следует нажать кнопку Применить. Затем следует выбрать узел **C/C++** и в выпадающем меню выбрать пункт **Code Generation**, через который следует обратиться в правой части панели к закладке **Enable C++ Exceptions**, для которой установить **No** (запрещение исключений C++). Результат установки выбранного свойства показан на рисунке 14. После произведенного выбора нажать кнопку **Применить**.

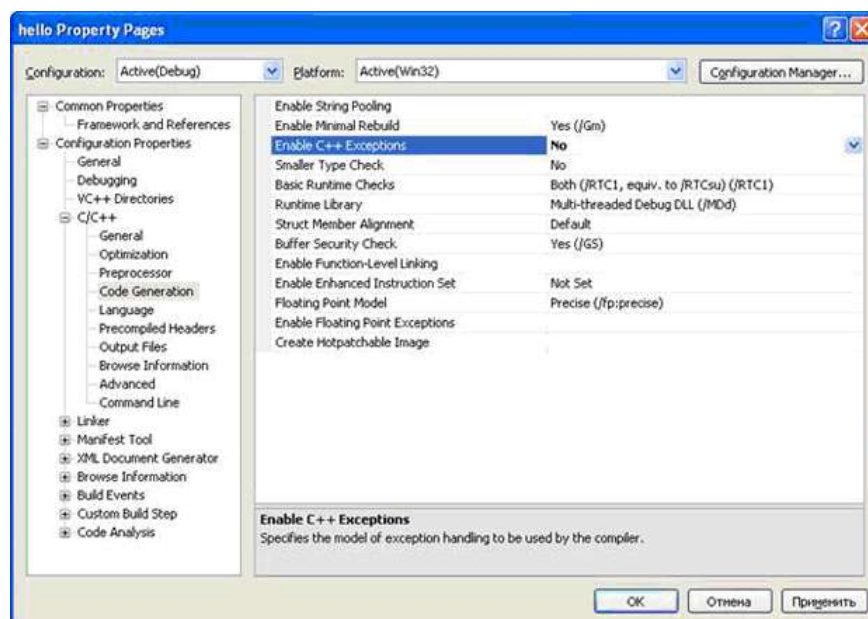


Рисунок 14 - Страница свойств для запрещения исключений C++

Далее в выпадающем меню узла **C/C++** необходимо выбрать пункт **Language** и через него обратиться в правую часть панели, где установить следующие свойства: свойство **Disable Language Extensions** (дополнительные языковые расширения фирмы Microsoft) в **Yes (/Za)**, свойство **Treat wchar_t as Built-in Type** (рассматривать тип `wchar_t` как встроенный тип)

установить в **No (/Zc:wchar_t-)**, свойство **Force Conformance in For Loop Scope** (соответствие стандарту определения локальных переменных в операторе цикла `for`) установить в **Yes (/Zc:forScope)**, свойство **Enable Run-Time Type Info** (разрешить информацию о типах во время выполнения) установить в **No (/GR-)**, свойство **Open MP Support** (разрешить расширение Open MP – используется при написании программ для многопроцессорных систем) установить в **No(/openmp-)**.

Результат выполнения этих действий показан на рисунок 2.15.

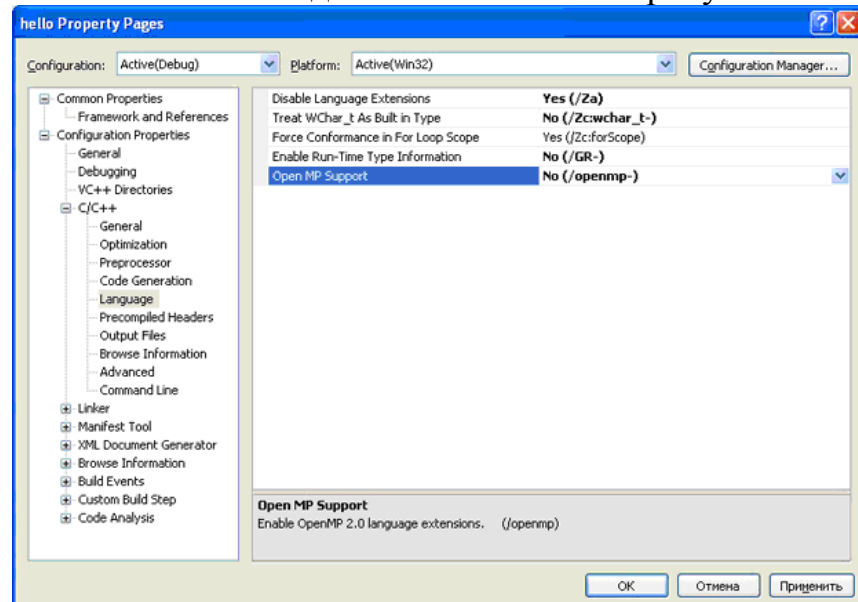


Рисунок 15 - Страница свойств закладки Language

После выполнения указанных действий следует нажать клавишу **Применить**. Далее в ниспадающем списке узла **C/C++** следует выбрать пункт **Advanced** и в правой панели изменить свойство **Compile As** в свойство компиляции языка **C**, т.е. **Compile as C Code (/TC)**. Результат установки компилятора языка **C** показан на рисунке 16.

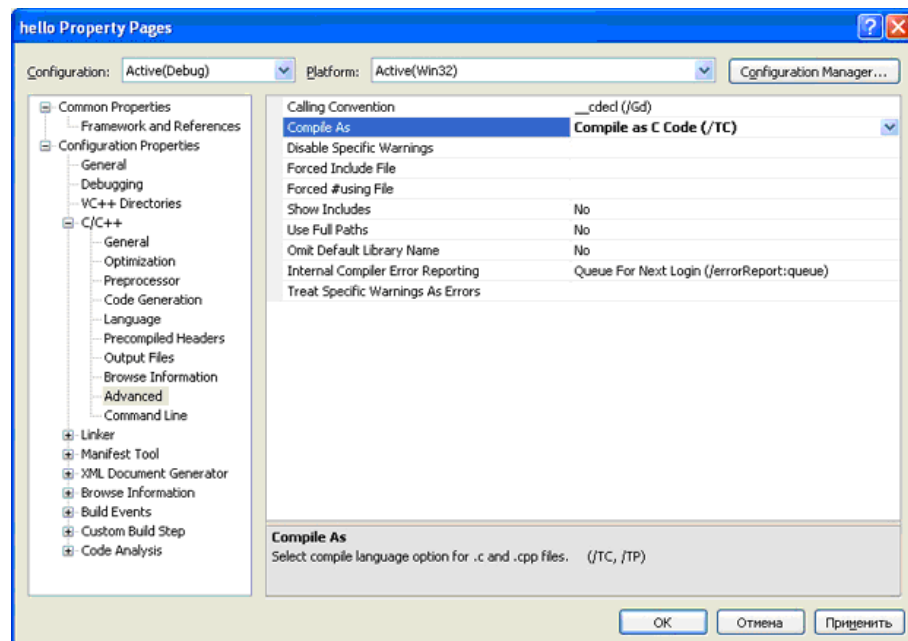


Рисунок 16 - Результат выбора режима компиляции языка С

После нажатия клавиш **Применить** и **ОК** сначала откроется подготовленный проект с пустым полем редактора кода, в котором можно начать писать программы. В этом редакторе наберем программу, выводящую традиционное приветствие "Hello World". Для компиляции созданной программы можно обратиться в меню **Build**, или, например, набрать клавиши **Ctrl+F7**. В случае успешной компиляции получим следующую экранную форму, показанную на рисунке 17.

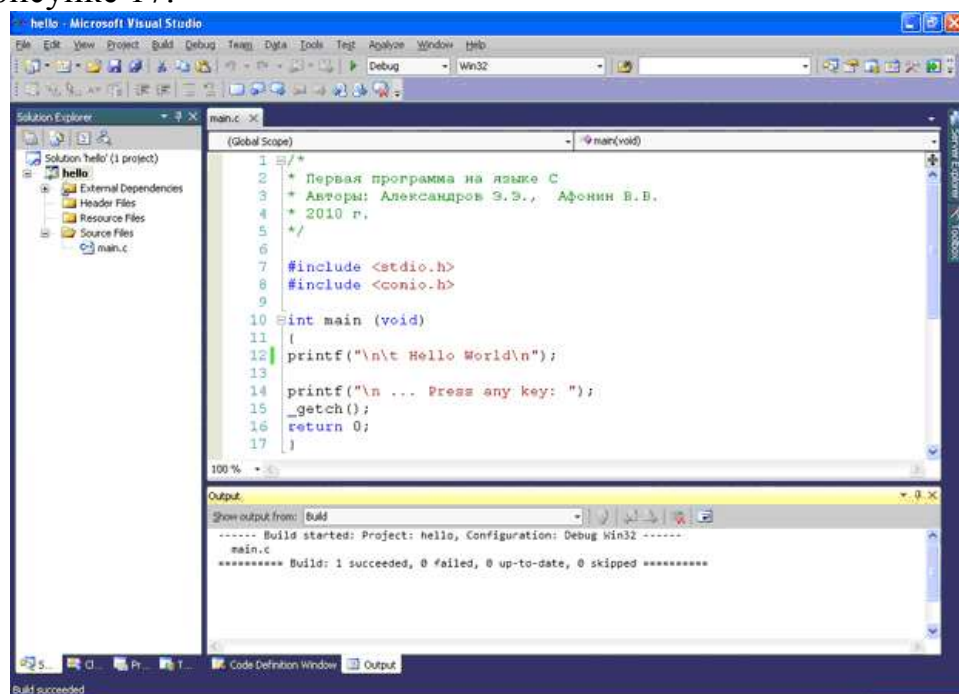


Рисунок 17 - Успешно откомпилированная первая программа на языке С

Для приведенного кода программы запуск на ее исполнение из окна ре-

дактора в Visual Studio 2010 можно нажать клавишу **F5**. На рисунке 18 показан результат исполнения первой программы.

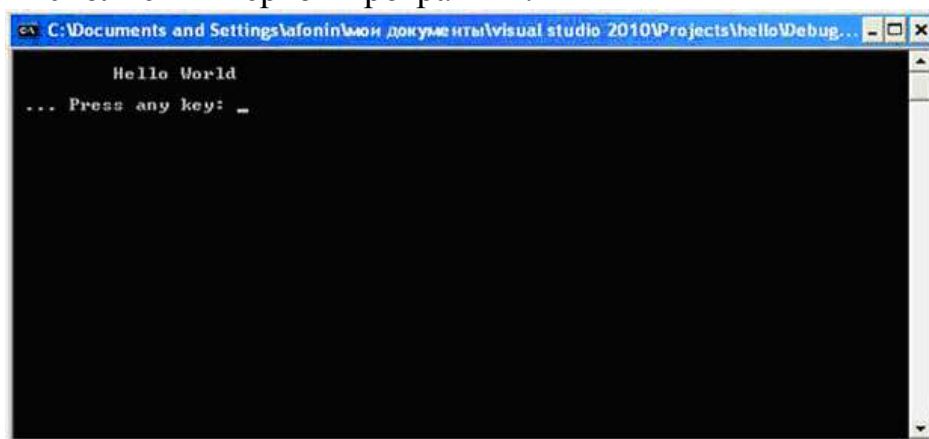


Рисунок 18 - Консольный вывод первой программы на языке C

Произведем разбор первой программы. Во-первых, надо отметить, что в языке C нет стандартных инструкций (операторов) для вывода сообщений на консоль (окно пользователя). В языке C предусматриваются специальные библиотечные файлы, в которых имеются функции для этих целей. В приведенной программе используется заголовочный файл с именем `stdio.h` (стандартный ввод-вывод), который должен быть включен в начало программы. Для вывода сообщения на консоль используется функция `printf()`. Для работы с консолью включен также заголовочный файл `conio.h`, который поддерживает функцию `_getch()`, которая извлекает символ из потока ввода, т. е. она предназначена для приема сообщения о нажатии какой-либо (почти любой) клавиши на клавиатуре. С другими компиляторами, возможно, потребуются `getch()`, т.е. без префиксного нижнего подчеркивания. Строка программы

```
int main (void)
```

сообщает системе, что именем программы является `main()` — главная функция, и что она возвращает целое число, о чем указывает аббревиатура "**int**". Имя `main()` — это специальное имя, которое указывает, где программа должна начать выполнение. Наличие круглых скобок после слова `main()` свидетельствует о том, что это имя функции. Если содержимое круглых скобок отсутствует или в них содержится служебное слово **void**, то это означает, что в функцию `main()` не передается никаких аргументов. Тело функции `main()` ограничено парой фигурных скобок. Все утверждения программы, заключенные в фигурные скобки, будут относиться к функции `main()`.

В теле функции `main()` имеются еще три функции. Во-первых, функции `printf()` находятся в библиотеке компилятора языка C, и они печатают или отображают те аргументы, которые были подставлены вместо параметров. Символ "`\n`" составляет единый символ newline (новая строка), т.е. с помощью этого символа осуществляется перевод на новую строку. Символ "`\t`" осуществляет табуляцию, т.е. начало вывода результатов программы с отступом вправо.

Функция без параметров `_getch()` извлекает символ из потока ввода (т.е. ожидает нажатия почти любой клавиши). С другими компиляторами, возможно, потребуется `getch()`, т.е. без префиксного нижнего подчеркивания.

Последнее утверждение в первой программе
`return 0;`

указывает на то, что выполнение функции `main()` закончено и что в систему возвращается значение 0 (целое число). Нуль используется в соответствии с соглашением об индикации успешного завершения программы.

В завершение следует отметить, что все действия в программе завершаются символом точки с запятой.

Все файлы проекта сохраняются в той папке, которая сформировалась после указания в поле Location имени проекта (hello). На рисунок 19 показаны папки и файлы проекта Visual Studio 2010.

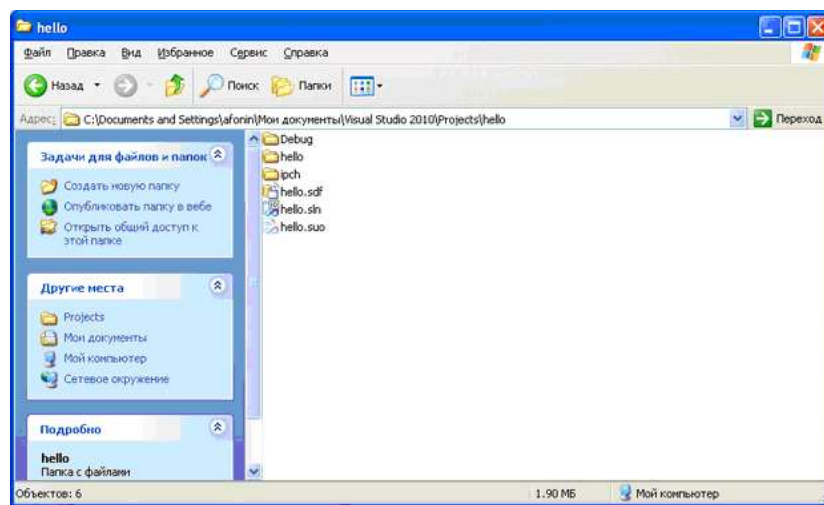


Рисунок 19 - Файлы и папки созданного проекта

На рисунке 19 файлы с полученными расширениями означают:

hello.sln – файл решения для созданной программы. Он содержит информацию о том, какие проекты входят в данное решение. Обычно, эти проекты расположены в отдельных подкаталогах. Например, наш проект находится в подкаталоге hello;

hello.suo – файл настроек среды Visual Studio при работе с решением, включает информацию об открытых окнах, их расположении и прочих пользовательских параметрах.

hello.sdf – файл содержащий вспомогательную информацию о проекте, который используется инструментами анализа кода Visual Studio, такими как IntelliSense для отображения подсказок об именах и т.д.

Файлы папки **Debug** показаны на рисунке 20.

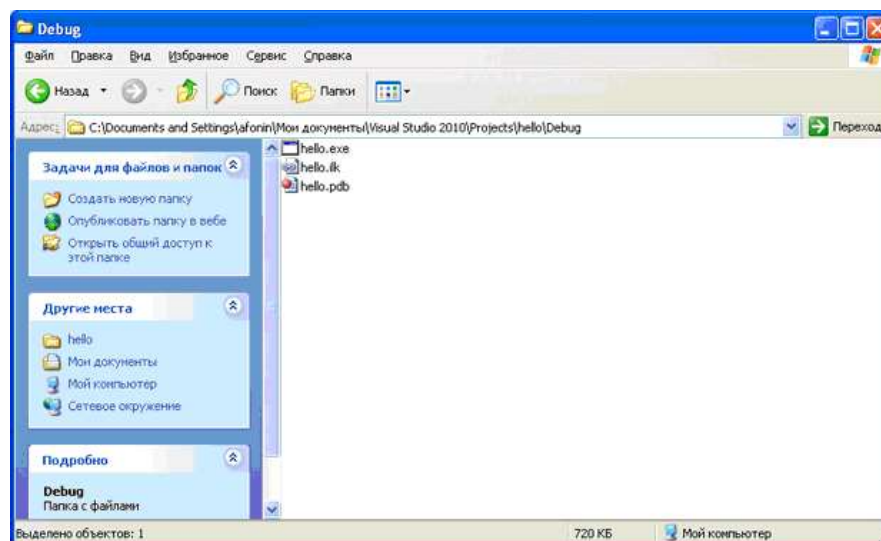


Рисунок 20 - Файлы папки Debug

Рассмотрим файлы в соответствии с рисунком 2.20.

hello.exe – исполняемый файл проекта;

hello.ilc – файл "incremental linker", используемый компоновщиком для ускорения процесса компоновки;

hello.pdb – отладочная информация/информация об именах в исполняемых файлах, используемая отладчиком.

Файлы папки **hello** показаны на рисунке 21.

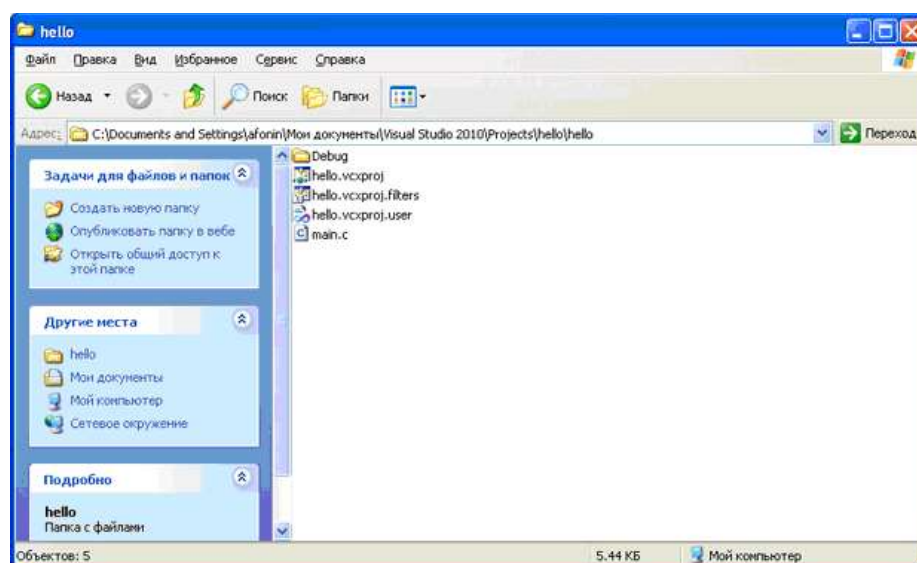


Рисунок 21 - Содержимое папки hello

Характеристика содержимого папки **hello**:

main.c – файл исходного программного кода,

hello.vcxproj – файл проекта,

hello.vcxproj.user – файл пользовательских настроек, связанных с проектом,

hello.vcxproj.filters – файл с описанием фильтров, используемых Visual

Studio Solution Explorer для организации и отображения файлов с исходным кодом.

ЗАДАНИЕ

В практической части выполните следующие задания на основе рассмотренной программы **hello**:

1. Напишите программу, которая выводила бы на консоль название факультета, где учитесь, номер группы, свою фамилию, имя и отчество в разных строках дисплея (консоли) с помощью одной функции `printf()`.

2. Вывод выполните с помощью нескольких функций `printf()` (количество функций должно соответствовать каждой порции информации).

3. Для задания пункта 2 вывод информации выполните в различных строках подряд, т.е. без межстрочного пропуска.

4. Проверьте программу без ключевого слова `void` для функции `main()`.

Примечание. Вывод требуемой информации осуществляется с помощью букв латинского алфавита. Комментарии в программе могут быть сделаны после символа `/*` или внутри комбинации символов `/* */`.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Какие компиляторы языка C вам известны?
2. Какое имя имеет исполняемый файл созданного проекта?
3. Объясните назначение заголовочных файлов `stdio.h`, `conio.h`.
4. Как будет работать программа без заголовочного файла `conio.h`?
5. В каком месте программы находится точка ее входа?
6. Как осуществляется табуляция строки на консоли и на сколько позиций выполняется отступ от левого края?
7. Какое значение имеет главная функция проекта `main()` в программах на языке C?

5 СОЗДАНИЕ ПРОЕКТА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Цель: Изучить процесс создания проекта параллельного программирования на основе технологии MPI.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Требования для использования отладчика MPI-кластера

На компьютере разработчика должна быть установлена среда Visual Studio 2010 Professional Edition или более высокой версии (с удаленным отладчиком).

У пользователя должны быть права администратора кластера.

У Visual Studio должен быть доступ ко всем вычислительным узлам, на которых предполагается запускать сеансы отладки. Необходимый доступ обеспечивают сценарии, указанные ниже.

Приложение разрабатывается на головном узле кластера или на выделенном узле входа.

Используется кластер, в котором вычислительные узлы подключены к корпоративной сети, а компьютер разработчика присоединен к тому же домену или к домену, для которого настроены отношения доверия с доменом кластера.

Для отправки приложения в кластер HPC с клиентского компьютера необходимо установить пакет Microsoft HPC Pack 2008.

Для построения MPI-программ с использованием интерфейса передачи сообщений корпорации Майкрософт необходимо, чтобы на компьютере разработчика был установлен пакет Windows HPC Server 2008 SDK.

Создание проекта MPI на C++ в Visual Studio 2010

Пример кода, приведенный в этом разделе, предназначен для создания параллельного приложения, вычисляющего значение числа пи с использованием моделирования методом Монте-Карло.

Этот пример кода выполняет 50 000 000 итераций в каждом процессе MPI. При каждой итерации генерируются случайные числа в интервале [0;1] для определения набора координат x и y . Набор координат оценивается для определения того, попадает ли точка в область, ограниченную линией с уравнением $x^2 + y^2 = 1$. Если точка попадает в эту область, значение переменной *count* увеличивается на единицу. Значения *count* из каждого процесса MPI суммируются, и результат присваивается переменной *result*. Общее количество точек, попавших в область (*result*), умножается на четыре и делится на общее количество итераций для получения приблизительного значения числа пи.

Приведенная ниже процедура содержит две реализации метода Монте-Карло.

В примере используются интерфейс MPI и библиотека PPL (Parallel

Patterns Library).

Создание образца проекта

1. Запустите Visual Studio 2010 (рисунок 1).

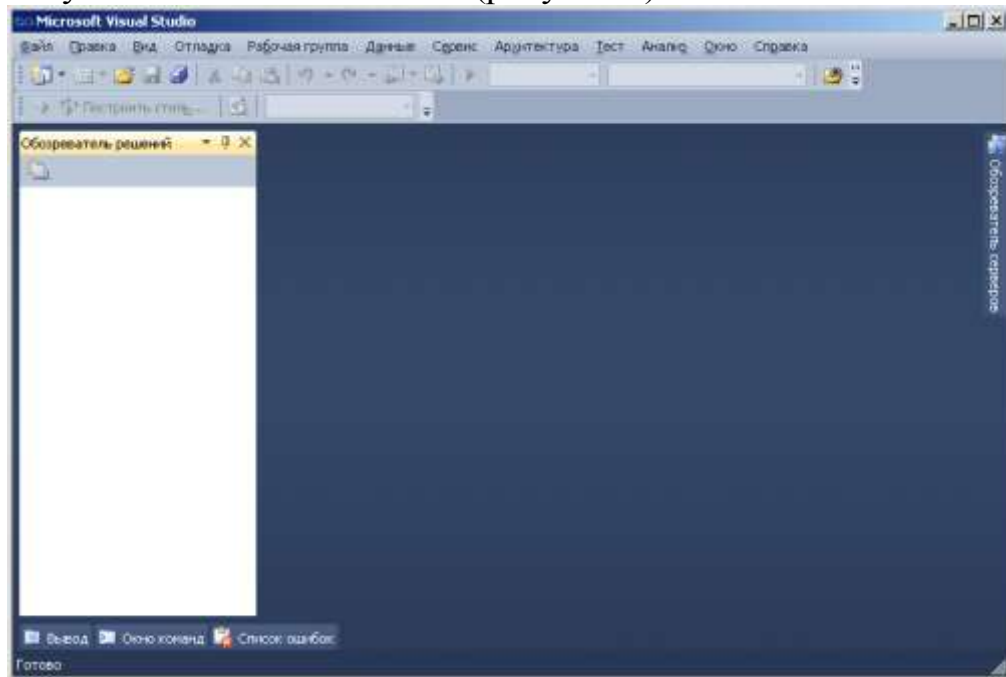


Рисунок 1. – Среда разработки приложений Visual Studio 2010

2. Создайте консольное приложение Win32 на C++ с именем **ParallelPI**. Используйте проект без предкомпилированных заголовков.
 - a. В меню **Файл** выберите пункт **Создать**, а затем — **Проект**.
 - b. В диалоговом окне **Новый проект** щелкните элемент **Установленные шаблоны** и выберите пункт **Visual C++**. (В зависимости от настроек Visual Studio пункт **Visual C++** может находиться под узлом **Другие языки**.)
 - c. В списке шаблонов щелкните элемент **Консольное приложение Win32**.
 - d. В качестве имени проекта введите **ParallelPI** (рисунок 2).

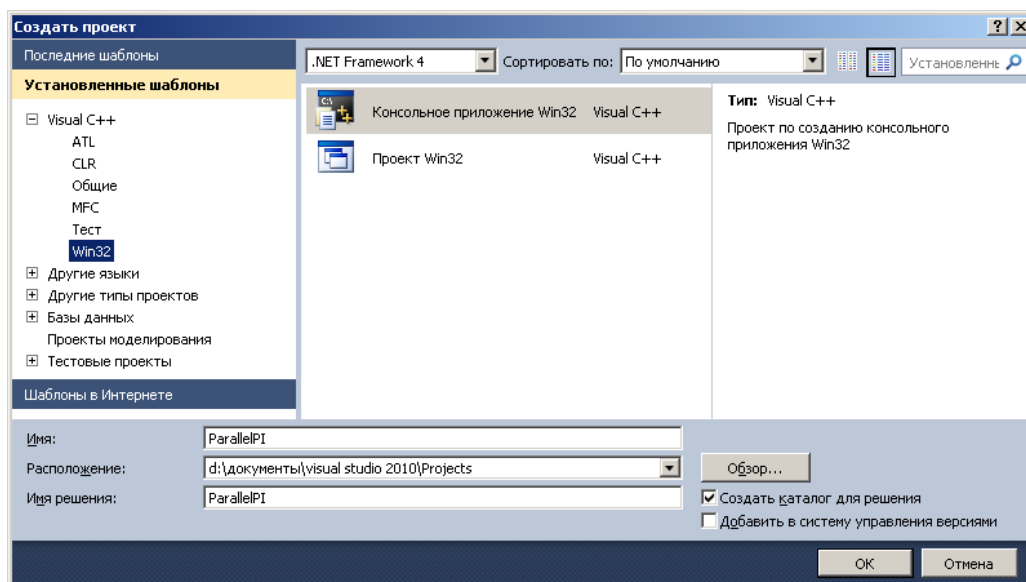


Рисунок 2. – Создание проекта

- е. Нажмите кнопку **ОК**. Откроется мастер создания консольного приложения Win32 (рисунок 3).

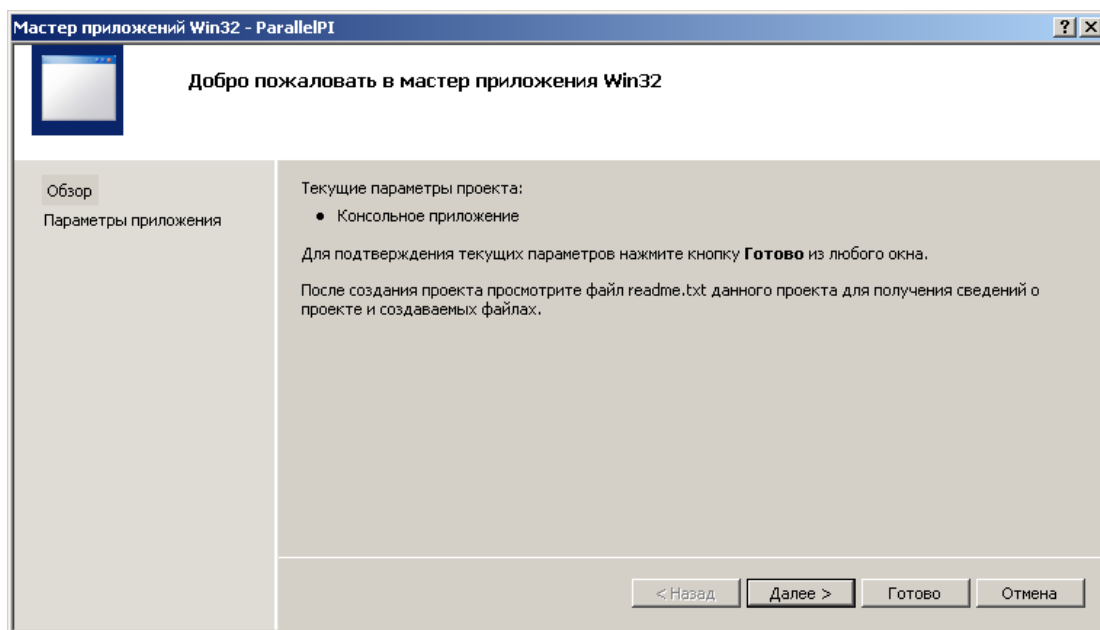


Рисунок 3. – Мастер создания проекта

- ф. Нажмите кнопку **Далее**.
- г. В разделе **Параметры приложения** в группе **Дополнительные параметры** снимите флажок **Предкомпилированный заголовок** (рисунок 4).

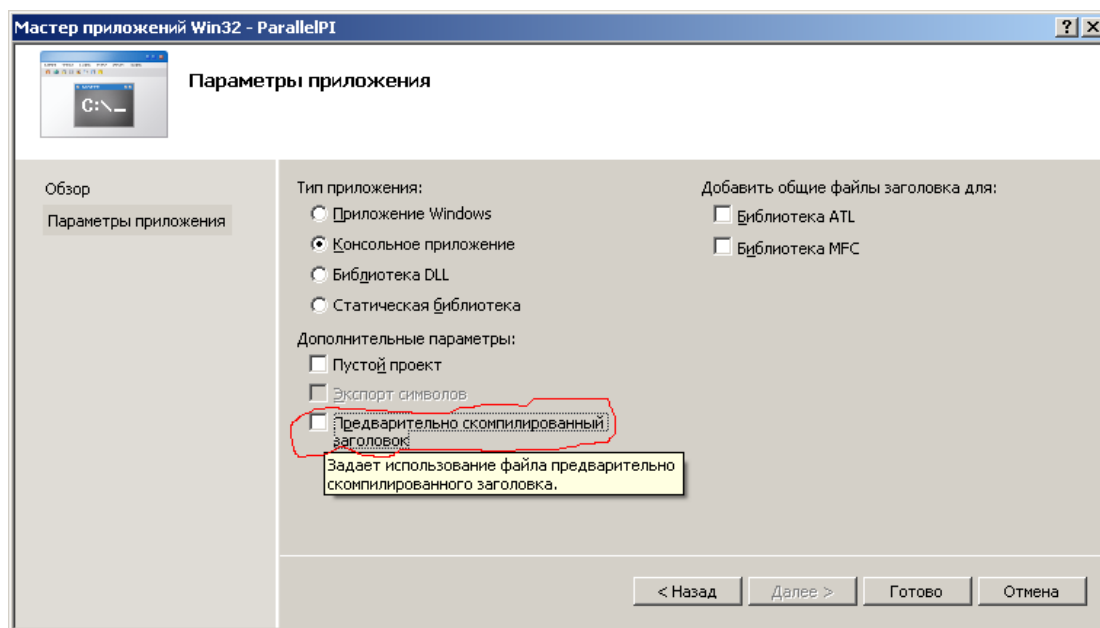


Рисунок 4. - Установка параметров приложения проекта

- h. Нажмите кнопку **Готово**, чтобы закрыть окно мастера и создать проект.
3. Задайте дополнительные свойства проекта.
 - a. В **обозревателе решений** щелкните правой кнопкой мыши элемент **Parallel PI** и выберите пункт **Свойства**. Откроется диалоговое окно **Страницы свойств**.
 - b. Разверните элемент **Свойства конфигурации** и выберите пункт **Каталоги VC++**. В области **Каталоги включения** поместите курсор в начало списка, который отображается в текстовом окне, и укажите местоположение файлов заголовков MS MPI C, после чего введите точку с запятой (;).
C:\Program Files\Microsoft HPC Pack 2008 SDK\Include; (рисунок 5).

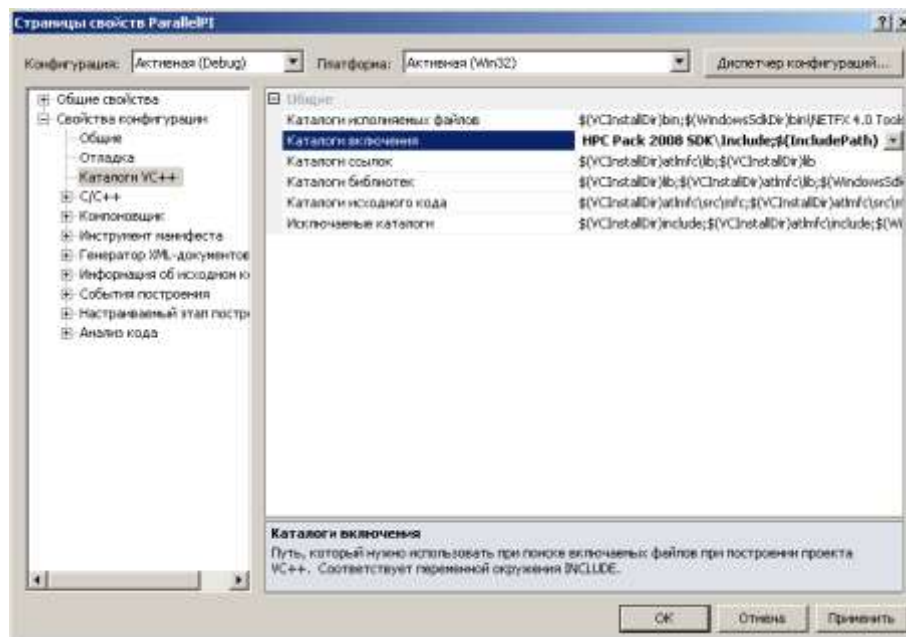


Рисунок 5. – Редактирование каталогов включения

- с. В области **Каталоги библиотек** поместите курсор в начало списка, который отображается в текстовом окне, и укажите местоположение файла библиотеки Microsoft HPC Pack 2008 SDK, после чего введите точку с запятой (;).
C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\i386;
 - d. В разделе **Компоновщик** выберите элемент **Ввод**. В разделе **Дополнительные зависимости** поместите курсор в начало списка, который отображается в текстовом окне, и введите следующее: `msmpi.lib`;
 - е. Нажмите кнопку **ОК**, чтобы закрыть страницы свойств.
4. В главном исходном файле выделите весь код и удалите его.
- Вставьте в пустой исходный файл приведенный ниже пример кода. В примере используются интерфейсы MPI.

```
// ParallelPI.cpp : Defines the entry point for the MPI application.
//
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "limits.h"
#include <ppl.h>
#include <random>
#include <time.h>
using namespace Concurrency;
int ThrowDarts(int iterations)
{
    combinable<int> count;
    int result = 0;
```

```

parallel_for(0, iterations, [&](int i){
    std::tr1::uniform_real<double> MyRandom;
    double RandMax = MyRandom.max();
    std::tr1::minstd_rand0 MyEngine;
    double x, y;
    MyEngine.seed((unsigned int)time(NULL));
    x = MyRandom(MyEngine)/RandMax;
    y = MyRandom(MyEngine)/RandMax;
    if(x*x + y*y < 1.0)
    {
        count.local() += 1;
    }
});
result = count.combine([](int left, int right) { return left + right; });
return result;
}
void main(int argc, char* argv[])
{
    int rank;
    int size;
    int iterations;
    int count;
    int result;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0)
    {
        //Rank 0 reads the number of iterations from the command line.
        //50M iterations is the default.
        iterations = 50000000;
        if(argc > 1)
        {
            iterations = atoi(argv[argc-1]);
        }
        printf("Executing %d iterations on %d nodes.\n", iterations, size);
        fflush(stdout);
    }
    //Broadcast the number of iterations to execute.
    MPI_Bcast(&iterations, 1, MPI_INT, 0, MPI_COMM_WORLD);
    count = ThrowDarts(iterations);
    //Gather and sum results
    MPI_Reduce(&count, &result, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

```

```

if(rank == 0)
{
    printf("The value of PI is approximated to be: %16f",
4*(((double)result/(double)(iterations*size)));
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
}

```

5. В меню **Файл** выберите пункт **Сохранить все**.

6. В меню **Построение** выберите пункт **Перестроить решение**.

Настройка и запуск отладчика MPI-кластера

После построения приложения можно настроить и запустить отладчик.

В этом разделе описаны три варианта отладки.

- Отладка одного процесса MPI на локальном компьютере,
- Отладка нескольких процессов MPI на локальном компьютере,
- Отладка одного или нескольких процессов MPI в кластере.

Отладка одного процесса MPI на локальном компьютере

Для отладки на локальном компьютере с применением только одного процесса MPI используйте тот же процесс, с помощью которого отлаживали бы любое другое приложение. Установите точку останова в необходимом месте программы и нажмите клавишу F5, чтобы запустить отладчик.

Программы MPI взаимодействуют по протоколу IP через порты. При первом запуске программы MPI может быть выведено предупреждение брандмауэра, уведомляющее об открытии порта. Ознакомьтесь с сообщением и разберитесь с тем, какие изменения будут внесены в систему. Для продолжения отладки на локальном компьютере необходимо разблокировать брандмауэр.

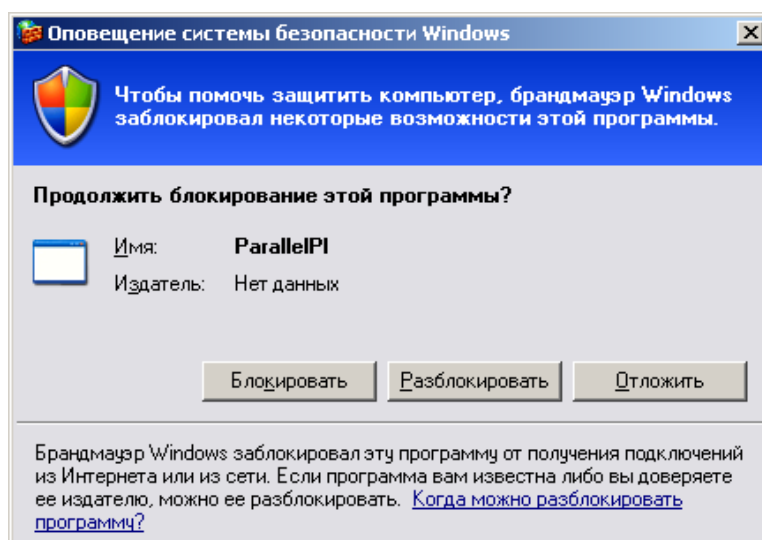


Рисунок 6. – Разблокировка брандмауэра.

Отладка нескольких процессов MPI на локальном компьютере

Ниже описано, как запустить локальный сеанс отладки для приложения **ParallelPI**.

Запуск отладчика MPI-кластера с четырьмя процессами MPI, запущенными на локальном компьютере

1. В обозревателе решений щелкните правой кнопкой мыши элемент **Parallel PI** и выберите пункт **Свойства**. Откроется диалоговое окно **Страницы свойств**.
2. Разверните узел **Свойства конфигурации** и выберите элемент **Отладка**.
3. В группе **Загружаемый отладчик** выберите элемент **Отладчик MPI-кластера**.
4. В раскрывающемся списке **Среда выполнения** выберите элемент **Правка узла Нрс**. Откроется диалоговое окно **Выбор узла**.
5. В раскрывающемся списке **Головной узел** выберите элемент **localhost**.
6. В поле **Количество процессов** выберите значение **4**.
7. Нажмите кнопку **ОК**, чтобы сохранить изменения и закрыть диалоговое окно **Выбор узла**.
8. Приложение **ParallelPI** принимает один аргумент, который определяет количество выполняемых итераций. По умолчанию выполняется 50 000 000 итераций. Для локального сеанса отладки уменьшите количество итераций до 5 000, выполнив указанное ниже действие.
9. В поле **Аргументы приложения** введите значение **5000**.
10. Нажмите кнопку **ОК**, чтобы сохранить изменения и закрыть окно **Страницы свойств**.
11. Установите точку останова в теле параллельного цикла `for`.
12. Нажмите клавишу F5, чтобы запустить отладчик.
13. Появятся пять окон консоли: одно окно **cmd.exe** и четыре окна **ParallelPI.exe** (по одному для каждого запущенного процесса). В окне консоли, соответствующем процессу с рангом 0, будут выведены количество итераций и вычисленное приблизительное значение числа пи.
14. В меню **Отладка** выберите пункт **Окна**, а затем — **Процессы**.
15. Выберите активный процесс для отладки, дважды щелкнув его в окне **Процессы**.

По умолчанию при отладке нескольких процессов точка останова влияет на все отлаживаемые процессы. Чтобы избежать прерывания процессов в непредвиденном месте, снимите флажок **Прерывать все процессы при прерывании одного**. (В меню **Сервис** выберите пункт **Настройки**, а затем — **Отладка**).

Отладка одного или нескольких процессов MPI в кластере

При запуске отладчика MPI в кластере отладчик отправляет приложение в кластер как задание. Среды выполнения Visual C, соответствующие проекту (x86 или x64, отладка или выпуск), должны находиться в рабочем каталоге на вычислительных узлах. Если необходимые среды выполнения отсутствуют на вычислительных узлах, необходимо включить их в развертывание отладчика, указав их с помощью свойства **Дополнительные файлы для развертывания**. Приведенная ниже процедура включает действие по развертыванию отладочной библиотеки DLL OpenMP времени выполнения. По умолчанию библиотека времени выполнения C (CRT) развертывается при запуске отладчика MPI-кластера. Если необходимые библиотеки времени выполнения недоступны, при попытке запустить приложение возникнут ошибки. Если библиотека времени выполнения OpenMP не подключена, точки останова не работают.

Запуск отладчика MPI в кластере:

1. В **обозревателе решений** щелкните правой кнопкой мыши элемент **Parallel PI** и выберите пункт **Свойства**. Откроется диалоговое окно **Страницы свойств**.
2. Разверните узел **Свойства конфигурации** и выберите элемент **Отладка**.
3. В группе **Загружаемый отладчик** выберите элемент **Отладчик MPI-кластера**.
4. В раскрывающемся списке **Среда выполнения** выберите элемент **Правка узла Нрс**. Откроется диалоговое окно **Выбор узла**.
5. В раскрывающемся списке **Головной узел** выберите имя головного узла кластера, который необходимо использовать.

Список головных узлов заполняется с контроллера домена Active Directory. Список содержит только кластеры, входящие в домен пользователя. Если нужный головной узел не отображается, введите имя или адрес IPv4 головного узла в поле свойства.

6. В поле **Количество процессов** выберите значение **4**.
7. В поле **Назначить процесс для** выберите способ выделения процессов. Можно выделить один процесс на **ядро**, **сокет** или **узел**.
8. Нажмите кнопку **ОК**, чтобы сохранить изменения и закрыть диалоговое окно **Выбор узла**.
9. В поле **Каталог развертывания** укажите общий каталог на головном узле. Если каталог развертывания не существует и у пользователя имеются права на запись в указанный корневой каталог, то каталог развертывания будет создан автоматически.

Общий каталог **CcpSpoolDir** создается при установке пакета HPC Pack 2008 на головном узле. Например, введите следующую строку, где *<myHeadNode>* — имя используемого кластера:

\\<myHeadNode>\CcpSpoolDir\

10. В поле **Рабочий каталог** укажите локальный рабочий каталог на каждом вычислительном узле. Например, введите следующую строку, где *<myUserName>* — ваше имя пользователя:

C:\Users\<myUserName>\ParallelPI

11. Если используется пример кода с интерфейсом OpenMP, добавьте файл отладочной DLL-библиотеки OpenMP (**Microsoft.VC100.DebugOpenMP\vcomp100d.dll**).

а. В группе **Дополнительные файлы для развертывания** выберите пункт **<Изменить файл...>**. Откроется диалоговое окно **Выбор файла и папки**.

б. Нажмите кнопку **Добавить файл**, перейдите к файлу **Microsoft.VC100.DebugOpenMP\vcomp100d.dll**, выберите его и нажмите кнопку **Открыть**.

Например, на компьютерах с архитектурой x86 и 64-разрядной версией системы Windows Server 2008 этот файл по умолчанию расположен по следующему пути:

C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\redist\Debug_NonRedist\x86\Microsoft.VC100.DebugOpenMP\vcomp100d.dll

с. Нажмите кнопку **ОК**, чтобы добавить файл и закрыть диалоговое окно **Выбор файла и папки**.

12. Нажмите кнопку **ОК**, чтобы сохранить изменения и закрыть окно **Страницы свойств**.

13. Установите точку останова в теле параллельного цикла `for`.

14. Нажмите клавишу F5, чтобы запустить отладчик.

15. Поскольку задание отправляется в кластер, отобразится окно ввода пароля для подключения к кластеру. Введите пароль и нажмите клавишу ВВОД.

16. После запуска отладчика проверьте в окне процессов расположение процессов. Узнайте для каждого процесса вычислительный узел, на котором запущен процесс. Эти сведения содержатся в столбце **Квалификатор транспорта**.

Приложение. Файлы, развертываемые Visual Studio в дополнение к двоичным файлам приложения (и CRT по запросу):

- DebuggerProxy.dll
- DebuggerProxy.dll.manifest
- Delete_from_workdir.bat: скрипт для удаления развернутых файлов
- Deploy_to_workdir.bat: скрипт для копирования файлов из каталога развертывания в рабочий каталог
- dbghelp.dll
- mcee.dll
- Mpishim.bat: скрипт для запуска удаленного отладчика

- Mpishim.exe: программа, организующая взаимодействие между интегрированной средой разработки и файлом Msvsmon.exe
- Msvsmon.exe: удаленный отладчик
- Msvsmon.exe.config
- PfxTaskProvider.dll
- symsrv.dll
- symsrv.yes
- vbdebug.dll
- 1049\msdbgui.dll
- 1049\vbdebugui.dll

ЗАДАНИЕ

В практической части наберите и отладите программу, приведенную в теоретической части. Откомпилируйте и запустите программу на выполнение.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Какие технологии параллельного программирования Вам известны?
2. Какой стандарт поддерживает технология параллельного программирования на основе передачи сообщений?
3. Каковы структура параллельной программы на основе MPI?

6 ПРОГРАММИРОВАНИЕ НА MPI

Цель: Изучить возможности библиотеки MPI для создания параллельных программ.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В вычислительных системах с распределенной памятью процессоры работают независимо друг от друга. Для организации параллельных вычислений в таких условиях необходимо иметь возможность распределять вычислительную нагрузку и организовать информационное взаимодействие (передачу данных) между процессорами.

Решение всех перечисленных вопросов и обеспечивает интерфейс передачи данных (message passing interface – MPI).

В общем плане, для распределения вычислений между процессорами необходимо проанализировать алгоритм решения задачи, выделить информационно независимые фрагменты вычислений, провести их программную реализацию и затем разместить полученные части программы на разных процессорах. В рамках MPI принят более простой подход – для решения поставленной задачи разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах. При этом для того чтобы избежать идентичности вычислений на разных процессорах, можно, во-первых, подставлять разные данные для программы на разных процессорах, а во-вторых, использовать имеющиеся в MPI средства для идентификации процессора, на котором выполняется программа (тем самым предоставляется возможность организовать различия в вычислениях в зависимости от используемого программой процессора).

Подобный способ организации параллельных вычислений получил наименование модели "одна программа множество процессов" (single program multiple processes or SPMP).

Для организации информационного взаимодействия между процессорами в самом минимальном варианте достаточно операций приема и передачи данных (при этом, конечно, должна существовать техническая возможность коммуникации между процессорами – каналы или линии связи). В MPI существует целое множество операций передачи данных. Они обеспечивают разные способы пересылки данных - коммуникационные операции. Именно данные возможности являются наиболее сильной стороной MPI (об этом, в частности, свидетельствует и само название MPI).

Следует отметить, что попытки создания программных средств передачи данных между процессорами начали предприниматься практически сразу с появлением локальных компьютерных сетей. Однако подобные средства часто были неполными и, самое главное, являлись несовместимыми. Таким образом, одна из самых серьезных проблем в программировании – переносимость программ при переводе программного обеспечения на другие компьютерные системы – проявлялась при разработке параллельных программ в

максимальной степени. Как результат, уже с 90-х годов стали предприниматься усилия по стандартизации средств организации передачи сообщений в многопроцессорных вычислительных системах. Началом работ, непосредственно приведших к появлению MPI, послужило проведение рабочего совещания по стандартам для передачи сообщений в среде распределенной памяти (the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992). По итогам совещания была образована рабочая группа, позднее преобразованная в международное сообщество MPI Forum, результатом деятельности которых явилось создание и принятие в 1994 г. стандарта интерфейса передачи сообщений (message passing interface – MPI) версии 1.0. В последующие годы стандарт MPI последовательно развивался. В 1997 г. был принят стандарт MPI версии 2.0.

Итак, теперь можно пояснить, что означает понятие MPI. Во-первых, MPI – это стандарт, которому должны удовлетворять средства организации передачи сообщений. Во-вторых, MPI – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI. Так, по стандарту, эти программные средства должны быть организованы в виде библиотек программных функций (библиотеки MPI) и должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran. Подобную "двойственность" MPI следует учитывать при использовании терминологии. Как правило, аббревиатура MPI применяется при упоминании стандарта, а сочетание "библиотека MPI " указывает на ту или иную программную реализацию стандарта. Однако достаточно часто для краткости обозначение MPI используется и для библиотек MPI, и, тем самым, для правильной интерпретации термина следует учитывать контекст.

Приведем ряд его важных положительных моментов технологии MPI:

- MPI позволяет в значительной степени снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами – параллельная программа, разработанная на алгоритмическом языке C или Fortran с использованием библиотеки MPI, как правило, будет работать на разных вычислительных платформах;
- MPI содействует повышению эффективности параллельных вычислений, поскольку в настоящее время практически для каждого типа вычислительных систем существуют реализации библиотек MPI, в максимальной степени учитывающие возможности компьютерного оборудования;
- MPI уменьшает, в определенном плане, сложность разработки параллельных программ, т. к., с одной стороны, большая часть основных операций передачи данных предусматривается стандартом MPI, а с другой стороны, уже имеется большое количество библиотек параллельных методов, созданных с использованием MPI.

ЗАДАНИЕ

1. Разработайте программу для нахождения минимального (максимального) значения среди элементов вектора.
2. Разработайте программу для вычисления скалярного произведения двух векторов.
3. Разработайте программу, в которой два процесса многократно обмениваются сообщениями длиной n байт. Выполните эксперименты и оцените зависимость времени выполнения операции передачи данных от длины сообщения. Сравните с теоретическими оценками, построенными по модели Хокни.
4. Подготовьте варианты ранее разработанных программ с разными режимами выполнения операций передачи данных. Сравните время выполнения операций передачи данных при разных режимах работы.
5. Подготовьте варианты ранее разработанных программ с использованием неблокирующего способа выполнения операций передачи данных. Оцените количество вычислительных операций, необходимое для того, чтобы полностью совместить передачу данных и вычисления. Разработайте программу, в которой бы полностью отсутствовали задержки вычислений из-за ожидания передаваемых данных.
6. Выполните задание 3 с использованием операции одновременного выполнения передачи и приема данных. Сравните результаты вычислительных экспериментов.
7. Разработайте программу-пример для каждой имеющейся в MPI коллективной операции.
8. Разработайте реализации коллективных операций при помощи парных обменов между процессами. Выполните вычислительные эксперименты и сравните время выполнения разработанных программ и функций MPI для коллективных операций.
9. Разработайте программу, выполните эксперименты и сравните результаты для разных алгоритмов реализации операции сбора, обработки и рассылки данных всем процессам (функция `MPI_Allreduce`).
10. Разработайте программу-пример для каждого имеющегося в MPI способа конструирования производных типов данных.
11. Разработайте программу-пример с использованием функций упаковки и распаковки данных. Выполните эксперименты и сравните с результатами при использовании производных типов данных.
12. Разработайте производные типы данных для строк, столбцов, диагоналей матриц.
13. Разработайте программу-пример для каждой из рассмотренных функций для управления процессами и коммутаторами.
14. Разработайте программу для представления множества процессов в виде прямоугольной решетки. Создайте коммутаторы для каждой строки и столбца процессов. Выполните коллективную операцию для

всех процессов и для одного из созданных коммуникаторов. Сравните время выполнения операции.

15. Изучите самостоятельно и разработайте программы-примеры для передачи данных между процессами разных коммуникаторов.
16. Разработайте программу-пример для декартовой топологии.
17. Разработайте программу-пример для топологии графа.
18. Разработайте подпрограммы для создания некоторого набора дополнительных виртуальных топологий (звезда, дерево и др.).

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Какой минимальный набор средств является достаточным для организации параллельных вычислений в системах с распределенной памятью?
2. В чем состоит важность стандартизации средств передачи сообщений?
3. Что следует понимать под параллельной программой?
4. В чем различие понятий процесса и процессора?
5. Какой минимальный набор функций MPI позволяет начать разработку параллельных программ?
6. Как описываются передаваемые сообщения?
7. Как можно организовать прием сообщений от конкретных процессов?
8. Как определить время выполнения MPI -программы?
9. В чем различие парных и коллективных операций передачи данных?
10. Какая функция MPI обеспечивает передачу данных от одного процесса всем процессам?
11. Что понимается под операцией редукции?
12. В каких ситуациях следует применять барьерную синхронизацию?
13. Какие режимы передачи данных поддерживаются в MPI?
14. Как организуется неблокирующий обмен данными в MPI?
15. В чем состоит понятие тупика? Когда функция одновременного выполнения передачи и приема гарантирует отсутствие тупиковых ситуаций?
16. Какие коллективные операции передачи данных предусмотрены в MPI?
17. Что понимается под производным типом данных в MPI?
18. Какие способы конструирования типов имеются в MPI?
19. В каких ситуациях может быть полезна упаковка и распаковка данных?
20. Что понимается в MPI под коммуникатором?
21. Для чего может потребоваться создание новых коммуникаторов?
22. Что понимается в MPI под виртуальной топологией?
23. Какие виды топологий предусмотрены в MPI?
24. Для чего может оказаться полезным использование виртуальных топологий?
25. Какие основные дополнительные возможности предусмотрены в стандарте MPI -2?

7 ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ УМНОЖЕНИЯ МАТРИЦЫ НА ВЕКТОР

Цель: разработать программу реализующую параллельный алгоритм векторно-матричного умножения.

В лабораторной работе рассматривается задача умножения матрицы на вектор. Приводится постановка задачи и последовательный алгоритм ее решения. Описываются методы разделения матрицы между процессорами вычислительной системы, которые необходимы для параллельной реализации матричных операций. Далее излагаются три возможных подхода к параллельной реализации алгоритма умножения матрицы на вектор. Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

С учетом значимости эффективного выполнения матричных расчетов многие стандартные библиотеки программ содержат процедуры для различных матричных операций. Объем программного обеспечения для обработки матриц постоянно увеличивается – разрабатываются новые экономные структуры хранения для матриц специального типа (треугольных, ленточных, разреженных и т.п.), создаются различные высокоэффективные машинно-зависимые реализации алгоритмов, проводятся теоретические исследования для поиска более быстрых методов матричных вычислений.

Являясь вычислительно трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

Последовательный алгоритм умножения матрицы на вектор может быть представлен следующим образом.

Алгоритм 1. Последовательный алгоритм умножения матрицы на вектор

```
// Алгоритм 1
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++){
    c[i] = 0;
    for (j = 0; j < n; j++){
        c[i] += A[i][j]*b[j]
    }
}
```

}

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины n требует выполнения n операций умножения и $n-1$ операций сложения, его трудоемкость порядка $O(n)$. Для выполнения матрично-векторного умножения необходимо осуществить m операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка $O(mn)$.

Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по строкам. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы параллельных вычислений.

Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 2
// Умножение матрицы на вектор – ленточное горизонтальное разбиение
// (исходный и результирующий векторы дублируются между процессами)
void main(int argc, char* argv[]) {
double* pMatrix; // Первый аргумент – исходная матрица
double* pVector; // Второй аргумент – исходный вектор
double* pResult; // Результат умножения матрицы на вектор
int Size; // Размеры исходных матрицы и вектора
double* pProcRows;
double* pProcResult;
int RowNum;
double Start, Finish, Duration;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
// Выделение памяти и инициализация исходных данных
ProcessInitialization(pMatrix,pVector,pResult,pProcRows,pProcResult,Size,RowNum);
// Распределение исходных данных между процессами
DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);
// Параллельное выполнение умножения матрицы на вектор
ParallelResultCalculation(pProcRows, pVector, pProcResult, Size, RowNum);
// Сбор результирующего вектора на всех процессах
ResultReplication(pProcResult, pResult, Size, RowNum);
// Завершение процесса вычислений
ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);
MPI_Finalize();
}
```

Функция ProcessInitialization. Эта функция задает размер и элементы для матрицы A и вектора b. Значения для матрицы A и вектора b определяются в функции RandomDataInitialization.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pMatrix, double* &pVector, double*
&pResult, double* &pProcRows, double* &pProcResult, int &Size, int &RowNum)
{ int RestRows; // Количество строк матрицы, которые еще не распределены
  int i;
  if (ProcRank == 0) { do {
    printf("\nВведите размер матрицы: ");
    scanf("%d", &Size);
    if (Size < ProcNum) {
      printf("Размер матрицы должен превышать количество
процессов! \n "); } }
    while (Size < ProcNum); }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-1/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);
  pVector = new double [Size];
  pResult = new double [Size];
  pProcRows = new double [RowNum*Size];
  pProcResult = new double [RowNum];
  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    RandomDataInitialization(pMatrix, pVector, Size); } }
```

Функция DataDistribution. Осуществляет рассылку вектора b и распределение строк исходной матрицы A по процессам вычислительной системы. Следует отметить, что когда количество строк матрицы n не является кратным числу процессоров p, объем пересылаемых данных для процессов может оказаться разным и для передачи сообщений необходимо использовать функцию MPI_Scatterv библиотеки MPI.

```
// Функция для распределения исходных данных между процессами
void DataDistribution(double* pMatrix, double* pProcRows,
double* pVector, int Size, int RowNum) {
  int *pSendNum; // Количество элементов, посылаемых процессу
  int *pSendInd; // Индекс первого элемента данных,
// посылаемого процессу
  int RestRows=Size; // Количество строк матрицы, которые еще
// не распределены
```

```

MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Выделение памяти для хранения временных объектов
pSendInd = new int [ProcNum];
pSendNum = new int [ProcNum];
// Определение положения строк матрицы, предназначенных
// каждому процессу
RowNum = (Size/ProcNum);
pSendNum[0] = RowNum*Size;
pSendInd[0] = 0;
for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
}
// Рассылка строк матрицы
MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);
// Освобождение памяти
delete [] pSendNum;
delete [] pSendInd;
}

```

Следует отметить, что такое разделение действий генерации исходных данных и их рассылки между процессами может быть неоправданным в реальных параллельных вычислениях при большом объеме данных. Широко используемый подход в таких случаях состоит в организации передачи сообщений процессам сразу же после того, как данные процессов будут сгенерированы. Снижение затрат памяти для хранения данных может быть достигнуто также и за счет организации генерации данных в последнем процессе (при таком подходе память для пересылаемых данных и для данных процесса может быть одной и той же).

Функция ParallelResultCalculation. Данная функция производит умножение на вектор тех строк матрицы, которые распределены на данный процесс, и таким образом получается блок результирующего вектора с.

```

// Функция для вычисления части результирующего вектора
void ParallelResultCalculation(double* pProcRows, double* pVector,
    double* pProcResult, int Size, int RowNum) {
    int i, j;
    for (i=0; i<RowNum; i++) {
        pProcResult[i] = 0;
        for (j=0; j<Size; j++)
            pProcResult[i] += pProcRows[i*Size+j]*pVector[j]; }
}

```

```
}
```

Функция ResultReplication. Объединяет блоки результирующего вектора *s*, полученные на разных процессах, и копирует вектор результата на все процессы вычислительной системы.

// Функция для сбора результирующего вектора на всех процессах

```
void ResultReplication(double* pProcResult, double* pResult,
int Size, int RowNum) {
    int *pReceiveNum; // Количество элементов, посылаемых процессом
    int *pReceiveInd; // Индекс элемента данных в результирующем векторе
    int RestRows=Size;//Кол-во строк матрицы, которые еще не распределены
    int i;
    // Выделение памяти для временных объектов
    pReceiveNum = new int [ProcNum];
    pReceiveInd = new int [ProcNum];
    // Определение положения блоков результирующего вектора
    pReceiveInd[0] = 0;
    pReceiveNum[0] = Size/ProcNum;
    for (i=1; i<ProcNum; i++) {
        RestRows -= pReceiveNum[i-1];
        pReceiveNum[i] = RestRows/(ProcNum-i);
        pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
    }
    // Сбор всего результирующего вектора на всех процессах
    MPI_Allgather(pProcResult, pReceiveNum[ProcRank],
        MPI_DOUBLE, pResult, pReceiveNum, pReceiveInd,
        MPI_DOUBLE, MPI_COMM_WORLD);
    // Освобождение памяти
    delete [] pReceiveNum;
    delete [] pReceiveInd;
}
```

ЗАДАНИЕ

1. Выполните реализацию параллельного алгоритма, основанного на ленточном разбиении матрицы на вертикальные полосы. Постройте оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.
2. Выполните реализацию параллельного алгоритма, основанного на разбиении матрицы на блоки. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими

оценками.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назовите основные способы распределения элементов матрицы между процессорами вычислительной системы.
2. В чем состоит постановка задачи умножения матрицы на вектор?
3. Какова вычислительная сложность последовательного алгоритма умножения матрицы на вектор?
4. Почему при разработке параллельных алгоритмов умножения матрицы на вектор допустимо дублировать вектор-операнд на все процессоры?
5. Какие подходы могут быть предложены для разработки параллельных алгоритмов умножения матрицы на вектор?
6. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матрицы на вектор.
7. Проведите анализ и получите показатели эффективности для одного из рассмотренных алгоритмов.
8. Какой из представленных алгоритмов умножения матрицы на вектор обладает лучшими показателями ускорения и эффективности?
9. Может ли использование циклической схемы разделения данных повлиять на время работы каждого из представленных алгоритмов?
10. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных? В чем различие необходимых операций передачи данных при разделении матрицы по строкам и столбцам?
11. Какие информационные взаимодействия выполняются для блочного алгоритма умножения матрицы на вектор?
12. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
13. Дайте общую характеристику программной реализации алгоритма умножения матрицы на вектор при разделении данных по строкам. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?
14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

8 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ МЕТОДЫ МАТРИЧНОГО УМНОЖЕНИЯ

Цель: разработать программу реализующую параллельный алгоритм матричного умножения.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

Алгоритм 1. Последовательный алгоритм умножения двух квадратных матриц

```
// Алгоритм 1
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
for (i=0; i<Size; i++){
    for (j=0; j<Size; j++){
        MatrixC[i][j] = 0;
        for (k=0; k<Size; k++){
            MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j]; }
    }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы С.

Программная реализация параллельных алгоритмов матричного умножения

Представим возможный вариант программной реализации алгоритма Фокса для умножения матриц при блочном представлении данных. Приводимый программный код содержит основные модули параллельной программы, отсутствие отдельных вспомогательных функций не сказывается на общем понимании реализуемой схемы параллельных вычислений.

Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.1
// Алгоритм Фокса умножения матриц – блочное представление данных
// Условия выполнения программы: все матрицы квадратные,
// размер блоков и их количество по горизонтали и вертикали
// одинаково, процессы образуют квадратную решетку
int ProcNum = 0; // Количество доступных процессов
int ProcRank = 0; // Ранг текущего процесса
```

```

int GridSize;    // Размер виртуальной решетки процессов
int GridCoords[2]; // Координаты текущего процесса в процессной
// решетке
MPI_Comm GridComm; // Коммуникатор в виде квадратной решетки
MPI_Comm ColComm; // коммуникатор – столбец решетки
MPI_Comm RowComm; // коммуникатор – строка решетки
void main ( int argc, char * argv[] ) {
    double* pAMatrix;    // Первый аргумент матричного умножения
    double* pBMatrix;    // Второй аргумент матричного умножения
    double* pCMatrix;    // Результирующая матрица
    int Size;            // Размер матриц
    int BlockSize; // Размер матричных блоков, расположенных
                    // на процессах
    double *pAblock;    // Блок матрицы A на процессе
    double *pBblock;    // Блок матрицы B на процессе
    double *pCblock;    // Блок результирующей матрицы C на процессе
    double *pMatrixAblock;
    double Start, Finish, Duration;
    setvbuf(stdout, 0, _IONBF, 0);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize) {
        if (ProcRank == 0) {
            printf ("Number of processes must be a perfect square \n");
        }
    }
    else {
        if (ProcRank == 0)
            printf("Parallel matrix multiplication program\n");
        // Создание виртуальной решетки процессов и коммуникаторов
        // строк и столбцов
        CreateGridCommunicators();
        // Выделение памяти и инициализация элементов матриц
        ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock,
            pBblock, pCblock, pMatrixAblock, Size, BlockSize );
        // Блочное распределение матриц между процессами
        DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
            BlockSize);
        // Выполнение параллельного метода Фокса
    }
}

```

```

ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
    pCblock, BlockSize);
// Сбор результирующей матрицы на ведущем процессе
ResultCollection(pCMatrix, pCblock, Size, BlockSize);
// Завершение процесса вычислений
ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
    pCblock, pMatrixAblock);
}
MPI_Finalize();
}

```

Функция CreateGridCommunicators. Данная функция создает коммунитор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке, а также создает коммуниторы отдельно для каждой строки и каждого столбца.

Создание решетки производится при помощи функции MPI_Cart_create (вектор Periodic определяет возможность передачи сообщений между граничными процессами строк и столбцов создаваемой решетки). После создания решетки каждый процесс параллельной программы будет иметь координаты своего положения в решетке; получение этих координат обеспечивается при помощи функции MPI_Cart_coords.

Формирование топологий завершается созданием множества коммуниторов для каждой строки и каждого столбца решетки в отдельности (функция MPI_Cart_sub).

```

// Создание коммунитора в виде двумерной квадратной решетки
// и коммуниторов для каждой строки и каждого столбца решетки
void CreateGridCommunicators() {
    int DimSize[2]; // Количество процессов в каждом измерении решетки
    int Periodic[2]; // =1 для каждого измерения, являющегося периодическим
    int Subdims[2]; // =1 для каждого измерения, оставляемого в подрешетке
    DimSize[0] = GridSize;
    DimSize[1] = GridSize;
    Periodic[0] = 0;
    Periodic[1] = 0;
    // Создание коммунитора в виде квадратной решетки
    MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);
    // Определение координат процесса в решетке
    MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);
    // Создание коммуниторов для строк процессной решетки
    Subdims[0] = 0; // Фиксация измерения
    Subdims[1] = 1; // Наличие данного измерения в подрешетке
    MPI_Cart_sub(GridComm, Subdims, &RowComm);
    // Создание коммуниторов для столбцов процессной решетки

```

```

Subdims[0] = 1;
Subdims[1] = 0;
MPI_Cart_sub(GridComm, Subdims, &ColComm);
}

```

Функция ProcessInitialization. Данная функция определяет параметры решаемой задачи (размеры матриц и их блоков), выделяет память для хранения данных и осуществляет ввод исходных матриц (или формирует их при помощи какого-либо датчика случайных чисел). Всего в каждом процессе должна быть выделена память для хранения четырех блоков – для указателей на выделенную память используются переменные pAblock, pBblock, pCblock, pMatrixAblock. Первые три указателя определяют блоки матриц A, B и C соответственно. Следует отметить, что содержимое блоков pAblock и pBblock постоянно меняется в соответствии с пересылкой данных между процессами, в то время как блок pMatrixAblock матрицы A остается неизменным и применяется при рассылках блоков по строкам решетки процессов (см. функцию AblockCommunication).

Для определения элементов исходных матриц будем использовать функцию RandomDataInitialization, реализацию которой читателю предстоит выполнить самостоятельно.

// Функция для выделения памяти и инициализации исходных данных

```

void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
double* &pCMatrix, double* &pAblock, double* &pBblock,
double* &pCblock, double* &pTemporaryAblock, int &Size,
int &BlockSize ) {
    if (ProcRank == 0) {
        do {
            printf("\nВведите размер матриц: ");
            scanf("%d", &Size);
            if (Size%GridSize != 0) {
                printf ("Размер матриц должен быть кратен размеру сетки! \n");
            }
        }
        while (Size%GridSize != 0);
    }
    MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    BlockSize = Size/GridSize;
    pAblock = new double [BlockSize*BlockSize];
    pBblock = new double [BlockSize*BlockSize];
    pCblock = new double [BlockSize*BlockSize];
    pTemporaryAblock = new double [BlockSize*BlockSize];
    for (int i=0; i<BlockSize*BlockSize; i++) {
        pCblock[i] = 0;
    }
}

```

```

}
if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    RandomDataInitialization(pAMatrix, pBMatrix, Size); }
}

```

Функции DataDistribution и ResultCollection. После задания исходных матриц на нулевом процессе необходимо осуществить распределение исходных данных. Для этого предназначена функция DataDistribution. Может быть предложено два способа выполнения блочного разделения матриц между процессорами, организованными в двумерную квадратную решетку. В первом из них для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средствами MPI производный тип данных. Во втором способе можно организовать двухэтапную процедуру. На первом этапе матрица разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки. Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки.

Для выполнения сбора результирующей матрицы из блоков предназначена функция ResultCollection. Сбор данных также можно выполнить либо с использованием производного типа данных, либо при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы.

Реализация функций DataDistribution и ResultCollection представляет собой задание для самостоятельной работы.

Функция ABlockCommunication. Функция выполняет рассылку блоков матрицы A по строкам процессорной решетки. Для этого в каждой строке решетки определяется ведущий процесс Pivot, осуществляющий рассылку. Для рассылки используется блок pMatrixAblock, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции MPI_Bcast. Следует отметить, что данная операция является коллективной и ее локализация пределами отдельных строк решетки обеспечивается за счет использования коммуникаторов RowComm, определенных для набора процессов каждой строки решетки в отдельности.

```

// Рассылка блоков матрицы A по строкам решетки процессов
void ABlockCommunication (int iter, double *pAblock,
    double* pMatrixAblock, int BlockSize) {
    // Определение ведущего процесса в строке процессной решетки
    int Pivot = (GridCoords[0] + iter) % GridSize;
    // Копирование передаваемого блока в отдельный буфер памяти
    if (GridCoords[1] == Pivot) {
        for (int i=0; i<BlockSize*BlockSize; i++)

```

```

    pAblock[i] = pMatrixAblock[i];
}
// Рассылка блока
MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot,
RowComm);
}

```

Функция BlockMultiplication. Функция обеспечивает перемножение блоков матриц А и В. Следует отметить, что для более легкого понимания рассматриваемой программы приводится простой вариант реализации функции – выполнение операции блочного умножения может быть существенным образом оптимизировано для сокращения времени вычислений. Данная оптимизация может быть направлена, например, на повышение эффективности использования кэша процессоров, векторизации выполняемых операций и т.п.

```

// Умножение матричных блоков
void BlockMultiplication (double *pAblock, double *pBblock,
double *pCblock, int BlockSize) {
// Вычисление произведения матричных блоков
for (int i=0; i<BlockSize; i++) {
for (int j=0; j<BlockSize; j++) {
double temp = 0;
for (int k=0; k<BlockSize; k++ )
temp += pAblock [i*BlockSize + k] * pBblock [k*BlockSize + j];
pCblock [i*BlockSize + j] += temp; }
}
}

```

Функция BblockCommunication. Функция выполняет циклический сдвиг блоков матрицы В по столбцам процессорной решетки. Каждый процесс передает свой блок следующему процессу NextProc в столбце процессов и получает блок, переданный из предыдущего процесса PrevProc в столбце решетки. Выполнение операций передачи данных осуществляется при помощи функции MPI_SendRecv_replace, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти pBblock. Кроме того, эта функция гарантирует отсутствие возможных тупиков, когда операции передачи данных начинают одновременно выполняться несколькими процессами при кольцевой топологии сети.

```

// Циклический сдвиг блоков матрицы В вдоль столбца процессной
// решетки
void BblockCommunication (double *pBblock, int BlockSize) {
MPI_Status Status;
int NextProc = GridCoords[0] + 1;
if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
int PrevProc = GridCoords[0] - 1;

```

```

if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;
MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColComm, &Status);
}

```

Функция ParallelResultCalculation. Для непосредственного выполнения параллельного алгоритма Фокса умножения матриц предназначена функция ParallelResultCalculation, которая реализует логику работы алгоритма.

// Функция для параллельного умножения матриц

```

void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,
    double* pBblock, double* pCblock, int BlockSize) {
    for (int iter = 0; iter < GridSize; iter++) {
        // Рассылка блоков матрицы A по строкам процессной решетки
        ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);
        // Умножение блоков
        BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
        // Циклический сдвиг блоков матрицы B в столбцах процессной
        // решетки
        BblockCommunication(pBblock, BlockSize); }
}

```

ЗАДАНИЕ

1. Выполните реализацию двух ленточных алгоритмов умножения матриц. Сравните время выполнения этих алгоритмов.
2. Выполните реализацию алгоритма Кэннона. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее полученными теоретическими оценками.
3. Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных процессорных решеток общего вида.

Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит постановка задачи умножения матриц?
2. Приведите примеры задач, в которых используется операция умножения матриц.
3. Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
4. Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?

5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матриц.
6. Проведите анализ и получите показатели эффективности ленточного алгоритма при горизонтальном разбиении перемножаемых матриц.
7. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
8. Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?
9. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
10. Какой из рассмотренных алгоритмов характеризуется наименьшими и наибольшими требованиями к объему необходимой памяти?
11. Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
12. Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
13. Дайте общую характеристику программной реализации алгоритма Фокса. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?
14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

9 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ МЕТОДОВ РЕШЕНИЯ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ

Цель: разработать программу реализующую параллельный алгоритм решения систем линейных уравнений.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Матрицы коэффициентов систем линейных уравнений могут иметь различные структуру и свойства. Матрицы решаемых систем могут быть плотными, и их порядок может достигать несколько тысяч строк и столбцов. При решении многих задач могут появляться системы, обладающие симметричными положительно определенными ленточными матрицами с порядком в десятки тысяч и шириной ленты в несколько тысяч элементов. И, наконец, при рассмотрении большого ряда задач могут возникать системы линейных уравнений с разреженными матрицами с порядком в миллионы строк и столбцов.

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

Рассмотрим возможный вариант параллельной реализации метода Гаусса для решения систем линейных уравнений. Следует отметить, что для получения более простого вида программы для разделения матрицы между процессами используется ленточная последовательная схема (полосы матрицы образуют последовательные наборы соседних строк матрицы). В описании программы реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 1. — Алгоритм Гаусса решения систем линейных уравнений
int ProcNum;           // Число доступных процессоров
int ProcRank;          // Ранг текущего процессора
int *pParallelPivotPos; // Номера строк, которые были выбраны ведущими
int *pProcPivotIter;    // Номера итераций, на которых строки
                        // процессора использовались в качестве ведущих
void main(int argc, char* argv[]) {
    double* pMatrix;    // Матрица линейной системы
    double* pVector;    // Вектор правых частей линейной системы
```

```

double* pResult;           // Вектор неизвестных
double *pProcRows;  // Строки матрицы A
double *pProcVector; // Блок вектора b
double *pProcResult; // Блок вектора x
int  Size;           // Размер матрицы и векторов
int  RowNum;         // Количество строк матрицы
double start, finish, duration;
setvbuf(stdout, 0, _IONBF, 0);
MPI_Init ( &argc, &argv );
MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank );
MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum );
if (ProcRank == 0)
    printf("Параллельный метод Гаусса для решения систем линейных уравне-
ний\n");
// Выделение памяти и инициализация данных
ProcessInitialization(pMatrix, pVector, pResult,
    pProcRows, pProcVector, pProcResult, Size, RowNum);
// Распределение исходных данных
DataDistribution(pMatrix, pProcRows, pVector, pProcVector,
    Size, RowNum);
// Выполнение параллельного алгоритма Гаусса
ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size,
    RowNum);
// Сбор найденного вектора неизвестных на ведущем процессе
ResultCollection(pProcResult, pResult);
// Завершение процесса вычислений
ProcessTermination(pMatrix, pVector, pResult, pProcRows,
    pProcVector, pProcResult);
MPI_Finalize(); }

```

Следует пояснить использование дополнительных массивов. Элементы массива `pParallelPivotPos` определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений. Массив `pParallelPivotPos` является глобальным, и любое его изменение в одном из процессов требует выполнения операции рассылки измененных данных всем остальным процессам программы.

Элементы массива `pProcPivotIter` определяют номера итераций прямого хода метода Гаусса, на которых строки процесса использовались в качестве ведущих (т.е. строка `i` процесса выбиралась ведущей на итерации `pProcPivotIter[i]`). Начальное значение элементов массива устанавливается нулевым, и, тем самым, нулевое значение элемента массива `pProcPivotIter[i]`

является признаком того, что строка i процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемые в элементах массива `pProcPivotIter` номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строки уравнения. Массив `pProcPivotIter` является локальным для каждого процесса.

Функция `ProcessInitialization` определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных, осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи какого-либо датчика случайных чисел).

Функция `DataDistribution` реализует распределение матрицы линейной системы и вектора правых частей между процессорами вычислительной системы.

Функция `ResultCollection` осуществляет сбор со всех процессов отдельных частей вектора неизвестных.

Функция `ProcessTermination` выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

Функция `ParallelResultCalculation`. Реализует логику работы параллельного алгоритма Гаусса, последовательно вызывает функции, выполняющие прямой и обратный ход метода.

```
// Функция для параллельного выполнения метода Гаусса
void ParallelResultCalculation (double* pProcRows,
double* pProcVector, double* pProcResult, int Size, int RowNum) {
    ParallelGaussianElimination (pProcRows, pProcVector, Size,
    RowNum);
    ParallelBackSubstitution (pProcRows, pProcVector, pProcResult,
    Size, RowNum); }
```

Функция `ParallelGaussianElimination`. Функция выполняет параллельный вариант прямого хода алгоритма Гаусса.

```
// Функция для параллельного выполнения прямого хода метода Гаусса
void ParallelGaussianElimination (double* pProcRows,
double* pProcVector, int Size, int RowNum) {
    double MaxValue; // Значение ведущего элемента на процессоре
    int PivotPos; // Положение ведущей строки в полосе линейной
// системы данного процессора
// Структура для выбора ведущего элемента
struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;
// pPivotRow используется для хранения ведущей строки матрицы и
// соответствующего элемента вектора b
```

```

double* pPivotRow = new double [Size+1];
// Итерации прямого хода метода Гаусса
for (int i=0; i<Size; i++) {
    // Нахождение ведущей строки среди строк процесса
    double MaxValue = 0;
    for (int j=0; j<RowNum; j++) {
        if ((pProcPivotIter[j] == -1) &&
            (MaxValue < fabs(pProcRows[j*Size+i]))) {
            MaxValue = fabs(pProcRows[j*Size+i]);
            PivotPos = j; } }
    ProcPivot.MaxValue = MaxValue;
    ProcPivot.ProcRank = ProcRank;
    // Нахождение ведущего процесса (процесса, который содержит
    // максимальное значение переменной MaxValue)
    MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT,
MPI_MAXLOC, MPI_COMM_WORLD);
    // Рассылка ведущей строки
    if ( ProcRank == Pivot.ProcRank )
{ pProcPivotIter[PivotPos]= i; // номер итерации
  pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos; }
    MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
MPI_COMM_WORLD);
    if ( ProcRank == Pivot.ProcRank )
{
    // заполнение ведущей строки
    for (int j=0; j<Size; j++)
{ pPivotRow[j] = pProcRows[PivotPos*Size + j]; }
    pPivotRow[Size] = pProcVector[PivotPos]; }
    MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
MPI_COMM_WORLD);
    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow,
Size, RowNum, i); } }

```

Функция `ParallelEliminateColumns` проводит вычитание ведущей строки из строк процесса, которые еще не использовались в качестве ведущих (т.е. для которых элементы массива `pProcPivotIter` равны нулю).

Функция `ParallelBackSubstitution`. Функция реализует параллельный вариант обратного хода Гаусса.

```

// Функция для параллельного выполнения обратного хода метода Гаусса
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
double* pProcResult, int Size, int RowNum) {
    int IterProcRank; // Ранг процессора, который содержит ведущую строку
    int IterPivotPos; // Положение ведущей строки в полосе процессора

```

```

double IterResult;      // Вычисленное значение очередной неизвестной
double val;
// Итерации обратного хода метода Гаусса
for (int i=Size-1; i>=0; i--) {
    // Вычисление ранга процессора, который содержит ведущую строку
    FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
        IterPivotPos);
    // Вычисление значения неизвестной
    if (ProcRank == IterProcRank) {
        IterResult = pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
        pProcResult[IterPivotPos] = IterResult; }
    // Рассылка значения очередной неизвестной
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank, MPI_COMM_WORLD);
    // Обновление вектора правых частей
    for (int j=0; j<RowNum; j++)
        if ( pProcPivotIter[j] < i ) {
            val = pProcRows[j*Size + i] * IterResult;
            pProcVector[j]=pProcVector[j] - val; } } }

```

Функция FindBackPivotRow определяет строку, из которой можно вычислить значение очередного элемента результирующего вектора. Номер этой строки хранится в массиве pParallelPivotIter. По номеру функция FindBackPivotRow определяет номер процесса, на котором эта строка хранится, и номер этой строки в полосе pProcRows этого процесса.

ЗАДАНИЕ

1. Выполните анализ эффективности параллельных вычислений в отдельности для прямого и обратного этапов метода Гаусса. Оцените, на каком этапе происходит большее снижение показателей.
2. Выполните разработку параллельного варианта метода Гаусса при вертикальном разбиении матрицы по столбцам. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.
3. Выполните реализацию параллельного метода сопряженных градиентов. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.
4. Выполните разработку параллельных вариантов методов Якоби и Зейделя решения систем линейных уравнений. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты.

рименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что представляет собой система линейных уравнений? Какие типы систем вам известны? Какие методы могут быть использованы для решения систем разных типов?
2. В чем состоит постановка задачи решения системы линейных уравнений?
3. В чем идея параллельной реализации метода Гаусса?
4. Какие информационные взаимодействия имеются между базовыми подзадачами для параллельного варианта метода Гаусса?
5. Каковы показатели эффективности для параллельного варианта метода Гаусса?
6. В чем состоит схема программной реализации параллельного варианта метода Гаусса?
7. В чем состоит идея параллельной реализации метода сопряженных градиентов?
8. Какой из алгоритмов обладает большей коммуникационной сложностью?

10 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ МЕТОДОВ СОРТИРОВКИ

Цель: разработать программу реализующую параллельный алгоритм решения систем линейный уравнений.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Сортировка массива - расположение его элементов в определенном порядке (по возрастанию или убыванию). Она необходима для улучшения работы с большим количеством данных. Сортировка массивов облегчает поиск элементов, так как в упорядоченном массиве на него тратится намного меньше времени, а некоторые алгоритмы поиска не выполняются, если данные не отсортированы. Существует много различных методов сортировки, различающихся по эффективности.

Примеры наиболее распространенных непараллельных алгоритмов сортировки:

- Сортировка вставками
- Сортировка пузырьком
- Сортировка слиянием
- Пирамидальная сортировка
- Сортировка подсчетом
- Быстрая сортировка (QSORT)

Среди параллельных алгоритмов сортировки наиболее распространены:

- Метод сдваивания
- «Обменная сортировка со слиянием» Бэтчера
- Параллельная быстрая сортировка
- Параллельная сортировка пузырьком

Рассмотрим алгоритм "пузырьковой" сортировки и его модификации-алгоритм "нечетно-четных перестановок" и "блочный алгоритм", позволяющие распараллелить вычисления на несколько процессоров.

Последовательный алгоритм пузырьковой сортировки сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности $(a_1, a_2, \dots, a_n)$ алгоритм сначала выполняет $n-1$ базовых операций "сравнения-обмена" для последовательных пар элементов (a_1, a_2) , (a_2, a_3) , ..., (a_{n-1}, a_n) . В результате после первой итерации алгоритма самый большой элемент перемещается ("всплывает") в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности $(a'_1, a'_2, \dots, a'_{n-1})$. Как можно увидеть, последовательность будет отсортирована после $n-1$ итерации.

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен

для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения обычно используется не сам этот алгоритм, а его модификация, известная в литературе как метод чет-нечетной перестановки (odd-even transposition).

Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары (a_1, a_2) , (a_3, a_4) , ..., (a_{n-1}, a_n) (при четном n), а на четных итерациях обрабатываются элементы (a_2, a_3) , (a_4, a_5) , ..., (a_{n-2}, a_{n-1}) . После n -кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

```
// Параллельный алгоритм чет-нечетной перестановки
ParallelOddEvenSort(double A[], int n) {
    int id = GetProcId();          // номер процесса
    int np = GetProcNum();         // количество процессов
    for (int i = 0; i < np; i++) {
        if (i % 2 == 1) {         // нечетная итерация
            if (id % 2 == 1) {     // нечетный номер процесса
                // Сравнение-обмен с процессом – соседом справа
                if (id < np - 1) compare_split_min(id + 1);
            }
            else
                // Сравнение-обмен с процессом – соседом слева
                if (id > 0) compare_split_max(id - 1);
        }
        else {                    // четная итерация
            if(id % 2 == 0) {      // четный номер процесса
                // Сравнение-обмен с процессом – соседом справа
                if (id < np - 1) compare_split_min(id + 1);
            }
            else
                // Сравнение-обмен с процессом – соседом слева
                compare_split_max(id - 1);
        }
    }
}
```

Рассмотрим ситуацию, когда количество вычислительных элементов является меньшим числа упорядочиваемых значений ($p < n$). Разделим сортируемый массив на блоки данных размера n/p . На первом этапе параллельного метода пузырьковой сортировки каждый вычислительный элемент выполняет сортировку одного из блоков данных при помощи какого-либо быстрого

алгоритма; эти действия могут быть выполнены параллельно. На следующем этапе выполняется параллельный алгоритм чет-нечетной перестановки над блоками данных:

- на четных итерациях алгоритма вычислительные элементы с четными индексами $2i$ выполняют операцию «Сравнить и разделить» для двух упорядоченных блоков с номерами $2i$ и $2i+1$,

- на нечетных итерациях алгоритма вычислительные элементы с нечетными индексами $2i+1$ выполняют операцию «Сравнить и разделить» для двух упорядоченных блоков с номерами $2i+1$ и $2i+2$.

После выполнения p итераций чет-нечетной перестановки исходный массив оказывается упорядоченным.

Будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для сортировки массива данных методом чет-нечетной перестановки требуется выполнение n итераций алгоритма, на каждой из которых параллельно выполняется сравнение $n/2$ пар значений. Значит, время выполнения вычислений составляет:

$$T_p \text{ calc} = \frac{n^2 - n}{2p} \cdot \tau$$

Если объем сортируемых данных настолько велик, что не может быть полностью помещен в кэш вычислительного элемента, то на каждой итерации метода выполняется постепенное вытеснение значений, располагаемых в начале массива, для того, чтобы записать на их место значения, располагаемые в конце массива. Таким образом, для перехода к выполнению следующей итерации необходимо снова считывать значения из начала сортируемого набора. Таким образом, на каждой итерации происходит повторное считывание всего массива данных из оперативной памяти в кэш, и затраты на доступ к памяти составляют:

$$T_p \text{ mem} = \frac{n^2 - n}{2} \cdot \frac{64}{\beta}$$

Если учесть латентность памяти, то время доступа к памяти можно получить по следующей формуле:

$$T_p \text{ mem} = \frac{n^2 - n}{2} \cdot \alpha + 64 / \beta$$

При получении итоговой оценки времени выполнения базового параллельного алгоритма пузырьковой сортировки необходимо также учитывать затраты на организацию и закрытие параллельных секций:

$$T_p = \frac{n^2 - n}{2p} \cdot \tau + \frac{n^2 - n}{2} \cdot \alpha + 64 / \beta + n\delta$$

где δ есть накладные расходы на организацию параллельности на каждой итерации алгоритма.

где t есть время выполнения базовой операции сортировки.

где b есть эффективная скорость доступа к оперативной памяти.

Для более точной оценки необходимо учесть частоту кэш-промахов γ :

$$T_p = \frac{n^2 - n}{2p} \cdot \tau + \gamma \cdot \frac{n^2 - n}{2} \cdot \alpha + 64/\beta + n\delta$$

Определим теперь сложность блочность алгоритма упорядочивания данных. Как отмечалось ранее, на начальной стадии работы метода каждый процессор проводит упорядочивание своих блоков данных (размер блоков при равномерном распределении данных является равным n/p). Трудоемкость начальной стадии вычислений можно определить выражением вида:

$$T_p^1 = \frac{n^2}{p^2}$$

Далее, на каждой выполняемой итерации параллельной сортировки взаимодействующие пары процессоров осуществляют передачу блоков между собой, после чего получаемые на каждом процессоре пары блоков объединяются при помощи процедуры слияния. Общее количество итераций не превышает величины p , и, как результат, общее количество операций этой части параллельных вычислений оказывается равным:

$$T_p^2 = 2p \frac{n}{p} = 2n$$

С учетом полученных соотношений показатели эффективности и ускорения параллельного метода сортировки имеют вид:

$$S_p = \frac{n^2}{\frac{n^2}{p^2} + 2n} = \frac{p^2 n}{n + 2p^2} \quad E_p = \frac{n^2}{p(\frac{n^2}{p^2} + 2n)}$$

Расширим приведенные выражения – учтем длительность выполняемых вычислительных операций и оценим трудоемкость операции передачи блоков между процессорами. При использовании модели Хокни общее время всех выполняемых в ходе сортировки операций передачи блоков можно оценить при помощи соотношения вида:

$$T_p(\text{comm}) = p \cdot (\alpha + w \cdot (n/p) / \beta),$$

где α – латентность, β – пропускная способность сети передачи данных, а w есть размер элемента упорядочиваемых данных в байтах.

С учетом трудоемкости коммуникационных действий общее время выполнения параллельного алгоритма сортировки определяется следующим выражением:

$$T_p = \left(\frac{n^2}{p^2} + 2n \right) \tau + p \left(\alpha + \frac{w \left(\frac{n}{p} \right)}{\beta} \right)$$

где t есть время выполнения базовой операции сортировки.

ЗАДАНИЕ

1. Выполните реализацию параллельного алгоритма пузырьковой сортировки. Проведите эксперименты. Постройте теоретические оценки с учетом тех операций пересылок данных, которые использовались при реализации, и параметров вычислительной системы. Сравните получаемые теоретические оценки с результатами экспериментов.
2. Выполните реализацию параллельного алгоритма быстрой сортировки по одной из приведенных схем. Определите значения параметров латентности, пропускной способности и времени выполнения базовой операции для используемой вычислительной системы и получите оценки показателей ускорения и эффективности для реализованного метода параллельных вычислений.
3. Разработайте параллельную схему вычислений для широко известного алгоритма сортировки слиянием. Выполните реализацию разработанного алгоритма и постройте все необходимые теоретические оценки сложности метода.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит постановка задачи сортировки данных?
2. Приведите несколько примеров алгоритмов сортировки. Какова вычислительная сложность приведенных алгоритмов?
3. Какая операция является базовой для задачи сортировки данных?
4. В чем суть параллельного обобщения базовой операции задачи сортировки данных?
5. Что представляет собой алгоритм чет-нечетной перестановки?
6. В чем состоит параллельный вариант алгоритма Шелла? Каковы основные отличия этого параллельного алгоритма сортировки от метода чет-нечетной перестановки?
7. Что представляет собой параллельный вариант алгоритма быстрой сортировки?
8. Что зависит от правильного выбора ведущего элемента для параллельного алгоритма быстрой сортировки?
9. Какие способы выбора ведущего элемента могут быть предложены?
10. Для каких топологий могут применяться рассмотренные алгоритмы сортировки?
11. В чем состоит алгоритм сортировки с использованием регулярного набора образцов?

11 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ МЕТОДОВ НА ГРАФАХ

Цель: разработать программу реализующую параллельный алгоритм Флойда.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Исходной информацией для задачи является взвешенный граф $G=(V, R)$, содержащий n вершин ($|V|=n$), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать ориентированным, т.е. если из вершины i есть ребро в вершину j , то из этого не следует наличие ребра из j в i . В случае если вершины все же соединены взаимнообратными ребрами, веса, приписанные им, могут не совпадать. Рассмотрим задачу, в которой для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется алгоритм Флойда.

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок $O(n^3)$. В общем виде данный алгоритм может быть представлен следующим образом:

Алгоритм 1. Общая схема алгоритма Флойда

```
// Алгоритм 1
```

```
// Последовательный алгоритм Флойда
```

```
for (k = 0; k < n; k++)
```

```
    for (i = 0; i < n; i++)
```

```
        for (j = 0; j < n; j++)
```

```
            A[i, j] = min(A[i, j], A[i, k] + A[k, j]);
```

(реализация операции выбора минимального значения $\min$ должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значений. Данная операция является достаточно простой, и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A .

Покажем корректность такого способа организации параллелизма. Для этого нужно доказать, что операции обновления значений матрицы A на одной и той же итерации внешнего цикла k могут выполняться независимо. Иными словами, следует показать, что на итерации k не происходит изменения элементов A_{ik} и A_{kj} ни для одной пары индексов (i, j) . Рассмотрим выражение, по которому происходит изменение элементов матрицы A :

$$A_{ij} \leftarrow \min(A_{ij}, A_{ik} + A_{kj}).$$

Для $i=k$ получим

$$A_{kj} \leftarrow \min(A_{kj}, A_{kk} + A_{kj}),$$

но тогда значение A_{kj} не изменится, т.к. $A_{kk}=0$.

Для $j=k$ выражение преобразуется к виду

$$A_{ik} \leftarrow \min(A_{ik}, A_{ik} + A_{kk}),$$

что также показывает неизменность значений A_{ik} . Как результат, необходимые условия для организации параллельных вычислений обеспечены, и, тем самым, в качестве базовой подзадачи может быть использована операция обновления элементов матрицы A (для указания подзадач будем применять индексы обновляемых в подзадачах элементов).

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача (i, j) содержит необходимые для расчетов элементы A_{ij} , A_{ik} , A_{kj} матрицы A . Для исключения дублирования данных разместим в подзадаче (i, j) единственный элемент A_{ij} , тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким образом, каждый элемент A_{kj} строки k матрицы A должен быть передан всем подзадачам (k, j) , $1 \leq j \leq n$, а каждый элемент A_{ik} столбца k матрицы A должен быть передан всем подзадачам (i, k) , $1 \leq i \leq n$ (рисунок 1).

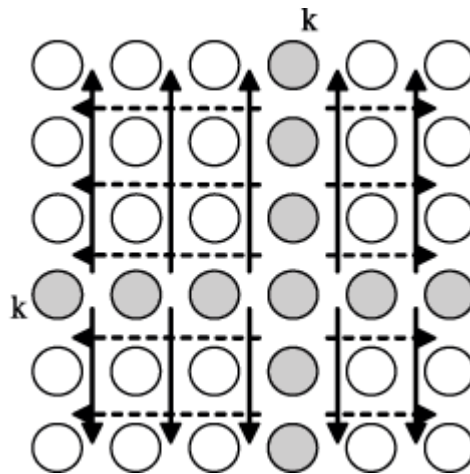


Рисунок 1 - Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p < n^2$). Возможный способ укрупнения вычислений состоит в использовании ленточной схемы разбиения матрицы A – та-

кой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (горизонтальное разбиение) или столбцов (вертикальное разбиение) матрицы A . Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка C массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

Следует отметить, что при таком способе разбиения данных на каждой итерации алгоритма Флойда потребуется передавать между подзадачами только элементы одной из строк матрицы A . Для оптимального выполнения подобной коммуникационной операции топология сети должна обеспечивать эффективное представление структуры сети передачи данных в виде гиперкуба или полного графа.

Выполним анализ эффективности параллельного алгоритма Флойда, обеспечивающего поиск всех кратчайших путей. Как и ранее, проведем этот анализ в два этапа. На первом оценим порядок вычислительной сложности алгоритма, затем на втором этапе уточним полученные оценки и учтем трудоемкость выполнения коммуникационных операций.

Общая трудоемкость последовательного алгоритма, как уже отмечалось ранее, имеет порядок сложности n^3 . Для параллельного алгоритма на отдельной итерации каждый процессор выполняет обновление элементов матрицы A . Всего в подзадачах n^2/p таких элементов, число итераций алгоритма равно n – таким образом, показатели ускорения и эффективности параллельного алгоритма Флойда имеют вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \quad (1)$$

Следовательно, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений введем в полученные выражения время выполнения базовой операции выбора минимального значения и учтем затраты на выполнение операций передачи данных между процессорами.

Коммуникационная операция, выполняемая на каждой итерации алгоритма Флойда, состоит в передаче одной из строк матрицы A всем процессорам вычислительной системы. Как уже показывалось ранее, такая операция может быть выполнена за $\log_2 p$ шагов. С учетом количества итераций алгоритма Флойда при использовании модели Хокни общая длительность выполнения коммуникационных операций может быть определена при помощи следующего выражения

$$T'_p(comm) = n \lceil \log_2 p \rceil (\alpha + wn/\beta), \quad (2)$$

где, как и ранее, α – латентность сети передачи данных, β – пропускная способность сети, а w есть размер элемента матрицы в байтах.

С учетом полученных соотношений общее время выполнения параллельного алгоритма Флойда может быть определено следующим образом:

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2 p \rceil (\alpha + w \cdot n/\beta), \quad (3)$$

где τ - есть время выполнения операции выбора минимального значения.

Для параллельной реализации алгоритма Флойда используем технологию MPI. При этом описание отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 2 – Алгоритм Флойда
int ProcRank;      // Ранг текущего процесса
int ProcNum;       // Количество процессов
// Функция вычисления минимума
int Min(int A, int B) {
    int Result = (A < B) ? A : B;
    if((A < 0) && (B >= 0)) Result = B;
    if((B < 0) && (A >= 0)) Result = A;
    if((A < 0) && (B < 0)) Result = -1;
    return Result;
}
// Главная функция программы
int main(int argc, char* argv[]) {
    int *pMatrix;    // Матрица смежности
    int Size;        // Размер матрицы смежности
    int *pProcRows; // Строки матрицы смежности текущего процесса
    int RowNum;      // Число строк для текущего процесса
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    // Инициализация данных
    ProcessInitialization(pMatrix, pProcRows, Size, RowNum);
    // Распределение данных между процессами
    DataDistribution(pMatrix, pProcRows, Size, RowNum);
    // Параллельный алгоритм Флойда
    ParallelFloyd(pProcRows, Size, RowNum);
    // Сбор результатов работы алгоритма
    ResultCollection(pMatrix, pProcRows, Size, RowNum);
    // Завершение вычислений процесса
    ProcessTermination(pMatrix, pProcRows);
    MPI_Finalize();
    return 0;
}
```

Функция Min вычисляет меньшее из двух целых чисел, учитывая применяемый способ задания несуществующих дуг в матрице смежности (в рассматриваемой реализации для этого используется значение -1).

Функция ProcessInitialization определяет исходные данные решаемой

задачи (количество вершин графа), выделяет память для хранения данных, осуществляет ввод матрицы смежности (или формирует ее при помощи какого-либо датчика случайных чисел).

Функция `DataDistribution` распределяет исходные данные между процессами. Каждый процесс получает горизонтальную полосу матрицы смежности.

Функция `ResultCollection` осуществляет сбор со всех процессов горизонтальных полос результирующей матрицы кратчайших расстояний между любыми парами вершин графа.

Функция `ProcessTermination` выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

Данная функция `ParallelFloyd` осуществляет итеративное изменение матрицы смежности в соответствии с алгоритмом Флойда.

```
// Параллельный алгоритм Флойда
void ParallelFloyd(int *pProcRows, int Size, int RowNum)
{ int *pRow = new int[Size];
  int t1, t2;
  for(int k = 0; k < Size; k++) {
    // Распределение k-й строки среди процессов
    RowDistribution(pProcRows, Size, RowNum, k, pRow);
    // Обновление элементов матрицы смежности
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if((pProcRows[i * Size + k] != -1) && (pRow[j] != -1)){
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2); } }
  delete []pRow;}
```

Функция `RowDistribution` рассылает k -ю строку матрицы смежности всем процессам программы.

```
// Функция для рассылки строки всем процессам
void RowDistribution(int *pProcRows, int Size, int RowNum, int k, int *pRow)
{int ProcRowRank; // Ранг процесса, которому принадлежит k-я строка
  int ProcRowNum; // Номер k-й строки в полосе матрицы
  // Нахождение ранга процесса - владельца k-й строки
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;
```

```

for (ProcRowRank=1; ProcRowRank<ProcNum+1; ProcRowRank++)
{
    if (k < Ind + Num) break;
    RestRows -= Num;
    Ind      += Num;
    Num      = RestRows / (ProcNum - ProcRowRank);
    ProcRowRank = ProcRowRank - 1;
    ProcRowNum  = k - Ind;
    if (ProcRowRank == ProcRank)
        // Копирование строки в массив pRow
        copy(&pProcRows[ProcRowNum*Size],
        &pProcRows[ProcRowNum+1)*Size], pRow);
    //Распределение k-й строки между процессами
    MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD); }

```

ЗАДАНИЕ

Используя приведенный программный код, выполните реализацию параллельного алгоритма Флойда. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите определение графа. Какие основные способы используются для задания графов?
2. В чем состоит задача поиска всех кратчайших путей?
3. Приведите общую схему алгоритма Флойда. Какова трудоемкость алгоритма?
4. В чем состоит способ распараллеливания алгоритма Флойда?
5. В чем заключается задача нахождения минимального охватывающего дерева? Приведите пример использования задачи на практике.
6. Приведите общую схему алгоритма Прима. Какова трудоемкость алгоритма?
7. В чем состоит способ распараллеливания алгоритма Прима?
8. В чем отличие геометрических и комбинаторных методов разделения графа? Какие методы являются более предпочтительными? Почему?
9. Приведите описание метода покоординатного разбиения и алгоритма разделения с учетом связности. Какой из этих методов является более простым для реализации?

12 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ МЕТОДОВ РЕШЕНИЯ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ В ЧАСТНЫХ ПРОИЗВОДНЫХ

Цель: разработать программу реализующую параллельный алгоритм решения дифференциальных уравнений в частных производных.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Одним из наиболее распространенных подходов численного решения дифференциальных уравнений является *метод конечных разностей* (*метод сеток*). Следуя этому подходу, область решения D представляется в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*). Так, например, прямоугольная сетка в области D может быть задана в виде (рис. 1):

$$\begin{cases} D_h = \{(x_i, y_j) : x_i = ih, y_j = jh, 0 \leq i, j \leq N + 1, \\ h = 1/(N + 1) \end{cases}$$

где величина N задает количество узлов по каждой из координат области D .

Обозначим оцениваемую при подобном дискретном представлении проксимацию функции $u(x, y)$ в точках (x_i, y_j) через u_{ij} . Тогда, используя *пяти-точечный шаблон* (см. рис. 1) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*:

$$\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} = f_{ij}$$

Данное уравнение может быть разрешено относительно u_{ij} :

$$u_{ij} = 0,25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение u_{ij} по известным значениям функции $u(x, y)$ в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij} , а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, *метод Гаусса-Зейделя* для проведения итераций уточнения использует правило:

$$u_{ij}^k = 0,25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij})$$

по которому очередное k -ое приближение значения u_{ij} вычисляется по последнему k -ому приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему $(k-1)$ -ому приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений u_{ij} не станут меньше некоторой заданной величины (*требуемой точности вычислений*). Сходимость описанной процедуры (получение решения с любой желаемой точностью) является предметом всестороннего мате-

матического анализа, здесь же отметим, что последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2 .

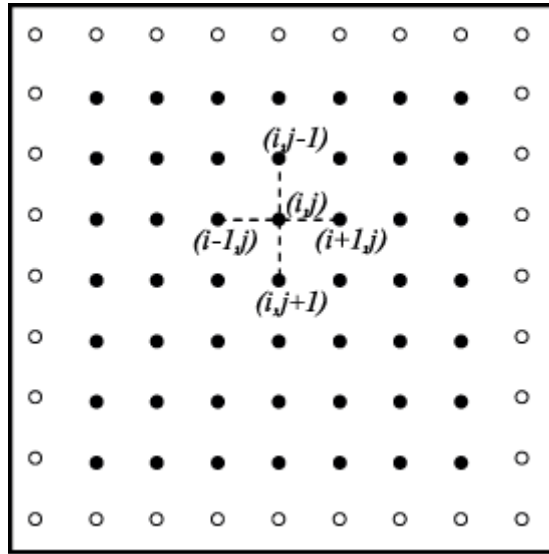


Рис. 1 - Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах - сверху вниз)

Рассмотренный алгоритм (метод Гаусса-Зейделя) на псевдокоде, приближенном к алгоритмическому языку C++, может быть представлен в виде:

```
//Алгоритм 1
//Метод Гаусса-Зейделя
//do { dmax = 0; // максимальное изменение значений u
for ( i=1; i<N+1; i++ ) for ( j=1; j<N+1; j++ )
{ temp = u[i][j];
u[i][j]=0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]-
h*h*f[i][j]);
dm = fabs(temp-u[i][j]);
if ( dmax < dm ) dmax = dm;}}
while ( dmax > eps );
```

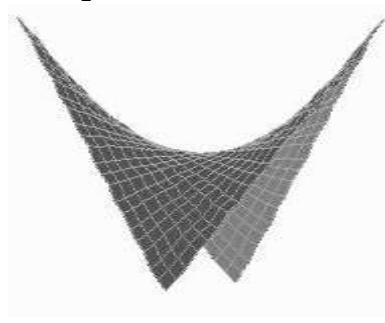


Рис 2 - Вид функции $u(x,y)$ в примере для задачи Дирихле

Для примера на рис. 2 приведен вид функции $u(x,y)$, полученной для задачи Дирихле при следующих граничных условиях:

$$\begin{cases} f(x, y) = 0, & (x, y) \in D, \\ 100 - 200x, & y = 0, \\ 100 - 200x, & x = 0, \\ -100 + 200x, & y = 1, \\ -100 + 200x, & x = 1. \end{cases}$$

Общее количество итераций метода Гаусса-Зейделя составило 210 при точности решения $eps = 0.1$ и $N = 100$ (в качестве начального приближения величин u_{ij} использовались значения, сгенерированные датчиком случайных чисел из диапазона $[-100, 100]$).

Использование процессоров с распределенной памятью является другим общим способом построения многопроцессорных вычислительных систем. Актуальность их становится все более высокой в последнее время в связи с широким развитием высокопроизводительных кластерных вычислительных систем.

Многие проблемы параллельного программирования (согласование вычислений, тупики, сериализация) являются общими для систем с общей и распределенной памятью. Основным момент, который отличает параллельные вычисления с распределенной памятью, состоит в том, что взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений (message passing)*.

Следует отметить, что вычислительный узел системы с распределенной памятью является, как правило, более сложным вычислительным устройством, чем процессор в многопроцессорной системе с общей памятью. Для учета этих различий в дальнейшем процессор с распределенной памятью будет именоваться *вычислительным сервером* (сервером может быть, в частности, многопроцессорная система с общей памятью).

Первая проблема, которую приходится решать при организации параллельных вычислений на системах с распределенной памятью, обычно состоит в выборе способа разделения обрабатываемых данных между вычислительными серверами. Успешность такого разделения определяется достигнутой степенью локализации вычислений на серверах (в силу больших временных задержек при передаче сообщений интенсивность взаимодействия серверов должна быть минимальной).

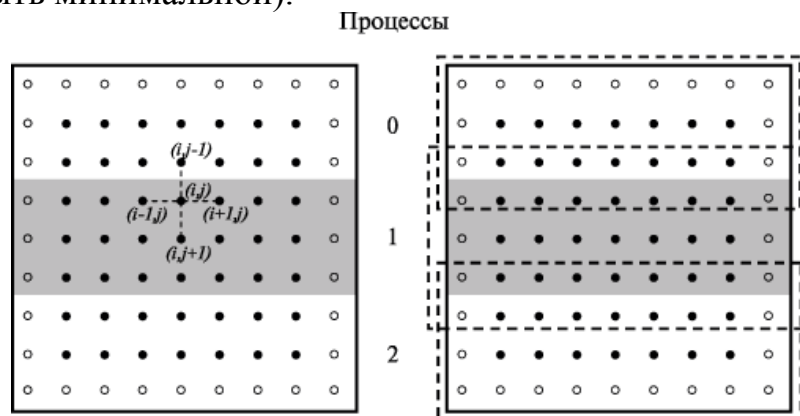


Рис. 3 - Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемом учебном примере по решению задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема (см. рис. 3) и *двумерное* или *блочное* разбиение (см. рис. 4) вычислительной сетки. Дальнейшее изложение учебного материала будет проводиться на примере первого подхода; блочная схема будет рассмотрена позднее в более кратком виде.

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (не уменьшая общности, далее будем рассматривать только горизонтальные полосы). Число полос определяется количеством процессоров, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессорами.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий обработку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки. Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым, дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся серверах одновременно в соответствии со следующей схемой работы:

Алгоритм 2. Параллельный алгоритм, реализующий метод сеток при ленточном разделении данных

```
// Алгоритм 2
// схема Гаусса-Зейделя, ленточное разделение дан-
ных
// действия, выполняемые на каждом процессоре
do {
    // <обмен граничных строк полос с соседями>
    // <обработка полосы>
    // <вычисление общей погрешности вычислений
dmax>}
while ( dmax > eps ); // eps – точность решения
```

Для конкретизации представленных в алгоритме действий введем обозначения:

ProcNum – номер процессора, на котором выполняются описываемые действия,

PrevProc, NextProc – номера соседних процессоров, содержащих предшествующую и следующую полосы,

NP – количество процессоров,

M – количество строк в полосе (без учета продублированных граничных строк),

N – количество внутренних узлов в строке сетки (т.е. всего в строке $N+2$ узла).

При нумерации строк полосы будем считать, что строки 0 и $M+1$ есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от 1 до M .

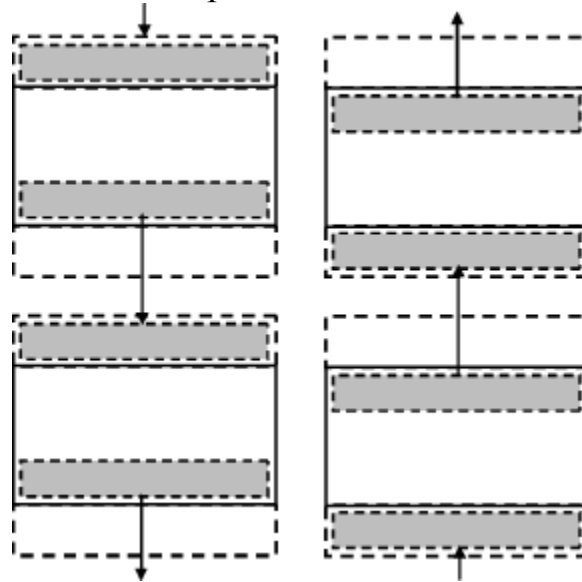


Рис. 4. Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис. 4). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для краткости рассмотрим только первую часть процедуры обмена):

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
```

Для передачи данных могут быть задействованы два различных механизма. При первом из них выполнение программ, инициировавших операцию передачи, приостанавливается до полного завершения всех действий по пересылке данных (т.е. до момента получения процессором-адресатом всех передаваемых ему данных). Операции приема-передачи, реализуемые подобным образом, обычно называются *синхронными* или *блокирующими*. Иной подход

– *асинхронная* или *неблокирующая* передача — может состоять в том, что операции приема-передачи только инициируют процесс пересылки и на этом завершают свое выполнение. В результате программы, не дожидаясь завершения длительных коммуникационных операций, могут продолжать свои вычислительные действия, проверяя по мере необходимости готовность передаваемых данных. Оба эти варианта операций передачи широко используются при организации параллельных вычислений и имеют свои достоинства и свои недостатки. Синхронные процедуры передачи, как правило, более просты для применения и более надежны; неблокирующие операции могут позволить совместить процессы передачи данных и вычислений, но обычно приводят к повышению сложности программирования. С учетом вышесказанного во всех последующих примерах для организации пересылки данных будут использоваться операции приема-передачи блокирующего типа.

Приведенная выше последовательность блокирующих операций приема-передачи данных (вначале Send, затем Receive) приводит к строго последовательной схеме выполнения процесса пересылок строк, т.к. все процессоры одновременно обращаются к операции Send и переходят в режим ожидания. Первым процессором, который окажется готовым к приему пересылаемых данных, будет сервер с номером NP-1. В результате процессор NP-2 выполнит операцию передачи своей граничной строки и перейдет к приему строки от процессора NP-3 и т.д. Общее количество повторений таких операций равно NP-1. Аналогично происходит выполнение и второй части процедуры пересылки граничных строк перед началом обработки строк (см. [рис. 1.11](#)).

Последовательный характер рассмотренных операций пересылок данных определяется выбранным способом очередности выполнения. Изменим этот порядок очередности при помощи чередования приема и передачи для процессоров с четными и нечетными номерами.

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum % 2 == 1 ) { // нечетный процессор
    if ( ProcNum != NP-1 )
Send(u[M][*], N+2, NextProc);
    if ( ProcNum != 0 )
Receive(u[0][*], N+2, PrevProc);
}
else { // процессор с четным номером
    if ( ProcNum != 0 )
Receive(u[0][*], N+2, PrevProc);
    if ( ProcNum != NP-1 )
Send(u[M][*], N+2, NextProc);
}
```

Данный прием позволяет выполнить все необходимые операции пере-

дачи всего за два последовательных шага. На первом шаге все процессоры с нечетными номерами отправляют данные, а процессоры с четными номерами осуществляют прием этих данных. На втором шаге роли процессоров меняются – четные процессоры выполняют Send, нечетные процессоры исполняют операцию приема Receive.

Рассмотренные последовательности операций приема-передачи для взаимодействия соседних процессоров широко используются в практике параллельных вычислений. Как результат, во многих базовых библиотеках параллельных программ имеются процедуры для поддержки подобных действий. Так, в стандарте MPI предусмотрена операция Sendrecv, с использованием которой предыдущий фрагмент программного кода может быть записан более кратко:

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
Sendrecv(u[M][*], N+2, NextProc, u[0][*], N+2, PrevProc);
```

Реализация подобной объединенной функции Sendrecv обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

Для завершения круга вопросов, связанных с параллельной реализацией метода сеток на системах с распределенной памятью, осталось рассмотреть способы вычисления общей для всех процессоров погрешности вычислений. Возможный очевидный подход состоит в передаче всех локальных оценок погрешности, полученных на отдельных полосах сетки, на один какой-либо процессор, вычислении на нем максимального значения и последующей рассылке полученного значения всем процессорам системы. Однако такая схема является крайне неэффективной – количество необходимых операций передачи данных определяется числом процессоров и выполнение этих операций может происходить только в последовательном режиме. Между тем, как показывает анализ требуемых коммуникационных действий, выполнение операций сборки и рассылки данных может быть реализовано с использованием рассмотренной *каскадной схемы* обработки данных. На самом деле, получение максимального значения локальных погрешностей, вычисленных на каждом процессоре, может быть обеспечено, например, путем предварительного нахождения максимальных значений для отдельных пар процессоров (такие вычисления могут выполняться параллельно), затем может быть снова осуществлен попарный поиск максимума среди полученных результатов и т.д. Всего, как полагается по каскадной схеме, необходимо выполнить $\log_2 NP$ параллельных итераций для получения конечного значения (NP – количество процессоров).

С учетом возможности применения каскадной схемы для выполнения

коллективных операций передачи данных большинство базовых библиотек параллельных программ содержит процедуры для поддержки подобных действий. Так, в стандарте MPI предусмотрены операции:

`Reduce(dm,dmax,op,proc)` – процедура сборки на процессоре `proc` итогового результата `dmax` среди локальных на каждом процессоре значений `dm` с применением операции `op` ;

`Broadcast(dmax,proc)` – процедура рассылки с процессора `proc` значения `dmax` всем имеющимся процессорам системы.

С учетом перечисленных процедур общая схема вычислений на каждом процессоре может быть представлена в следующем виде:

```
// Алгоритм 1.8 – уточненный вариант
// схема Гаусса-Зейделя, ленточное разделение дан-
ных
// действия, выполняемые на каждом процессоре
do {
    // обмен граничных строк полос с соседями
    Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
    Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
    // <обработка полосы с оценкой погрешности dm>
    // вычисление общей погрешности вычислений dmax
    Reduce(dm,dmax,MAX,0);
    Broadcast(dmax,0);
} while ( dmax > eps ); // eps – точность решения
```

(в приведенном алгоритме переменная `dm` представляет собой локальную погрешность вычислений на отдельном процессоре, а параметр `MAX` задает операцию поиска максимального значения для операции сборки). Следует отметить, что в составе MPI имеется процедура `Allreduce`, которая совмещает действия редукции и рассылки данных.

Представленные алгоритмы определяют общую схему параллельных вычислений для метода сеток в многопроцессорных системах с распределенной памятью. Далее эта схема может быть конкретизирована реализацией практически всех вариантов методов, рассмотренных для систем с общей памятью (применение дополнительной памяти для схемы Гаусса – Якоби, чередование обработки полос и т.п.). Проработка таких вариантов не привносит каких-либо новых эффектов с точки зрения параллельных вычислений, и их разбор может использоваться как тема заданий для самостоятельных упражнений.

В завершение рассмотрим возможность организации параллельных вычислений, при которых обеспечивалось бы нахождение таких же решений задачи Дирихле, что и при использовании исходного последовательного метода Гаусса – Зейделя. Как отмечалось ранее, такой результат может быть получен за счет организации волновой схемы расчетов. Для образования волны вычислений представим логически каждую полосу узлов области расчетов в виде набора блоков (размер блоков можно положить, в частности, равным ши-

рине полосы) и организуем обработку полос поблочно в последовательном порядке. Тогда для полного повторения действий последовательного алгоритма вычисления могут быть начаты только для первого блока первой полосы узлов; после того как этот блок будет обработан, для вычислений будут готовы уже два блока – блок 2 первой полосы и блок 1 второй полосы (для обработки блока полосы 2 необходимо передать граничную строку узлов первого блока полосы 1). После обработки указанных блоков к вычислениям будут готовы уже 3 блока, и мы получаем знакомый уже процесс волновой обработки данных.

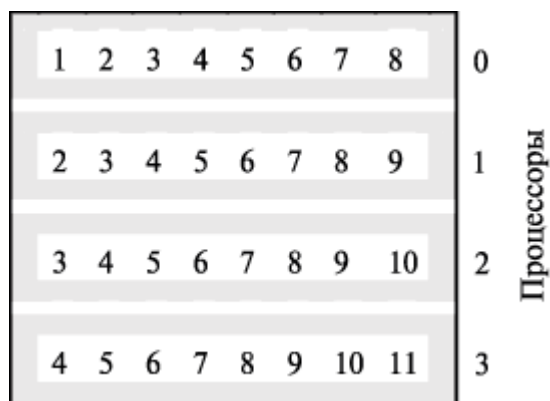


Рис. 5 - Организация волны вычислений при ленточной схеме разделения данных

Интересный момент при организации подобной схемы параллельных вычислений может состоять в попытке совмещения операций пересылки граничных строк и действий по обработке блоков данных.

Ленточная схема разделения данных может быть естественным образом обобщена на блочный способ представления сетки области расчетов. При этом столь радикальное изменение способа разбиения сетки практически не потребует каких-либо существенных корректировок рассмотренной схемы параллельных вычислений. Основной новый момент при блочном представлении данных состоит в увеличении количества граничных строк на каждом процессоре (для блока их количество становится равным 4), что приводит, соответственно, к большему числу операций передачи данных при обмене граничных строк. Сравнивая затраты на организацию передачи граничных строк, можно отметить, что при ленточной схеме для каждого процессора выполняется 4 операции приема-передачи данных, в каждой из которых пересылаются $(N+2)$ значения. Для блочного же способа происходит 8 операций пересылки и объем каждого сообщения равен $(N/\sqrt{NP} + 2)$ (N – количество внутренних узлов сетки, NP – число процессоров, размер всех блоков предполагается одинаковым). Тем самым, блочная схема представления области расчетов становится оправданной при большом количестве узлов сетки, когда увеличение количества коммуникационных операций приводит к снижению затрат на пересылку данных в силу сокращения размеров передаваемых сообщений.

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов. Пусть процессоры образуют прямоугольную решетку размером $NB \times NB$ ($NB = \sqrt{NP}$) и процессоры пронумерованы от 0 слева направо по строкам решетки.

Общая схема параллельных вычислений в этом случае имеет вид:

Алгоритм 2. Блочная схема разделения данных

```
// Алгоритм 2
// схема Гаусса-Зейделя, блочное разделение данных
// действия, выполняемые на каждом процессоре
do {
    // получение граничных узлов
    if ( ProcNum / NB != 0 ) { // строка не нулевая
        // получение данных от верхнего процессора
        Receive(u[0][*], M+2, TopProc); // верхняя строка
        Receive(dmax, 1, TopProc); // погрешность
    }
    if ( ProcNum % NB != 0 ) { // столбец не нулевой
        // получение данных от левого процессора
        Receive(u[*][0], M+2, LeftProc); // левый столбец
        Receive(dm, 1, LeftProc); // погрешность
        if ( dm > dmax ) dmax = dm;
    }
    // <обработка блока с оценкой погрешности dmax>
    // пересылка граничных узлов
    if ( ProcNum/NB != NB-1 ) { // строка решетки не послед-
        няя
            // пересылка данных нижнему процессору
            Send(u[M+1][*], M+2, DownProc); // нижняя строка
            Send(dmax, 1, DownProc); // погрешность
        }
        if ( ProcNum%NB != NB-1 ) { // столбец решетки не послед-
            ний
                // пересылка данных правому процессору
                Send(u[*][M+1], M+2, RightProc); // правый столбец
                Send(dmax, 1, RightProc); // погрешность
            }
        // синхронизация и рассылка погрешности dmax
        Broadcast(dmax, NP-1);
    } while ( dmax > eps ); // eps – точность решения
```

При реализации алгоритма необходимо обеспечить, чтобы в начальный момент времени все процессоры (кроме процессора с нулевым номером) оказались в состоянии передачи своих граничных узлов (верхней строки и левого столбца). Вычисления должен начинать процессор с левым верхним блоком, после завершения обработки которого обновленные значения правого столбца и нижней строки блока нужно переправить правому и нижнему про-

цессорам решетки соответственно. Данные действия обеспечат снятие блокировки процессоров второй диагонали процессорной решетки и т.д.

Анализ эффективности организации волновых вычислений в системах с распределенной памятью показывает значительное снижение полезной вычислительной нагрузки для процессоров, которые занимаются обработкой данных только в моменты, когда их блоки попадают во фронт волны вычислений. При этом балансировка (перераспределение) нагрузки является крайне затруднительной, поскольку связана с пересылкой между процессорами блоков данных большого объема. Возможный интересный способ улучшения ситуации состоит в организации *множественной волны вычислений*, в соответствии с которой процессоры после отработки волны текущей итерации расчетов могут приступить к выполнению волны следующей итерации метода сеток. Так, например, процессор 0, передав после обработки своего блока граничные данные и запустив тем самым вычисления на процессорах 1 и 4, оказывается готовым к исполнению следующей итерации метода Гаусса – Зейделя. После обработки блоков первой (процессоры 1 и 4) и второй (процессор 0) волн к вычислениям окажутся готовыми следующие группы процессоров (для первой волны — процессоры 2, 5 и 8, для второй — процессоры 1 и 4). Кроме того, процессор 0 опять окажется готовым к запуску очередной волны обработки данных. После выполнения NB подобных шагов в обработке будет находиться одновременно NB итераций и все процессоры окажутся задействованными. Подобная схема организации расчетов позволяет рассматривать имеющуюся процессорную решетку как *вычислительный конвейер* поэтапного выполнения итераций метода сеток. Остановка конвейера может осуществляться, как и ранее, по максимальной погрешности вычислений (проверку условия остановки следует начинать только при достижении полной загрузки конвейера после запуска NB итераций расчетов). Необходимо отметить также, что получаемое после выполнения условия остановки решение задачи Дирихле будет содержать значения узлов сетки от разных итераций метода и не будет, тем самым, совпадать с решением, получаемым при помощи исходного последовательного алгоритма.



Рис. 6 - Организация волны вычислений при блочной схеме разделения данных

ЗАДАНИЕ

1. Выполните реализацию первого и второго вариантов параллельного алгоритма Гаусса – Зейделя для систем с общей памятью. Проведите вычислительные эксперименты и сравните время выполнения разработанных программ.
2. Выполните реализации параллельного алгоритма Гаусса – Зейделя при волновой схеме организации вычислений и блочном представлении обрабатываемых данных. Проведите вычислительные эксперименты при разном размере блоков и сравните получаемые характеристики эффективности параллельных вычислений.
3. Выполните реализацию очереди заданий для параллельного алгоритма Гаусса – Зейделя. Подготовьте несколько разных правил выделения заданий из очереди и проведите оценку эффективности для каждого использованного правила.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как определяется задача Дирихле для уравнения Пуассона?
2. В чем состоят основные положения метода конечных разностей?
3. Какие способы распараллеливания сеточных методов могут быть использованы для многопроцессорных вычислительных систем с общей памятью?
4. В каких ситуациях необходима синхронизация параллельных вычислений?
5. Как характеризуется поведение параллельных участков программы при наличии условий состязания потоков?
6. В чем состоит проблема взаимоблокировки?
7. Какие методы могут быть использованы для достижения однозначности результатов параллельных вычислений для сеточных методов?

8. Как изменяется объем вычислений при применении методов волновой обработки данных?
9. Как повысить эффективность методов волновой обработки данных?
10. Как очередь заданий позволяет улучшить балансировку вычислительной нагрузки процессоров?
11. Какие проблемы приходится решать при организации параллельных вычислений на системах с распределенной памятью?
12. Какие основные схемы распределения данных между процессорами могут быть использованы для сеточных методов?
13. Какие основные операции передачи данных используются в параллельных методах решения задачи Дирихле?
14. Каким образом организация множественной волны вычислений позволяет повысить эффективность волновых вычислений в системах с распределенной памятью?

ЛИТЕРАТУРА

1. Барский, А.Б. Параллельные информационные технологии Электронный ресурс : учебное пособие / А.Б. Барский. - Параллельные информационные технологии, 2020-07-28. - Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017. - 503 с. - Книга находится в базовой версии ЭБС IPRbooks. - ISBN 978-5-4487-0087-3
2. Параллельные вычислительные системы Электронный ресурс / Сиротина Н. Ю., Непомнящий О. В., Коршун К. В., Васильев В. С. : учебное пособие. - Красноярск : СФУ, 2019. - 178 с. - ISBN 978-5-7638-4180-0
3. Биллиг, В.А. Параллельные вычисления и многопоточное программирование Электронный ресурс : учебное пособие / В.А. Биллиг. - Параллельные вычисления и многопоточное программирование, 2021-01-23. - Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 310 с. - Книга находится в базовой версии ЭБС IPRbooks.
4. Гергель, В.П. Теория и практика параллельных вычислений Электронный ресурс : учебное пособие / В.П. Гергель. - Теория и практика параллельных вычислений, 2019-12-20. - Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 480 с. - Книга находится в базовой версии ЭБС IPRbooks. - ISBN 978-5-94774-645-7
5. Соснин, В. В. Введение в параллельные вычисления Электронный ресурс / В. В. Соснин, П. В. Балакшин. - Введение в параллельные вычисления, 2022-10-01. - Санкт-Петербург : Университет ИТМО, 2015. - 54 с. - Книга находится в премиум-версии ЭБС IPR BOOKS. - ISBN 2227-8397