УДК 681.3.066
ББК 32.973.26-018.2
Р89

Рецензент: А.В. Корочкин, к. т. н., доцент (кафедра вычислительной Техники Национального Технического Университета Украины «КПИ»)

Русанова О.В.

Р89 **Программное обеспечение компьютерных систем.- К.: «Корнійчук», 2003.- 94 с.**

ISBN 966-7599-27-2

Рассмотрены особенности построения программного обеспечения параллельных компьютерных систем, дополнительные трудности, которые возникают при этом и пути их преодоления. Изложены и проанализированы подходы к программированию, применяемые для различных параллельных систем. Рассмотрены основные языковые конструкции, описывающие как синхронные, так и асинхронные параллельные процессы. Приведены особенности алгоритмов компиляции для различных компьютерных систем. Рассмотрены алгоритмы автоматического распараллеливания программ. Особое внимание уделяется вопросам распараллеливания и векторизации циклов.

Для студентов, аспирантов и специалистов в области компьютерных систем.

ББК 32.973.26-018.2

Русанова О.В.

Р89 **Программне забезпечення комп'ютерних систем.- К.: «Корнійчук», 2003.- 94 с.**

ISBN 966-7599-27-2

Розглянуто особливості побудови програмного забезпечення паралельних комп'ютерних систем, додаткові труднощі, що виникають при цьому і шляхи їх подолання. Викладені і проаналізовані підходи до програмування, що застосовуються для різних паралельних систем. Розглянуті основні мовні конструкції, що описують як синхронні. так і асинхронні паралельні процеси. Подано особливості алгоритмів компіляції для різних комп'ютерних систем. Розглянуто алгоритми автоматичного розпаралелювання програм. Особлива увага приділяється питанням розпаралелювання і векторизаціє циклів.

Для студентів, аспірантів та спеціалістів з комп'ютерних систем.

ББК 32.973.26-018.2

ISBN 966-7599-27-2

© Русанова О.В., 2003

Русанова О.В.

Программное обеспечение компьютерных систем

Особенности программирования и компиляции

Киев
«Корнійчук»
2003

Содержание

I. Введение 3
1. Классификация компьютерных систем 3
2. Основные этапы выполнения вычислительных задач в компьютерных системах 11
II. Языковые средства описания параллельных процессов 14
3. Основные подходы к созданию параллельных языков программирования 14
4. Специфические проблемы параллельного программирования 17
5. Языковые средства описания синхронных параллельных процессов 20
5.1 Размещение массивов в памяти матричных КС 21
5.2 Способы выборки объектов массивов 26
5.3 Функции соответствия размерности массивов 28
5.4 Язык Parallaxis 30
6. Языковые средства описания асинхронных параллельных процессов 39
6.1. Описание параллельных процессов 40
6.2. Программные средства инициализации и завершения параллельных процессов 40
6.3. Языковые средства синхронизации доступа к общим ресурсам 44
6.4. Языковые средства синхронизации сообщений для МКМД–систем с распределенной памятью 48
7. Заключение 52
III. Особенности компиляции программ для КС 53
8. Традиционные этапы компиляции 53
9. Особенности компиляции при использовании различных способов программирования для параллельных КС 54
10. Способы распараллеливания процесса компиляции 55
11. Автоматическое распараллеливание программ 56
11.1 Распараллеливание линейных программ 57
11.2 Распараллеливание программ с ветвлениями 62
11.3 Распараллеливание и векторизация циклических участков программ 67
11.4 Разбиение программы на асинхронные параллельно выполняемые процессы 79
11.5 Распараллеливание программ для КС, управляемых потоком данных 85
Литература 93

I. Введение

1. Классификация компьютерных систем

Прежде всего рассмотрим важнейшие характеристики компьютерных систем (КС):

1. ***Синхронность***.

Если в КС все процессоры работают под управлением единого машинного такта, то такие системы относятся к классу синхронных. В случае отсутствия единого машинного такта – системы асинхронные. С точки зрения программного обеспечения КС данная характеристика является одной из важнейших.

2. ***Зернистость*** (уровень параллелизма).

Различают системы:

- крупнозернистые;
- средне зернистые;
- мелко зернистые.

3. ***Связность***.

Различают системы:

- слабосвязанные;
- связанные;
- сильносвязанные.

Эта характеристика (ее тип) определяется по скорости связи между процессорами.

4. ***Тип памяти***:

- разделяемая (общая);
- распределенная;
- разделяемо-распределенная.

5. ***Способ управления***:

- общее управления;
- распределенное управление.

6. ***Масштабируемость***.

Это способность к наращиванию процессоров в системе. Существуют КС с масштабируемостью:

- ограниченной;
- неограниченной.

7. ***Симметричность систем***.

Существует несколько определений симметричности систем. В соответствии с первым из них симметричная система все процессоры имеют равные права на доступ к памяти. В соответствии со вторым определением, в симметричной системе все процессоры равноправны с

точки зрения выполнения системных и пользовательских задач, а в несимметричной - одна часть процессоров предназначена для выполнения системных, а другая часть – для пользовательских задач.

Рассмотрим классификацию современных компьютерных систем (КС), представленную на рис.1.1. Различные типы КС будем различать по следующим признакам:

- По степени связанности.
- По типу управления.
- По типу параллельной обработки.
- По организации памяти.
- По типу взаимосвязи.
- По типу топологии.

По степени связанности различают три типа КС: параллельные КС (ПКС) (сильно связанные); кластерные мультикомпьютерные системы (связанные) и распределенные системы (слабосвязанные).

ПКС - это комплекс аппаратных и программных средств, предназначенных для эффективного выполнения вычислительной задачи. В состав ПКС входит множество, как правило, идентичных процессоров, либо процессор конвейерного типа, в котором производится совмещение обработки информации. В ПКС может применяться как разделяемая, так и распределенная оперативная память (ОП), взаимодействие между процессорами возможно либо через ОП, либо посредством специальных каналов. Также имеются общие каналы ввода-вывода (КВВ), внешняя память, как общая, так и раздельная. Работа происходит, как правило, под управлением единой операционной системой (ОС). Целью ПКС является повышение реальной пользовательской производительности системы.

Кластерные мультикомпьютерные системы (КМС) - это комплекс, состоящий из множества компьютеров, объединенных высокоскоростными коммуникационными средствами, представляющий собой единое целое для ОС, системного программного обеспечения, прикладных программ и пользователей. Как и ПКС, кластерные системы прежде всего предназначены для эффективного выполнения конкретной вычислительной задачи, т.е. для повышения реальной пользовательской производительности. Часто КМС называют более дешевым вариантом многопроцессорных систем с распределенной памятью, хотя, по сути они являются более совершенным продолжением многомашинных комплексов 60-х годов. В настоящее время наиболее распространенными коммуникационными средствами кластерных систем являются SCI, MyRinet, Fast Ethernet и Gigabit Ethernet, отличающихся ценой и техническими характеристиками.

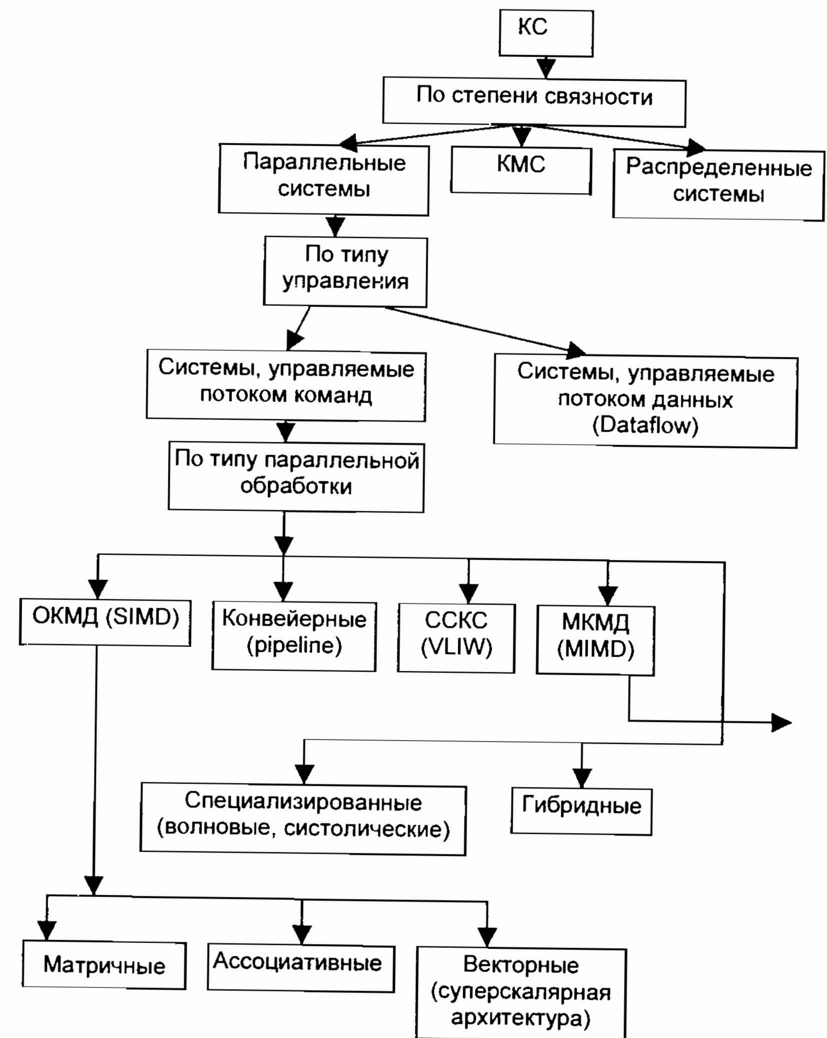


Рис. 1.1. Классификация компьютерных систем

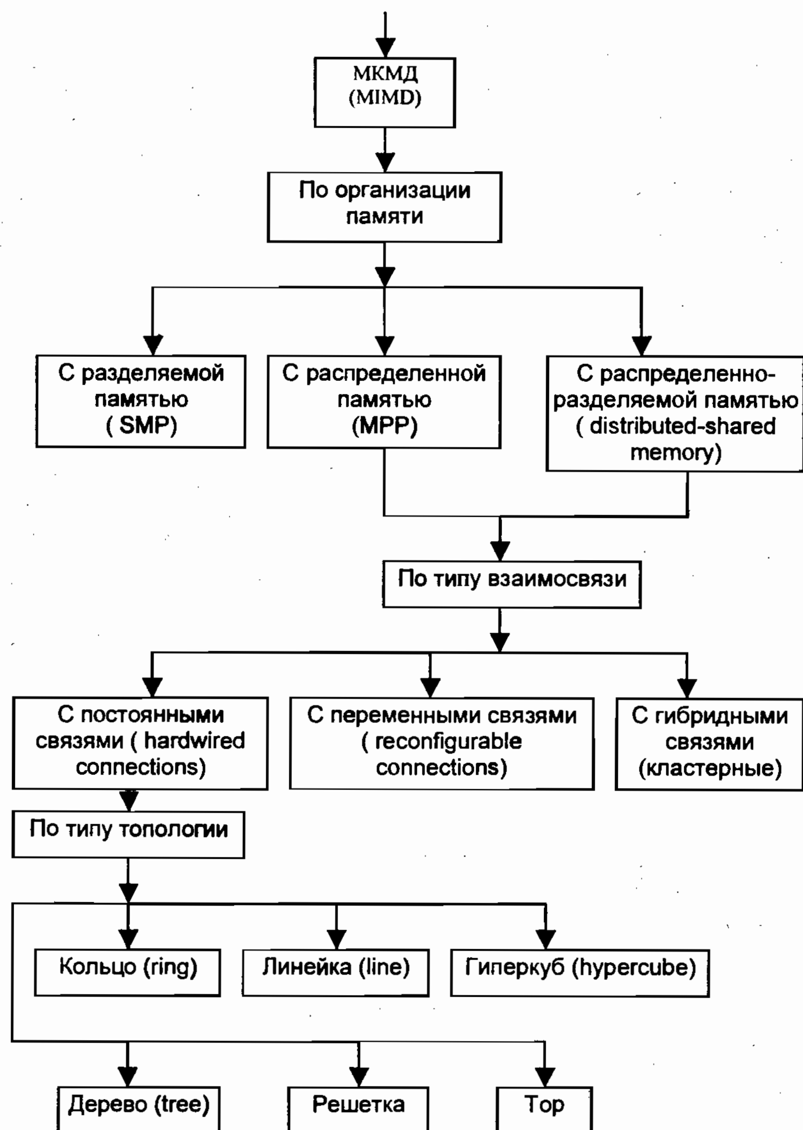


Рис. 1.1. Классификация компьютерных систем (продолжение)

Распределенные системы (РС) – это комплекс аппаратных и программных средств, предназначенных для эффективного одновременного выполнения множества вычислительных задач. Аппаратные средства представляют собой множество компьютеров (часто различных по производительности), объединенных между собой посредством либо сетевых средств связи, либо высокоскоростных коммутаторов, применяемых в кластерных системах. Можно считать, что ПКС и КМС являются частными случаями РС, т.к. они предназначены для эффективного выполнения единственной вычислительной задачи, в то время, как РС – для эффективного выполнения множества вычислительных задач. Таким образом, основной целью РС – является повышение системной производительности системы.

По типу управления ПКС могут быть разделены на два типа:

- Системы, управляемые потоком данных.
- Системы, управляемые потоком команд.

Системы, управляемые потоком данных (Dataflow systems) состоят из множества идентичных процессоров, с общим управлением и с общей (разделяемой) оперативной памятью. Из-за разделяемой памяти потоковые системы обладают ограниченной масштабируемостью. Эти системы предназначены для реализации мелкозернистого параллелизма (на уровне операций). Распараллеливание в этих системах осуществляется вначале, во время компиляции, а затем - динамически в процессе выполнения программы. Выполнение команд и соответствующих им операций производится в тот момент времени, когда для них готовы операнды т. е. данные диктуют порядок выполнения операций. Потоковые системы относятся к классу асинхронных. Преимуществом данных систем является простота программирования, поскольку на этапе программирования отсутствует проблема распараллеливания.

По типу параллельной обработки, ПКС, управляемые потоком команд, можно разделить на следующие классы:

- **ОКМД** (одиночный поток команд множественный поток данных)(SIMD – single instruction multiple data).
- **Конвейерные** (Pipeline).
- **Системы со сверхдлинным командным словом (ССКС)** (VLIW – very large instruction words).
- **МКМД** (множественный поток команд множественный поток данных) (MIMD – multiple instruction multiple data).
- **Специализированные** (проблемно-ориентированные, волновые, систолические).
- **Гибридные.**

Системы класса ОКМД – это синхронные ПКС, предназначенные, как правило, для реализации мелкозернистого уровня параллелизма. К ним относятся матричные, ассоциативные и векторные ПКС.

Матричные ПКС – это КС, состоящие из множества универсальных процессоров, предназначенных для полно разрядной либо для одноразрядной обработки информации. Работают все процессоры под управлением единого устройства управления, то есть в один и тот же момент времени все процессоры выполняют одну и ту же операцию над различными данными. Эти системы ориентированы на реализацию естественного параллелизма. Они относятся к системам с распределенной памятью, а значит процессоры могут быть объединены в различные топологии с неограниченной масштабируемостью. Наиболее распространенными топологиями для матричных систем являются решетка и тор.

Ассоциативные ПКС отличаются от матричных тем, что в качестве оперативной памяти используется ассоциативная память.

Векторный процессор (суперскалярная архитектура) состоит из множества одно функциональных конвейеров, которые используют общую, или разделяемую, память. Из-за разделяемой оперативной памяти масштабируемость ограничена. Эти системы ориентированы на реализацию как естественного параллелизма, так и параллелизма смежных операций, а в случае многопроцессорных вариантов векторных систем, состоящих из множества векторных процессоров, то и на реализацию параллелизма независимых ветвей (средне- и крупнозернистый параллелизм).

Конвейерные системы построены на базе многофункциональных конвейерных устройств. Такая система может содержать одно или множество таких устройств. Это синхронные системы. Конвейерные системы относятся к системам с общей, или разделяемой памятью и следовательно имеют ограниченную масштабируемость. Они ориентированы на параллелизм смежных операций, параллелизм микроопераций, естественный параллелизм (в частном случае) (мелкозернистые виды параллелизма).

Конвейерные системы можно разделить на три класса:

- *Конвейер с постоянным тактом.* Длительность такта определяется по самой сложной функции, которая может быть выполнена в данном конвейере (например, деление).
- *Конвейер со статической перестройкой такта.* Длительность такта определяется перед выполнением вычислительной функции, то есть статически. Такой конвейер в фиксированный момент времени работает как одно функциональный, несмотря на то, что по своим возможностям он является многофункциональным. Перестройка конвейера с одной функции на другую осуществляется только после его полного освобождения. В этот же момент переопределяется и длительность его такта.
- *Конвейер с динамической перестройкой такта.* Он является многофункциональным, длительность его такта определяется по самой сложной функции из тех, которые выполняются в данный момент в конвейере, то есть перестройка такта происходит динамически. Такие процессоры являются потенциально наиболее производительными, но и наиболее дорогими.

Системы со сверхдлинным командным словом (ССКС) состоят из набора универсальных процессоров, ориентированных на реализацию параллелизма смежных операций. Эти системы относятся к классу синхронных. В них используется разделяемая память и общее устройство управления. Система работает со специальными сверхдлинными командами, представляющими собой множества (вектор) команд (для каждого процессора - своя команда). Используется специальная широкоформатная оперативная память. Размеры команды достигают сотен разрядов. Эта система является системой с динамической перестройкой такта (длительность такта определяется по самой сложной команде, входящей вектор команд). Возможно неэффективное использование процессорных ресурсов из-за различных по сложности и, соответственно, по времени выполнения команд, входящих в одно и то же командное слово. Однако, если используются процессоры с RISC архитектурой, то данный недостаток устраняется. Обладают ограниченной масштабируемостью.

МКМД системы– это многопроцессорные системы, состоящие из множества универсальных и идентичных процессоров с распределенным управлением, имеющие различную организацию памяти (разделяемую, распределенную, разделяемо-распределенную). Эти системы ориентированы на параллелизм независимых ветвей и параллелизма независимых задач (крупнозернистый параллелизм). Они относятся к классу асинхронных.

МКМД системы с разделяемой памятью (shared memory) называют также SMP(symmetric multiprocessor) системами, т.к. в данной системе все процессоры имеют равные права на доступ к памяти. В SMP системах используется три варианта топологий, таких как:

- Шинная.
- Многомодульная коммутация.
- Посредством матричных коммутаторов.

Основным недостатком данных систем является ограниченная масштабируемость, связанных с конфликтами по памяти.. На практике число процессоров в SMP системах не превышает 32.

МКМД системы с распределенной памятью (distributed memory) называют также MPP(massively parallel) системами. Эти системы не имеют ограничений на масштабирование и поэтому в настоящее время являются наиболее перспективными с точки зрения роста производительности. Основные недостатки данных систем связаны с чрезвычайной сложностью их эффективного использования. Необходимо решать сложные задачи создания прикладного и системного программного обеспечения, включающих выполнение распараллеливания, эффективного параллельного программирования с учетом синхронизации обменов сообщений, маршрутизации, распределения ресурсов и т.п. От качества решения перечисленных задач зависит величина реальной пользовательской производительности системы.

МКМД системы с распределенно-разделяемой памятью(distributed-shared memory)(DSM) – это гибрид SMP и MPP систем. DSM-системы представляют собой архитектуры, в которых память физически распределена, но логически (программно) общедоступна. Такая идеология поддерживается в NUMA (non-uniform memory access) системах. Масштабируемость данных систем ограничивается объемом адресного пространства, возможностями аппаратуры поддержки когерентности кэш памяти и возможностями ОС по управлению большим числом процессоров. На настоящий момент, максимальное число процессоров в NUMA-системах составляет 256 (Origin2000).

Системы, в которых память физически распределена по типу взаимосвязи можно разделить на три группы:

- Системы с постоянными (статическими) связями.
- Системы с переменными (динамическими) связями.
- Системы с гибридными связями (постоянные+переменные связи).

В системах с постоянными связями связь между процессорами осуществляется через каналы (link). В этом случае формируется *статическая конфигурация*. Недостатками такой конфигурации являются жёсткие требования к элементной базе - наличие множества каналов и контроллеров ввода/вывода. В противном случае возможно построение только шинной топологии.

В системах с переменными связями используются различные средства коммутации, такие как однокаскадные(single-stage), многокаскадные (multistage) и матричные(crossbar). В этом случае формируется *динамическая конфигурация (реконфигурируемая система)*. Из перечисленных коммутаторов наиболее производительными и одновременно дорогими являются матричные коммутаторы. Тем не менее они являются наиболее используемыми в настоящее время.

В гибридных системах процессоры связываются как через каналы, так и через коммутаторы. В этом случае формируется так называемая *кластерная конфигурация*. Подобные системы могут состоять из иерархии групп (кластеров) процессоров. Обычно внутри группы процессоров используются статические связи, а кластеры между собой связываются через коммутаторы. Внутри кластера процессоры не равноправны, есть в наличии host – процессоры, которые, как правило, и подключаются к коммутаторам.

Гибридные ПКС – это системы, которые включают себя комбинацию структурных решений, таких как векторная, потоковая, МКМД, ССКС и конвейерная организация.

2. Основные этапы выполнения вычислительных задач в компьютерных системах

Рассмотрим шесть основных этапов подготовки к выполнению вычислительных задач в КС:

1. Физическая постановка задачи.
2. Математическая постановка задачи.
3. Выбор (разработка) параллельного численного метода.
4. Программирование.
5. Компиляция.
6. Организация параллельных процессов в рамках операционной системы (ОС).

Наилучший результат адаптации вычислительной задачи к вычислительным средствам достигается тогда, когда работа проводиться комплексно, и с учетом архитектуры КС. Однако,на практике работы выполняются, как правило, последовательно с вынужденными итеративными повторениями некоторых этапов работ. Именно этим и можно объяснить появление в 70-х годах гипотезы Минского (рис.2.1.), в соответствии с которой, производительность КС с ростом количества процессоров растёт, не по линейному, а по логарифмическому закону. Причинами такого роста является:

- Объективная причина, связанная с последовательностью выполнения задачи и пересылкой данных.
- Качество выполнения основных этапов выполнения задачи и пересылок данных.

КС 70-х годов подтверждали эту гипотезу, так как выполнение перечисленных этапов проводилось чисто последовательно, ОС таких систем практически не претерпели изменений, их производительность действительно резко падала. Для получения повышенной реальной пользовательской производительности необходимо выполнять этапы 3 – 6 комплексно. Все эти этапы должны быть выполнены с учетом специфики архитектуры и особенностей КС, (число процессоров, объем памяти и т.

д.), поскольку тип КС определяет эффективность тех или иных численных методов решения задачи, требования к языкам программирования и соответственно способ программирования, функции трансляторов, требования к конфигурациям и функциям ОС.

Так, например, для МКМД систем программист должен создать такую программу, при реализации которой на протяжении всего времени существовало бы n или более независимых процессов (n – число процессорных элементов (ПЭ) в КС), причем более предпочтительным вариантом является слабая связанность процессов. Более того, пользователь может (если позволяют конструкции языка программирования) создавать файлы конфигурации, в которых производится распределение процессов по ПЭ. Для этого он должен, с учетом интервалов времени выполнения отдельных процессов на ПЭ, оптимизировать график работ в интересах сокращения времени решения задач. Это сложнейшая задача по организации вычислений является практически не выполнимой в случае большого числа процессоров (для MPP систем). Эту задачу целесообразно решать в рамках ОС или специального системного ПО, выполнять анализ и планирование алгоритмов на КС. Реализация этой задачи - одна из главных задач ОС КС, с точки зрения получения высокой реальной пользовательской производительности.

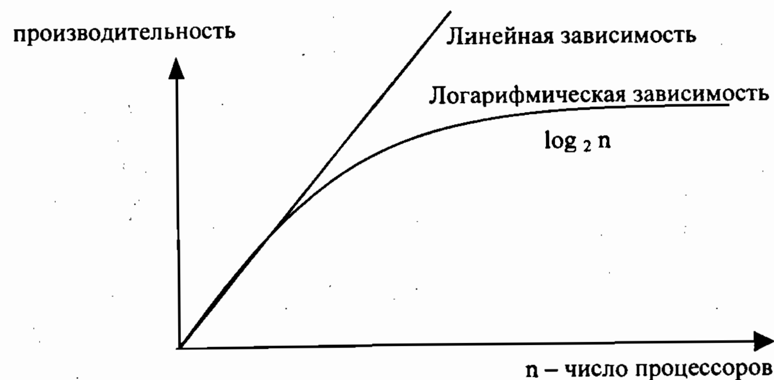


Рис.2.1. Гипотеза Минского.

Несколько проще задача программирования для пользователей таких систем, как конвейерные, ССКС и управляемые потоком данных. Они могут использовать традиционные языки программирования такие же, как и однопроцессорные КС. То есть тут проще выполняется 4 этап, но

усложняется 5 этап, поскольку компиляция должна включать функцию автоматического распараллеливания вычислений.

Понятно, что производительность КС возрастает пропорционально возрастанию загруженности вычислительных средств. Комплексное выполнение всех этапов ведет к повышению загруженности, однако, на самом деле,100% занятость реально не может быть достигнута. Всегда присутствуют определенные временные потери, связанные с обеспечением правильности хода вычислений, а также с необходимостью обмена данными между процессорами. Поэтому дальнейшее повышения степени загруженности КС возможно за счет выполнения множества вычислительных задач, если данная задача не может полностью загрузить все процессоры. За счет этого может быть достигнута максимальная занятость (эту ситуацию Минский не рассматривал), способа распараллеливания вычислительного алгоритма (соответственно полученной степени распараллеливания), способа распределения вычислительных работ по процессорам, а значит от качества выполнения 3-6 этапов.

Из всего сказанного ясно, что с одной стороны ПО КС значительно влияют на реальную пользовательскую производительность КС, а с другой – типы архитектуры КС влияют на их программное обеспечение.

Так матричные, векторные КС, SMP и MPP системы оказали существенное влияние на параллельные языки программирования и прикладные программы (а конвейерные, потоковые - нет). Векторные, матричные, конвейерные, потоковые, SMP и MPP системы оказали существенное влияние на трансляторы, а SMP и MPP системы - на операционные системы (в них наиболее сложно решаются вопросы синхронизации, диспетчеризации, маршрутизации и т.п.). А в векторных, матричных, конвейерных, синхронных системах на этапе трансляции и программирования производится адаптация вычислений к КС, размерность задачи приводится к размерности системы.

Таким образом, основной целью ПО КС является обеспечение максимальной пользовательской производительности за счет оптимального решения перечисленных выше проблем. Критерий оптимизации - минимальное время решения вычислительных задач. Данная цель может быть достигнута за счет создания эффективных методов и средств распараллеливания вычислительных алгоритмов, программирования , компиляции, а также распределения вычислительных работ по процессорам. Перечисленные методы и средства составляют основу для совершенствования языковые средства, компиляторов и ОС КС.

Данный курс будет состоять из 4-х частей: в 1-ой части мы будем рассматривать особенности языков программирования, описывающий

параллельные алгоритмы. Здесь, однако, мы постараемся рассматривать специфику программирования, продиктованную особенностями различных архитектур КС. Во 2-ой части рассмотрим особенности трансляции в ПКС. В 3-ей части рассмотрим основные механизмы ОС параллельных КС. Будем уделять основное внимание вопросам планирования и диспетчеризации для КС с различной архитектурой. При этом будем рассматривать алгоритмы планирования на статическом уровне и на динамическом уровне. На динамическом уровне рассмотрим алгоритмы балансировки загрузки КС. В 4-ой части рассмотрим примеры прикладного программного обеспечения для КС.

II. Языковые средства описания параллельных процессов

3. Основные подходы к созданию параллельных языков программирования

Появление параллельных КС привело к усложнению программирования на этих системах. Алгоритмы и структуры данных, которые разрабатываются для традиционных однопроцессорных систем, не могут быть слепо перенесены на КС параллельной архитектуры без потери эффективности в использовании процессоров и ограниченного повышения производительности. Таким образом, возникла задача создания новых параллельных алгоритмов и языков программирования, позволяющих описывать параллельные процессы. Однако, к моменту создания параллельных КС уже имелось огромное количество эффективных последовательных численных алгоритмов и ПО, реализующих эти алгоритмы и полностью отказаться от них было бы не разумно. Тем более, что для некоторых архитектур параллельных КС предусматривается автоматическое распараллеливание программ в процессе их выполнения.

Существует два пути программирования.

1. Адаптация программ, написанных для традиционных компьютеров к архитектуре ПКС.
2. Создание новых языковых средств параллельного программирования.

В первом случае функция распараллеливания выполняется в процессе трансляции. В этом случае путем моделирования исследуются имеющиеся программы (для выполнения определенной вычислительной задачи) с целью проверки конфликтов и уровня приспосабливания этих программ к параллельной обработке на заданной архитектуре. Здесь производится проверка отношений между элементами программы и тем самым

производится распараллеливание. В результате осуществляется выбор оптимальной программы для заданной архитектуры ПКС. Как правило, такое моделирование производится в процессе трансляции программы. Таким образом, в данном случае мы наблюдаем тесную связь реализации 4 и 5 этапов (программирование и трансляция программы) работ для ПКС. Тут результат 4 этапа влияет на результат 5 и наоборот, то есть 4 и 5 этапы выполняются совместно до тех пор, пока не будет найдена оптимальная программа и ее параллельное представление (в процессе трансляции). Проблемы автоматического распараллеливания последовательных программ мы будем рассматривать позже во 2-ой части курса, когда будем изучать проблемы трансляции параллельных КС. Примеры применения такого программирования - это конвейеры, системы со сверхдлинным командным словом, системы, управляемые потоком данных.

Во втором случае для всех остальных систем, существует три подхода к созданию языков параллельного программирования:

1. Расширение традиционных языков средствами параллельной обработки (параллельный Си, параллельный Fortran, параллельный Паскаль и др.).
2. Создание параллельных языков, ориентированных на конкретную архитектуру ПКС (ОККАМ, ВЕКТОР, Эль-76 для Эльбруса), конструкции которых могли бы эффективно транслироваться в систему команд конкретных типов ПКС.
3. Создание универсальных языков параллельного программирования, которые не ориентированны на конкретную архитектуру КС (Ada).

Первый подход широко применяется, благодаря следующим преимуществам:

- наиболее простой в применении, поскольку такие языки легко осваиваются, являясь расширениями уже известных и широко используемых языков программирования.
- дешевый подход, т.к. имеется возможность использования ранее разработанных программ.

Однако при реализации данного подхода известные языки необходимо дополнить новыми средствами, такими как описание параллельных процессов, методы синхронизации доступа к разделяемому ресурсу, методы синхронизации при обмене сообщениями. Такие средства, не свойственные идеологии создания традиционных языков программирования, могут быть не эффективными, механизмы синхронизации, как правило, не автоматизированы, что ведет к снижению надежности программ. Таким образом, необходимость сохранения концепций используемого языка, часто мешает, а в некоторых случаях затрудняет создание языковых средств параллельного программирования.

Преимуществами специализированных языков на базе второго подхода являются:

- высокая эффективность программного кода;
- полное соответствие структуры параллельной программы размерности и архитектуре системы.

Такие языки позволяют разрабатывать наиболее эффективные программы. Однако слишком дорого для каждой архитектуры создавать узкоспециализированные языки, с низкой эффективностью переноса программ на другие типы ПКС и необходимостью изучения множества специализированных языков пользователями.

Универсальные параллельные языки в рамках третьего подхода имеют следующие преимущества:

- высокая эффективность переноса программы с одной параллельной архитектуры на другую (изменяют только файл конфигурации, распределение по процессорам, подстройка под топологию);
- возможность введения новых конструкций, повышающих эффективность языка программирования и надежность написания на нем программ (по сравнению с первым подходом).

Недостатками таких языков являются:

- непроверенность новых концепций;
- зачастую сложность создания эффективных объектных программ при трансляции (по сравнению со вторым подходом);
- необходимость изучения пользователем не только синтаксиса языка, но и заложенных в него концепций.

Первый подход был реализован для большинства параллельных КС: ASC SYSTEM, ПС-3000, транспьютерные системы. Второй подход был реализован при создании языков “Вектор ” для ПС-2000, Эль-76 для Эльбруса, Оккам для транспьютерных систем . Ярким представителем третьего подхода является ADA, Modula-2, CSP, PLITS, SR, Jawa.

Рассмотрим влияние структуры и архитектуры КС на языковые средства параллельного программирования при помощи таблицы.

Таблица применения языковых средств параллельного программирования.

Тип ПКС	Языковые средства				
	Синхронная параллельная обработка	Асинхронная параллельная обработка			
	Операции над массивами	Описание процессов	Инициализация и завершение параллельных процессов и их синхронизация	Синхронизация доступа к разделяемым ресурсам	Обмен сообщениями и их синхронизация
Векторн. и матричн. КС	+	-	-	-	-
МКМД системы с общей (разделяемой) памятью	- +	+	+	+	-
МКМД с распределённой памятью	- +	+	+	+ -	+

4. Специфические проблемы параллельного программирования

Анализируя приведенные в таблице новые языковые средства, перечислим специфические для параллельного программирования понятия и проблемы.

В традиционном программировании, в отличие от параллельного, программе, как правило, соответствует единственный процесс. В параллельном программировании программе может соответствовать множество процессов (каждый процесс- это, например, ветвь алгоритма).

Пересылки могут осуществляться через оперативную память (SMP) или с помощью пересылки сообщений (MPP).

Если в SMP системах различные процессы используют одни и те же данные, то в этом случае существует понятие критического участка (области) программы, которая должна выполняться с исключительным правом доступа к разделяемым данным, на которые имеются ссылки в этой

программе. Процесс, готовящийся войти в критическую область, может быть задержан, если любой другой процесс в это время находиться в критической области. Одним из средств организации критической области является семафор. Кроме того, данные одних процессов необходимо передавать другим. При этом существует такое понятие, как ожидание события (то есть изменение состояния семафора).

В МРР системах при обмене данными с помощью пересылки сообщений следует координировать выполнение нескольких ветвей. При этом существует понятие барьера. Существуют барьеры нескольких типов. При использовании барьера одного типа предполагается, что все процессы некоторой группы должны достичь определенной точки своей программы (барьера), прежде чем любому из них будет разрешено двинуться дальше. Вариантом организации барьера может быть однониточная следующая за барьером критическая секция из одной ветви, которую выполняет любой (но только один) процесс из группы процессов. Модернизация этого типа барьера заключается в разрешение только последнему из прибывших к барьеру процессу выполнять критическую секцию из одной ветви.

Издержки от пересылок является возникновение обратимых и необратимых блокировок (тупиков). Обратимая блокировка (Livelock) означает ситуацию, когда два и более параллельных процессов зацикливаются,, т.е. обмениваются сообщениями , не выполняя при этом никакой полезной вычислительной работы. Процессы находятся в таком состоянии до тех пор, пока не исчерпают выделенные им ресурсы. Необратимая блокировка (Deadlock) – это ситуация, когда два или более процессов ожидают события, которое должно произойти в процессе, находящемся в том же заблокированном состоянии.

Для параллельных программ возможна ситуация недетерминированности результата. Это означает, что при одних и тех же исходных данных параллельная программа может давать различные результаты. Это может происходить из-за разброса времен исполнения ветвей программы, возникающего в условиях так называемых гонок.

Рассмотрим ситуацию недетерминированности, возникающую при выполнении следующей программы на Fortran

```
Do 100 I=1,N
CREATE CALC(I)
100 CONTINUE
```

В каждом процессе CALC используется формула (B-A)/N*(I-1)+A

Когда такая программа выполнялась на традиционном компьютере, проблем не было, но при использовании ПКС, результаты были каждый раз разные, сколько бы раз не выполняли программу. Более того, ни один ответ не был правилен. Почему это произошло? Выяснилось, что причиной некорректного поведения программы было непреднамеренное

распределение данных, которое возникало в операторе CREATE. В FORTRAN все параметры передаются по ссылке, а это значит, что адрес переменной I передается при каждом вызове подпрограммы CALC и таким образом, становится общим для всех N процессов и исходного процесса. При создании каждого процесса происходит обращение к I , однако, в то же самое время, исходный процесс производит изменение переменной I как часть цикла DO. В зависимости от точности синхронизации и планирования процессов, любой процесс мог получить либо предназначенное для него I либо более позднее I . Для устранения этой ошибки откорректируем:

```
DO 100 I=1,N
IARG(I)=(I)
CREATE CALC(IARG(I))
100 CONTINUE
```

Здесь мы вводим дополнительный массив IARG , элементами которого будут все значения I . Когда они будут заполнены, будут созданы подпрограммы CALC со своими аргументами.

Таким образом, разработка ПО ВС связана с решением следующих проблем:

1. Исключение взаимных блокировок и бесполезных обменов сообщениями.
2. Предохранение от нежелательных условий состязания.
3. Избежание образования слишком большого числа параллельных процессов.
4. Обнаружение завершения программы простейшим способом.

Перечислим также ряд новых вопросов, возникающих при проектировании параллельных программ:

1. Размер отдельных компонент (например, ветвей) параллельной программы.
2. Соотношение размерности задачи и системы (отображение виртуальных процессоров на физические)
3. Какой способ синхронизации применять лучше?
4. Какие данные следует сделать общими для всех процессов?
5. Как гарантировать детерминированность результатов?
6. Как разделить общие данные на части, чтобы обеспечить наиболее эффективное использование оборудования параллельных КС?
7. Если необходимо, выполнить распределение по процессорам и маршрутизацию.
8. Ввод-вывод данных.

Критерии оценки параллельных программ.

1. Коэффициент ускорения.

2. Коэффициент загруженности (максимальное число занятых процессоров при решении задачи).
3. Издержки на синхронизацию.
4. Минимизация времени на обмен данными.
5. Процент простоя системы.
6. Временные издержки на пересылки и маршрутизацию.
7. Масштабируемость программы.
8. Влияние размера задачи на ускорение.

5. Языковые средства описания синхронных параллельных процессов

Операции над массивами как средство укрупнение объектов, над которыми производятся вычисления, появилось в языках программирования задолго до создания векторных и матричных систем. Но только с появлением этих систем, операции над массивами в языках программирования стали средством ускорения вычислений. В традиционных языках программирования типы данных описываются как скаляры или как индексированные множества скаляров, называемые массивами. Это средства для удобного и сокращенного написания программ, имеющих в своем составе набор идентичных операций над разными данными.

Существует три метода описания массивов для параллельных КС.

1. Массивы как последовательные объекты. В этом случае массивы описываются как последовательность скаляров. При обращении к имени массива по умолчанию производится последовательная обработка его элементов. Для параллельной обработки элементов массива в теле программы указывается набор индексов, по которым и производится параллельная обработка. Пример такого метода: расширенный FORTRAN для системы STAR-100.

2. Массивы как параллельные объекты данных (альтернативный первому методу). Обработка по всем элементам этого массива выполняется параллельно (по умолчанию). Этот подход является наиболее общим и рассматривается как вариант в любом стандарте на обработку массивов. Здесь все массивы описываются как особые объекты данных заданной размерности. Любое обращение к массиву будет подразумевать весь массив. Однако этот подход не отменяет использование массива, как набора последовательных данных, когда это необходимо, для этого используют механизм индексации для понижения ранга посредством выборки. Например, если задан трехмерный массив А, то ранг равен трем и любое обращение к одному имени А подразумевает параллельное обращение ко всем его элементам. Индексация любой размерности может

быть рассмотрена как операция выборки, которая понижает ранг массива А на единицу. Если индексировать все три размерности , то из А выбирается скалярная величина или объекты данных ранга ноль. Этот подход был предложен в расширенном FORTRAN для CRAY-1.

3. Массивы как смесь последовательность и параллельных объектов. Это компромиссный вариант. В этом случае ограничивается число размерностей и общее количество элементов массива. Ограниченные данные полностью зависят от физической структуры КС, на которой будет выполняется обработка данного массива. В случае необходимости обработки массивов, размерность которых соответствует размерности системы (меньше или равно), выполняется параллельные вычисления. В случае, когда размерность массива превышает размерность системы, выполняется параллельно-последовательная обработка. Этот подход был адаптирован для многих процессорных матриц (матричных КС), где число размерностей, по которым можно осуществлять параллельное обращение, соответствует размеру и конфигурации самой КС. Этот подход реализован в языках (ACTUS, CFD и др.) для ILLIAC-4. Так, язык CFD оперирует одномерными массивами из 64-х элементов, которые преобразуются в матрицу для процессоров ILLIAV-4. Этот же подход реализован в DAP-FORTRAN, где параллельное обращение к массиву можно производить либо по первой , либо по первой и второй размерностям. И эти размерности должны соответствовать размеру КС DAP.

5.1 Размещение массивов в памяти матричных КС

Как известно, матричные КС относятся к системам с распределенной памятью. Размещение массивов в памяти данных систем влияет на показатель реальной производительности (пользовательской, производительности, при решении конкретной задачи). При выполнении матричных операций стандартными способами размещения элементов матриц в памяти МКС является размещения по строкам и по столбцам. Для параллельного выполнения большинства матричных задач необходимо осуществить одновременную выборку как строк матриц, так и их столбцов. Упомянутые способы размещения не обеспечивают этого требования. При размещении матриц по столбцам возможна параллельная выборка элементов строк и последовательная выборка элементов столбцов, а при размещении по строкам наоборот – параллельная выборка элементов столбцов и последовательная выборка элементов строк. Для одновременной выборки как столбцов, так и строк матрицы удобным способом является «скошенный» метод, приведенный ниже.

ПЭ0	ПЭ1	ПЭ2	ПЭ4
a_{11}	a_{12}	a_{13}	a_{14}
a_{24}	**a_{21}**	a_{22}	a_{23}
a_{33}	a_{34}	**a_{31}**	a_{32}
a_{42}	a_{43}	a_{44}	**a_{41}**

Этот способ позволяет минимизировать время выполнения матричных задач. Для обеспечения его работоспособности в системе должна быть предусмотрена возможность использования фигурной адресации, либо структура матричной системы должна иметь вид, представленный на рис.5.1.1.

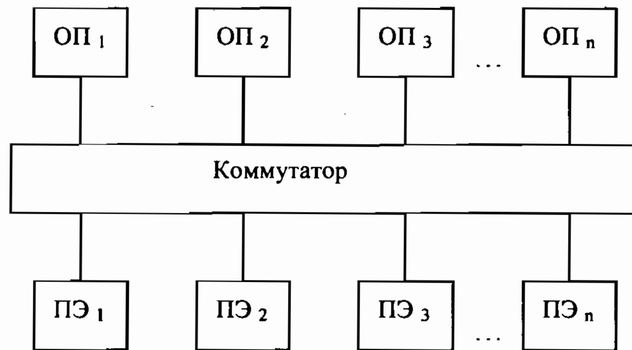


Рис. 5.1.1. Структура матричной КС

Немаловажным фактором повышения пользовательской производительности является минимизация времени, затрачиваемого на пересылку данных. Уменьшения этого времени возможно за счет адаптации алгоритма вычислений к структуре МКС. Рассмотрим пример выполнения задачи перемножения двух матриц, т.е. C=A*B, где A,B,C – матрицы размерностью n*n, а n≤64 в системе ПС – 2000 (рис.5.1.2.). В этом случае

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j} * b_{j,k} \quad (i,k = 1,2,\ldots,n) \; (1)$$

Известно, что в системе ПС-2000 минимизация времени, затрачиваемого на обмен данными, связано с использованием локальных пересылок по быстрым регулярным каналам (РК). Использование же пересылок по последовательному магистральному каналу (МК) приводит к увеличению времени пересылок данных.

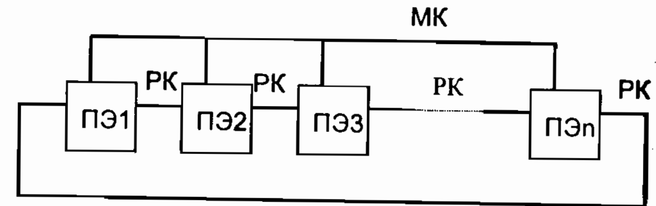


Рис.5.1.2. Структура системы ПС-2000

Будем размещать элементы матриц A, B и C в память ПЭ по столбцам, т.е. в первом ПЭ – 1-е столбцы матриц, во втором ПЭ – 2 -е столбцы матриц и т.д.

ПЭ1 ПЭ2 ПЭn

$$\begin{pmatrix} a_{11} & b_{11} & c_{11} \\ a_{21} & b_{21} & c_{21} \\ \vdots & \vdots & \vdots \\ a_{n1} & b_{n1} & c_{n1} \end{pmatrix} \quad \begin{pmatrix} a_{12} & b_{12} & c_{12} \\ a_{22} & b_{22} & c_{22} \\ \vdots & \vdots & \vdots \\ a_{n2} & b_{n2} & c_{n2} \end{pmatrix} \quad \begin{pmatrix} a_{1n} & b_{1n} & c_{1n} \\ a_{2n} & b_{2n} & c_{2n} \\ \vdots & \vdots & \vdots \\ a_{nn} & b_{nn} & c_{nn} \end{pmatrix}$$

Тогда выполняя вычисления по формуле (1), получим следующую схему вычислений:

$$
\begin{array}{cccc}
\text{ПЭ1} & \text{ПЭ2} & & \text{ПЭn} \\
\begin{bmatrix} a_{11}\,b_{11} \\ a_{12}\,b_{21} \\ \ldots \\ a_{1n}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{11}\,b_{12} \\ a_{12}\,b_{22} \\ \ldots \\ a_{1n}\,b_{n2} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{11}\,b_{1n} \\ a_{12}\,b_{2n} \\ \ldots \\ a_{1n}\,b_{nn} \end{bmatrix} \\
C_{11} & C_{12} & & C_{1n} \\
\begin{bmatrix} a_{21}\,b_{11} \\ a_{22}\,b_{21} \\ \ldots \\ a_{2n}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{21}\,b_{12} \\ a_{22}\,b_{22} \\ \ldots \\ a_{2n}\,b_{n2} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{21}\,b_{1n} \\ a_{22}\,b_{2n} \\ \ldots \\ a_{2n}\,b_{nn} \end{bmatrix} \\
C_{21} & C_{22} & & C_{2n} \\
\ldots & \ldots & & \ldots \\
\begin{bmatrix} a_{n1}\,b_{11} \\ a_{n2}\,b_{21} \\ \ldots \\ a_{nn}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{n1}\,b_{12} \\ a_{n2}\,b_{22} \\ \ldots \\ a_{nn}\,b_{n2} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{n1}\,b_{1n} \\ a_{n2}\,b_{2n} \\ \ldots \\ a_{nn}\,b_{nn} \end{bmatrix} \\
C_{n1} & C_{n2} & & C_{nn}
\end{array}
$$

Использование данной схемы вычислений с учетом размещения матриц в ПЭ приводит к большим временным затратам на обмен данных между ПЭ. Здесь используются как быстрые РК, так и медленный МК. С целью уменьшения временных затрат на обмен данными соотношение (1) преобразуем к виду:

$$c_{i,k} \quad \sum_{j\quad 1}^{n} a_{i,t} * b_{t,k} \ . \qquad (2)$$

где $t \quad (r \quad j) \bmod n$ (r - номер ПЭ, $r = 1,2,\ldots,n$).

Это приводит к циклическому сдвигу на r позиций частичных произведений при их суммировании в каждом r-м ПЭ.

Схема вычислений с использованием соотношения (2) имеет следующий вид:

$$
\begin{array}{cccc}
\text{ПЭ1} & \text{ПЭ2} & & \text{ПЭn} \\
\begin{bmatrix} a_{11}\,b_{11} \\ a_{12}\,b_{21} \\ \ldots \\ a_{1n}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{12}\,b_{22} \\ a_{13}\,b_{32} \\ \ldots \\ a_{11}\,b_{12} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{1n}\,b_{nn} \\ a_{11}\,b_{nn} \\ \ldots \\ a_{1\,n-1}\,b_{n-1\,n} \end{bmatrix} \\
C_{11} & C_{12} & & C_{1n} \\
\begin{bmatrix} a_{21}\,b_{11} \\ a_{22}\,b_{21} \\ \ldots \\ a_{2n}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{22}\,b_{22} \\ a_{23}\,b_{32} \\ \ldots \\ a_{21}\,b_{12} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{2n}\,b_{1n} \\ a_{21}\,b_{1n} \\ \ldots \\ a_{2\,n-1}\,b_{n-1\,n} \end{bmatrix} \\
C_{21} & C_{22} & & C_{n2} \\
\ldots & \ldots & & \ldots \\
\begin{bmatrix} a_{n1}\,b_{11} \\ a_{n2}\,b_{21} \\ \ldots \\ a_{nn}\,b_{n1} \end{bmatrix} &
\begin{bmatrix} a_{n2}\,b_{22} \\ a_{12}\,b_{21} \\ \ldots \\ a_{1n}\,b_{12} \end{bmatrix} & \ldots &
\begin{bmatrix} a_{nn}\,b_{nn} \\ a_{n1}\,b_{1n} \\ \ldots \\ a_{nn-1}\,b_{n-1n} \end{bmatrix} \\
C_{n1} & C_{n2} & & C_{nn}
\end{array}
$$

Таким образом, вторая схема вычислений отличается от первой порядком хранения элементов столбцов матриц в ПЭ, а также порядком вычисления частичных произведений, что приводит к минимизации времени пересылки данных за счет использования только регулярных каналов и за счет уменьшения общего числа пересылок.

5.2 Способы выборки объектов массивов

При матричных вычислениях возможны ситуации, когда необходимо обращение не ко всему массиву, а к его части. Например, выборкой из массива может быть вектор- строка, вектор- столбец, диагональ матрицы, матрица в трехмерном массиве. Могут понадобиться выборки и более сложных конфигураций, в том числе с использованием косвенной адресации. Рассмотрим различные способы выборки элементов массива:

1. ***Выборка объектов пониженного ранга.***

Если задан массив A размерности (или ранга) n, то индексация по любой из размерности, понижает ранг на единицу.

A(*,*); A – параллельная выборка всех элементов.

A(I, *); A(I,) – параллельная выборка i-й строки;

A(*, J): A(,J) – параллельная выборка j-го столбца.

A(I,J) – обращение к одному элементу.

2. ***Выборка диапазона значений.***

В данном случае вместо понижения рангов объектов массива уменьшается размер этого объекта. Поэтому диапазон должен специфицировать поднабор всего диапазона размерности. Возможны следующие варианты спецификаций:

a) Отрезки допустимой области изменения индекса по соответствующему измерению, который может задаваться нижней и верхней границей изменения индекса. В этом случае шаг изменения индекса равен 1. Например, A(1:3, 2:7).

b) Последовательность диапазона изменения индексов с заданным шагом. Шаг изменения индексов может быть задан различными способами. Наиболее известные из них приведены в примерах 1 и 2. Пример 1: A(1:5:2, 2:8:2). В данном случае по каждому измерению массива указывается нижняя граница, верхняя граница и шаг изменения индекса. Таким образом в данном примере происходит выборка элементов массива A со следующими значениями столбцов: 2,4,6,8 и строк: 1,3,5. Пример 2: A(1:(2)5, 2:(2)8). В данном случае шаг изменения индексов указывается в скобках. В этом примере указывается выборка тех же элементов массива A, что и в прмере 1. Различие состоит лишь в синтаксисе.

Кроме того возможны два способа формирования индексов:

- **Декартовый:** заранее определяется скорость изменения индексов. Например, быстрее всего изменяется левый индекс.

Например, пусть A - матрица 10*10 элементов, тогда A(1:3,5:9) - под массив, состоящий из 15 элементов: A(1:5), A(2:5), A(3:5), A(1:6), …A(3:9),

- **Синхронный**: равная скорость изменения индексов по всем измерениям. A <1:10, 1:10>-задается главная диагональ,

A <1:9, 2:10>-задается нижняя диагональ, расположенная под главной диагональю.

3. ***Выборка элементов массива с помощью целочисленных векторов.*** Здесь используется косвенная адресация, которая задается, как правило, с помощью специальных векторов. Пусть заданы целочисленные массивы IVи JV вида

IV→ 1,1,1,3 JV→2,1,3

а также двумерный массив A вида

$$A \to \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ k & l & m & n \\ o & p & r & s \end{bmatrix}$$

Тогда

$$A(IV, *) = \begin{bmatrix} a & b & c & d \\ a & b & c & d \\ a & b & c & d \\ c & g & m & r \end{bmatrix}$$

$$A(*, IV) = \begin{bmatrix} a & e & k & o \\ a & e & k & o \\ a & e & k & o \\ c & g & m & r \end{bmatrix}$$

$$A(IV,JV) = \begin{bmatrix} b & a & c \\ b & a & c \\ b & a & c \\ l & k & m \end{bmatrix}$$

При синхронной выборке:

A <IV, *> → a b c n

A <*, IV > →a e k n

4. ***Выборка с помощью булевых массивов.*** Для выборки элементов используются логические массивы. Выборка осуществляется в зависимости от значения истинности логических элементов заданного массива. Таким образом, I-ый элемент может быть выбран из одномерного

массива посредством индексации его одномерным булевым массивом со значением "истинно" в I-ом элементе. Аналогично можно использовать такой логический массив для выборки строки или столбца из двумерного массива. Пусть задан логический массив вида

LV → 0 0 1 0

Рассмотрим примеры выборки с помощью массива LV из матрицы A, описанной выше:

A(LV,*) → k l m n
A(* , LV) → c g m r
A(LV,LV) → m.

Кроме выборки массива с помощью булевого вектора, в языках параллельного программирования для синхронных КС существует аналог оператора условного оператора при выполнении векторной операции. Этот оператор определяет необходимость выполнения операции над тем или иным элементом массива в зависимости от результата предшествующих вычислений. Для выборки с помощью булевых массивов существует оператор IF. В языке Fortran он обозначается как IFA (чтобы не путать с классическим оператором IF) и представляется в виде IFA(L)e , где L – условие, e – исполнительный оператор. Для выполнения этого оператора массивы, входящие в L и e, должны быть одной и той же размерности. Например, пусть A,B,C,D,F-массивы одинаковой размерности. В операторе IFA(A.LT.B)C=D+F операция сложения элементов массивов D и F и присваивание полученной суммы соответствующим элементам C будет производится только в тех случаях, когда для соответствующих элементов массивов A и B логическое выражение A.LT.B истинно, то есть, если $a_{34}<b_{34}$, то $c_{34}=d_{34}+f_{34}$, если же $a_{34}>=b_{34}$, то значение c_{34} не измениться.

5.3 Функции соответствия размерности массивов

Основным правилом при параллельной обработке массивов является то, что два операнда должны иметь одинаковый ранг и одинаковую область в соответствующих размерностях. Ограничив представление операндов таким образом, можно осуществить гораздо больший контроль за появлением ошибок как во время компиляции, так и во время работы. Для обеспечения этого ограничения, в языках введены конструкции, которые дают возможность сжатия, расширения или переформирования массивов с целью их соответствия. Организация такого соответствия возлагается на программиста. Рассмотрим подробнее функции получения соответствия размерностей массивов.

В языках, предусматривающих параллельную обработку массивов, должны быть:

1. ***Функции сжатия*** (понижения ранга). Как было показано выше, ранг объекта массива может быть понижен посредством индексации или выборки. Другим способом, при котором он может быть понижен, является повторное использование двоичного оператора между всеми элементами в одной размерности массива. Он, безусловно, может быть описан на языке как последовательность операций, однако это не дает возможности использования параллелизма для понижения ранга. Например, сумма из N элементов может быть получена при параллельном выполнении шагов выражения log2N. Приведем перечень наиболее общих операций понижения ранга, где каждая имеет два параметра, массив и размерность, по которой будет выполнено понижение: SUM(A, k), PROD(A, k), AND(A, k), OR(A, k), MIN(A, k), MAX(A, k). Так SUM(A, k),означает функцию суммирования элементов массива A k-ой размерности.
2. ***Функции расширения*** (повышения ранга). Часто для достижения соответствия необходимо повысить ранг объекта. Повышения ранга необходимо в двух случаях: 1) при операциях между скалярами и векторами. 2) при операциях между векторами и матрицами. В первом случае, как правило, функция понижения ранга не применяется. Языки допускают ослабление правил соответствия для операций скаляр- вектор, то есть, имеется возможность произвольно смешивать скаляры и массивы. Во втором случае функции расширения сводятся к повторению вектора либо в качестве строки, либо в качестве столбца, чтобы добиться соответствия с матрицей. Примером оператора повышения ранга является XPND(A, k, N), в котором A – имя вектора; k – размерность по которой производится повторение (1 – по строке, 2 – по столбцу); N – число повторений.

Пусть имеется массив A → a b c d. Тогда:

	a	b	c	d
XPND(A, 1, 3) →	a	b	c	d
	a	b	c	d

	a	a	a
XPND(A, 2, 3)→	b	b	b
	c	c	c
	d	d	d

3. ***Функция переформирования массивов.*** Эти функции используются тогда, когда необходимо соответствие между массивами различных

размерностей, но содержащие одинаковое, общее число элементов (примером является алгоритмы быстрое преобразование Фурье).

Для выполнения данной функции в языке Fortran используется два оператора: DIMENSION и MAP.

Оператор DIMENSION определяет динамический диапазон массива, а оператор MAP используется для изменения формы массива. Например, пусть имеется линейный массив (вектор) A, состоящий из N элементов. Необходимо переформировать его в двумерный массив. Фрагмент такой программы можно представить в следующем виде:

```
DIMENSION A(N)
.
.
.
MAP A(N/2, 2).
```

В этом примере показано, что исходный вектор A из N элементов с помощью оператора MAP преобразован в двухмерный массив N/2*2. Такое преобразование корректно, когда N- четное. Подчеркнем, что обязательным условием такого преобразования является одно и тоже число элементом у исходного и результирующего массивов.

5.4 *Язык Parallaxis*

5.4.1 Основные характеристики

Это машинно-независимый язык создан на базе языка Modula-2. Программы на языке Parallaxis предназначены для работы на SIMD системах, кластерных, рабочих станциях или однопроцессорных компьютерах.(таких, как SIMD системы MP-1 И MP-2, кластерные станции , использующие PVM, Intel Paragon, однопроцессорные версии для почти всех UNIX систем:SUN SPARC- станций (OC Solaris), DEC-станциях, HP 9000, IBM RS 6000, IBM PC с LINUX. ПО Parallaxis охватывает компиляторы для параллельных и однопроцессорных КС, отладчики, множество прикладного ПО для различных приложений, особо для обработки изображений. Конструкции языка ориентированы на параллельную обработку задач с естественным параллелизмом. Основная особенность языка Parallaxis – это программирование на абстрактном уровне на виртуальной структуре SIMD-системы. Таким образом, каждая программа включает кроме описания алгоритма задание виртуальной SIMD–модель системы.

SIMD –модель включает основной процессор (предварительной обработки) фон-Неймановской архитектуры предназначенный для скалярной обработки и матрицу процессорных элементов (ПЭ) для

векторной обработки (рис.5.4.1). Матрица процессоров представляет собой множество идентичных ПЭ. Основной процессор связан с ПЭ с помощью шины. Связи между ПЭ могут быть произвольными и задаются в программе. Кроме того, в программе можно задавать множество виртуальных систем. Множество их зависит от количества и структур данных массивов, обрабатываемых в программе. Структуры SIMD – моделей совершенно не зависят от физической структуры системы. Перед выполнением происходит отображение виртуального массива на физическую структуру используемой системы. Если число виртуальных ПЭ больше числа физических ПЭ, то каждый физический ПЭ отвечает за множество виртуальных ПЭ и ОП этих виртуальных ПЭ считается единой.

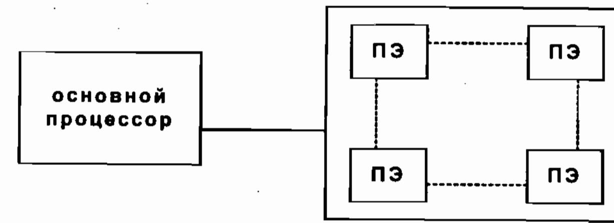


Рис.5.4.1. SIMD–модель системы

5.4.2 Структура программы

Программа состоит из главного модуля (который в свою очередь может вызывать множество других модулей) и имеет следующий вид:

```
MODULE <имя>
   <описание конфигурации виртуальной системы>
   <описание скалярных и векторных данных>
BEGIN
   <ввод векторных и скалярных данных>
   <операторы Parallaxis>
   <вывод результатов>
END.
```

5.4.3 Описание конфигурации

Оператор описания конфигурации предназначен для определения имени, количества и расположения ПЭ в виртуальной системе по аналогии

с операторами описания массивов и в общем виде представляется следующим образом:

CONFIGURATION <имя виртуальной системы><количество и расположение ПЭ>

Рассмотрим примеры операторов описания конфигурации:

CONFIGURATION list[1..4096]

В приведенном операторе описана конфигурация системы с именем *list*, представляющая собой вектор ПЭ (размерность один) из 4096 элементов.

CONFIGURATION grid[1..64][1..64]

В приведенном операторе описана конфигурация системы с именем *grid*, представляющая собой матрицу ПЭ (размерность два), которая включает 64 строки и 64 столбца.

CONFIGURATION hyper[0..1][0..1][0..1]

В приведенном операторе описана система с именем *hyper* представляющий собой гиперкубическую конфигурацию ПЭ размерностью три. В данном случае число элементов равно восьми с номерами: 000; 001; 010; 011; 100; 101; 110; 111.

5.4.4 Описание связей между ПЭ

Оператор описания связей между ПЭ виртуальной системы в общем виде может быть представлен следующим образом:

CONNECTION <имя связи>:<описание связей>.

При описании связей используются символы "→" и "↔". Первый из них означает однонаправленную связь, а второй – двунаправленную связь.

Существует четыре типа связей:

1) Один в один.
2) Один во множество.
3) Бинарные связи.
4) Многомерные (гиперкубические).

Рассмотрим примеры описания различных связей в виртуальных системах.

CONNECTION north : grid[i,j]→grid[i-1,j];
CONNECTION east : grid[i,j]→grid[i,j+1];
CONNECTION west : grid[i,j]→grid[i,j-1];
CONNECTION south : grid[i,j]→grid[I+1,j];

В приведенных выше операторах описываются однонаправленные связи конфигурации *grid* любой размерности в четырех направлениях (тип связи - один в один).

CONNECTION brd : gride[i,1]→grid[i, 2..32];

В приведенном примере описываются однонаправленные связи конфигурации *grid* первого элемента каждой строки с элементами 2-32 этой же строки (тип связи – один во множество).

Для описания бинарных связей рассмотрим конфигурацию виртуальной системы, состоящей из 15 ПЭ и организованной в виде бинарного дерева:

CONFIGURATION tree [1..15];

Тогда операторы описания бинарных двунаправленных связей имеют следующий вид:

CONNECTION l child : tree[i] ↔tree[2*i];
CONNECTION r child : tree[i]↔tree[2*i+1];

Рассмотрим пример описания двунаправленных многомерных (гиперкубических) связей для конфигурации *hyper*, представленной выше:

CONNECTION axis[1] : hyper[i,j,k]↔hyper[(i+1) mod 2, j,k];
CONNECTION axis[2] : hyper[i,j,k]↔hyper[i, (j+1) mod 2, k];
CONNECTION axis[3] : hyper[i,j,k]↔hyper[i,j,(k+1) mod 2];

5.4.5 Описание векторных и скалярных данных

В данном языке существует два типа данных:

- Скалярные;
- Векторные.

Скалярные данные располагаются на управляющем ПЭ. При описании векторных данных закладывается связь с ПЭ. Они распределяются между ПЭ виртуального массива. Таким образом, структура векторных данных

должна соответствовать структуре виртуального массива. Операторы описания векторных данных имеет следующий вид:

VAR<имя>:<имя конфигурации> OF <тип данных>;

Рассмотрим пример оператора описания векторных данных:

VAR Y : hyper of REAL;

В приведенном операторе описывается трехмерный массив данных Y типа REAL, расположенных в ПЭ виртуального гиперкуба.

Векторные данные для различных конфигураций могут одновременно существовать в одной и той же программе.

5.4.6 *Основные функции языка Parallaxis*

Существует набор функций , позволяющих определить позицию какого-либо ПЭ в виртуальном массиве и определенную информацию о виртуальном массиве. Перечислим некоторые из них:

1. DIM(config, axis). Эта функция имеет два аргумента: имя конфигурации и номер размерности. Она возвращает позицию каждого виртуального ПЭ внутри данной размерности.
2. LOWER(config, axis). Эта функция имеет те же аргументы, что и предыдущая и возвращает min значение данной размерности массива.
3. UPPER(config, axis). Эта функция имеет те же аргументы, что и предыдущая и возвращает max значение данной размерности массива.
4. LEN(config, axis). Эта функция с теми же агрументами, что и предыдущие. Она возвращает номер виртуального ПЭ в массивее.
5. LEN(config). Эта функция определяет общее число ПЭ, составляющих данную конфигурацию.
6. ID(config). Эта функция определяет номер каждого ПЭ в массиве.

5.4.7 *Параллельные операции*

Обращение по имени подразумевает параллельное выполнение операций над элементами векторных данных. Так, если x,y и z описаны как векторные данные одной и той же конфигурации, то оператор

x:= y+z;

означает одновременное суммирование значений y и z и присвоение полученного результата x для всех элементов этих данных.

Для того, чтобы выполнить операцию над ограниченным числом элементов векторных данных может быть применен условный оператор.

Пример 1: VAR X, Y, Z tree of REAL; -- массивы.

IF Z<>0.0 then X:=Y/Z;--эта операция параллельная над частями X, Y, Z.(ПЭ с нулевыми Z будут пассивными)

END;

В данном примере нулевые значения Z ограничивают выполнение операция X:=Y/Z над всеми массивами данных. Эта операция будет выполняться только для тех элементов X, Y, Z, где Z принимает ненулевые значения.

Пример 2: VAR X, Y, Z tree of REAL; -- массивы.

IF Z<>0.0 then X:=Y/Z;--эта операция параллельная над частями X, Y, Z.(ПЭ с нулевыми Z не будут выполнять эту операцию)

ELSE

X:= Y+Z; --это параллельная операция над частями X, Y, Z.(ПЭ с ненулевыми Z не будут выполнять эту операцию)

.

END;

В данном примере массивы данных X, Y и Z делятся на две активные группы: операция X:=Y/Z выполняется для тех элементов X, Y, Z, где Z принимает ненулевые значения, а операция X:= Y+Z для элементов X, Y, Z, где Z принимает нулевые значения. Здесь мы видим нетрадиционное выполнение оператора условного перехода.

Функции свертки

Данные функции выполняются только над векторными данными, в результате которых формируется скаляр. В общем виде такие функции можно представить следующим образом:

<scalar>:=REDUCE.(sum, product, min, max, and, or, first, last) (< имя векторного данного>) ;

где sum – сумма значений элементов векторных данных; , product - произведение значений элементов векторных данных; min и max - соответственно минимальное и максимальное значение элементов векторных данных; and и or – логические функции “И” и “ИЛИ” значений элементов векторных данных; , first и last – соответственно значения первого и последнего активных ПЭ.

5.4.8 *Операторы COMMUNICATION*

В языке Parallaxis существует два вида операторов коммуникации:

- Regular communication (Регулярные связи).
- General communication (Универсальные связи).

При использовании первого вида операторов предполагается использование стандартных связей между ПЭ, которые описаны в операторе конфигурации. При использовании второго не предполагается обязательное использование связей, описанных в конфигурации. Здесь применяют косвенная адресация (используются адреса – «куда переслать» и «откуда получить»). Операции пересылок могут быть использованы как для собственно передачи данных при выполнении прикладной задачи , так и для переупорядочивания массив данных (для осуществления различных вариантов их выборки).

Регулярные связи

Существует два варианта операторов:

- MOVE.< имя коммуникации>;
- SEND.< имя коммуникации>; или
  RECEIVE.< имя коммуникации>;

Отметим, что MOVE и RECEIVE являются функциями, а SEND – процедура. При использовании функции MOVE оба виртуальных ПЭ – источник данных и приемник данных должны находиться в активном состоянии, в то время, как при использовании функции RECEIVE и процедуры SEND в активном состоянии должен находиться лишь один виртуальный ПЭ, соответственно, для RECEIVE – ПЭ-приемник, а для SEND – ПЭ-источник.

Примеры регулярных связей:

CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;
IF ID(linear)<>2 THEN Y := MOVE.right(X); --ID – идентификатор процессора.

Проанализируем пример, изображенный на рис.5.4.2. В операторе IF с помощью функции ID определяются активные виртуальные ПЭ (все, кроме второго ПЭ). Таким образом определяются неактивные виртуальные ПЭ для конфигурации источника (X) и приемника данных (Y). В случае, когда i-й элемент ПЭ-источника не определен (например, 0-й,который не существует или 2-й – неактивный), то перезапись производится не с i–го в (i+1)-й, а из (i+1)–го в (i+1)–й. 2-й элемент ПЭ-приемника является неизменным.

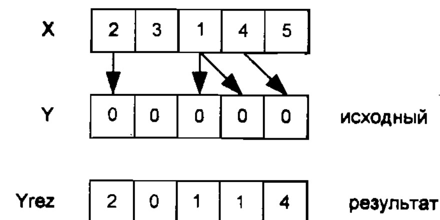


Рис.5.4.2. Применение функции MOVE

CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;
IF ID(linear)<>2 THEN Y := RECEIVE.right(X); --ID – идентификатор процессора.

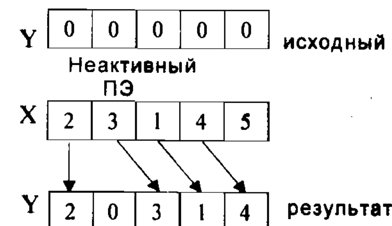


Рис.5.4.3. Применение функции RECEIVE

В примере, изображенном на рис.5.4.3., в операторе IF с помощью функции ID определяются активные виртуальные ПЭ для конфигурации приемника данных (Y). Поскольку 2-й элемент является неактивным, то его значение остается неизменным, а остальные значения элементов изменяются , как указано на рис.5.4.3.

CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;

IF ID(linear)<>2 THEN SEND.right(X,Y); --ID – идентификатор процессора.

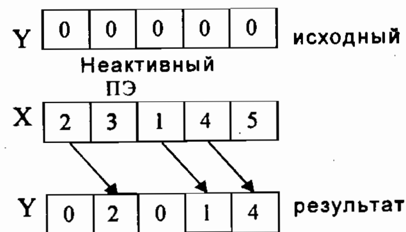


Рис.5.4.4. Применение процедуры SEND

В примере, изображенном на рис.5.4.4., в операторе IF с помощью функции ID определяются активные виртуальные ПЭ для конфигурации источника данных (X). Поскольку 2-й элемент является неактивным, то значение 3-го элемента массива Y остается неизменным, неизменным остается также значение 1-го элемента Y,а остальные значения элементов изменяются , как указано на рис.5.4.4.

Универсальные связи

Для организации универсальных связей могут быть использованы два вида операций:

- SEND.<<index>>
- RECEIVE.<<index>>

где - index – это массив, в котором записываются адреса виртуальных ПЭ-источников либо ПЭ-приемников данных.

Примеры универсальных связей:

```
CONFIGURATION linear[1..5];
CONNECTION right linear[i]→linear[i+1];
VAR X, Y, IA: linear if INTEGER;
Y := RECEIVE.<<IA>>(X); --Y получает из X то, что лежит по адресу IA.
```

```
CONFIGURATION linear[1..5];
CONNECTION right linear[i]→linear[i+1];
VAR X, Y, IA: linear if INTEGER;
SEND.<<IA>>(X,Y); -- послать из X в Y и разместить по адресу IA.
```

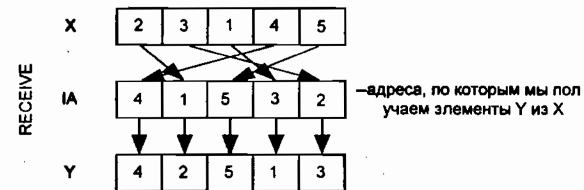



Рис.5.4.5. Пример применения операции RECEIVE

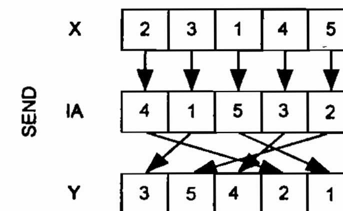


Рис.5.4.6. Пример применения операции SEND

6. Языковые средства описания асинхронных параллельных процессов

Существует множество прикладных задач, которые реализуются с помощью алгоритмов, допускающих параллельную обработку, но плохо векторизуемых. К таким задачам относятся некоторые задачи моделирования сложных систем, таких как модели нефтяных месторождений и другие. Эти задачи реализуются в системах класса МКМД, которые, как известно, относятся к асинхронному типу. В асинхронной КС, как уже упоминалось, не существует общего синхронизирующего сигнала (машинного такта системы). Поэтому в этих системах необходимо выполнять синхронизацию программными средствами. Рассмотрим основные причины, вызывающие необходимость синхронизации:

- обмен информацией между ветвями (параллельными процессами);
- доступ к общим ресурсам;
- завершение выполнения ветвей (если при выполнении параллельной программы допускается порождение и уничтожение параллельных ветвей, то при завершении выполнения ветви необходима синхронизация).

6.1. Описание параллельных процессов

Каждый процесс, который может быть выполнен параллельно с другим (другими) процессами, может быть описан как задача, процедура, подпрограмма, функция, блок или даже оператор (в ОККАМ), которые имеют специальный признак, указывающий на то, что данная часть программы может выполняться одновременно с некоторыми другими частями этой программы. Таким признаком может быть служебное слово:

- PARBEGIN;
- COBEGIN;
- PSUBROUTINE;
- PAR;
- PARPROCEDURE.

Так, например, в языке ОККАМ применяется служебное слово PAR, которое означает, что последующие за ним перечисленные процессы будут выполняться одновременно.

6.2. Программные средства инициализации и завершения параллельных процессов

Как уже упоминалось, в МКМД системах обрабатывается поток асинхронных процессов. В связи с этим, необходимы языковые средства

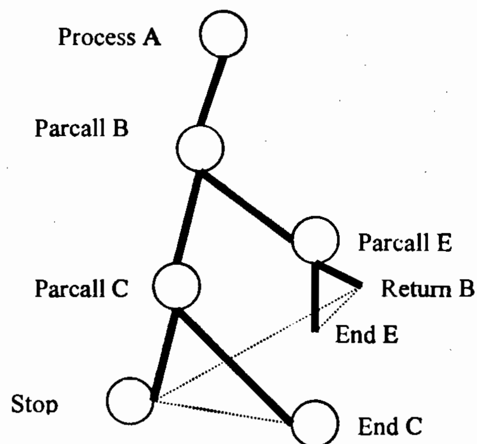


Рис. 6.2.1. Применение конструкции PAR CALL

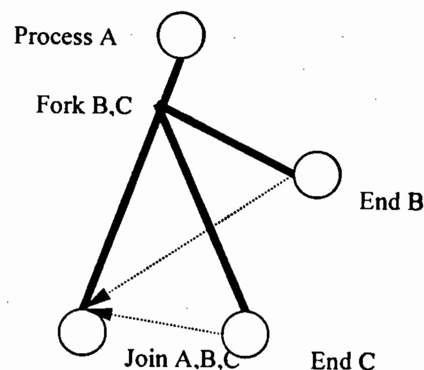


Рис.6.2.2. Применение конструкции FORK.

инициализации процессов, а также их завершения (для того, чтобы знать, когда инициировать последующие процессы). Эти средства близки (синтаксически) к языковым средствам обращения к подпрограммам или процедурам в данном языке. В языках параллельного программирования используется три типа конструкций для инициализации и завершения параллельных процессов:

1. Порождение одиночного процесса, вследствие чего вызывающий и вызываемый процессы выполняются одновременно (то есть параллельный процесс рассматривается как подпрограмма, но выполняющаяся параллельно с вызывающей программой). В этом случае для инициализации процесса используется оператор, включающий служебное слово PAR CALL, после которого следует имя процесса и параметры (в случае необходимости). Для завершения процессов могут использоваться операторы RETURN <имя процесса> либо END <имя процесса>. Первый из них применяется при завершении процессов, которые в свою очередь порождали еще какие-либо процессы (являются одновременно вызываемыми и вызывающими), а второй при завершении процессов, которые не порождали никаких процессов, т.е. являются только вызываемыми. Если процесс был последним в задаче, то при его завершении используется оператор STOP. Пример применения данной конструкции изображен на рис.6.2.1.
2. Одновременная инициализация множества процессов одним вызывающим процессам. В этом случае при инициализации процессов используется оператор FORK<имена вызывающих процессов>, которые будут выполняться до тех пор, пока не завершится какой-либо из них (с

помощью оператора END<имя процесса>) либо эти процессы не объединятся (с помощью оператора JOIN<имена объединяемых процессов>). Пример применения данной конструкции изображен на рис.6.2.2

3. Одновременное порождение и завершение параллельных процессов. Это белее жесткая конструкция по сравнению с предыдущим вариантом. Для реализации данной конструкции используются операторы COBEGIN <список процессов> – для инициализации и COEND<список процессов> – для завершения процессов. Пример применения данной конструкции изображен на рис.6.2.3.

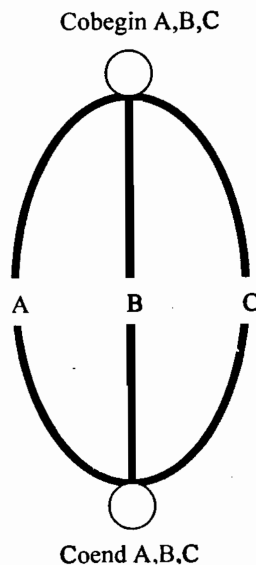


Рис.6.2.3. Применение конструкции COBEGIN

Следует отметить, что первый и второй типы конструкций предусматривают асинхронную инициализацию и завершение параллельных процессов, в то время как третий тип предусматривает синхронную инициализация и завершение процессов.

Операторы RETURN, JOIN, COEND являются фактически синхронизирующими примитивами типа "семафор". Процесс,

порождающий асинхронные параллельные процессы, проверяют с помощью семафоров, закончилось ли выполнение порожденных им процессов, и если нет, то ожидает их завершения, а если они закончились, то порождающий процесс также может завершиться.

Кроме того, в параллельном Pascal предусмотрены так называемые управляющие выражения. Они дают возможность непосредственно описывать порядок выполнения асинхронных параллельных процессов. Синтаксис выражения имеет вид:

PATH <управляющее выражение>END

Управляющее выражение содержит имена процессов и определяет порядок их выполнения. Список имен, разделяемых ";" означает последовательное выполнение процессов. Список имен, разделяемых "," означает параллельное выполнение перечисленных процессов. "n[управляющее выражение]" означает возможность инициирования указанных процессов не более, чем n раз до их завершения. "[управляющее выражение]" означает неограниченное число раз инициирования перечисленных процессов до их завершения.

Примеры:

PATH A, B, C, D – эти процессы выполняются последовательно.

PATH A; B; C; D -- эти процессы выполняются параллельно.

PATH n[A;B;C;D] – эта группа процессов будет инициализирована n раз (циклически).

Инициализация процессов сопровождается постановкой их в очередь на исполнение. Если в системе на момент инициализации процессов имеются свободные ресурсы, то диспетчер ОС назначает процессы на процессоры, т.е. инициализация является физической. Если же свободных процессоров нет, то процессы ожидают освобождения ресурсов в очереди. Выбор свободного процессора для процесса средствами ОС, часто не приводит к оптимальному решению всей задачи. Поэтому в некоторых языках параллельного программирования (например, СИ и OCCAM) предусмотрены средства назначения процессов на ресурсы пользователями, которые могут привести к получению более высокой реальной пользовательской производительности системы. Такие процедуры в программах называются **файлами конфигурации.**

Рассмотрим пример файла конфигурации на языке OCCAM:

PAR PLACED - - говорит о том, что будет указано место выполнения параллельных процессов.

PROCESSOR 0 T4 –тип процессора;

[последовательность процессов, выполняемых на процессоре 0]

PROCESSOR 1 T4

[последовательность процессов, выполняемых на процессоре 1].

Подведем итог: Первые три конструкции PARCALL, FORK, COBEGIN создают структуры древовидные (эти операторы ставят готовые к выполнению процессы в очередь, из которой они выбираются по мере освобождения процессоров).

Управляющие выражения в конструкциях [управляющее выражение] генерируют циклическое выполнение процессов. Но каждый вновь инициируемый процесс должен иметь свое имя. Это имя ему дает ОС, которое программисту неизвестно, и если произошла ошибка, ее трудно найти. Это пример того, что слишком большие языковые возможности в управлении процессами имеют свои недостатки.

6.3. Языковые средства синхронизации доступа к общим ресурсам

Последовательность операторов процесса, которые претендуют на использование некоторого разделяемого (общего) ресурса, называют критическим интервалом (областью, зоной, участком). Программист, составляющий программу, должен следить за появлением критических интервалов и обеспечивать такие условия , чтобы к критическому интервалу имел доступ только один процесс, а во время его действия остальным процессам не разрешался бы доступ к общим переменным. Такая синхронизация при обращении к общим ресурсам сводится к решению так называемой *задачи взаимного исключения*. Взаимное исключение обеспечивается окаймлением критического интервала(пролог и эпилог).

При работе с критическим интервалом устанавливаются следующие правила:

1. Если критический участок свободен, процесс может войти в него без каких- либо задержек.
2. Когда процесс находится внутри критического участка, другие процессы, которые пытаются войти в тот же критический участок , задерживаются на его входе.
3. Если процесс покидает участок и есть процессы, претендующие на него, один из них получает доступ на вход.
4. Дисциплина выбора процесса из очереди, ожидающих входа, зависит от требований, предъявляемых к системе, или от заданного приоритета.

В качестве общих ресурсов чаще всего используется общая память (для ПКС с общей памятью) или ее банки (в случае ПКС с разделяемой памятью). Но вообще общие ресурсы могут быть любыми.

Существуют следующие механизмы, которые применяются при решении задачи взаимного исключения:

- флажки;
- семафоры;
- условные критические участки;

- мониторы.

Рассмотрим суть первого механизма. В каждом процессе, который претендует на общий ресурс вводится переменная, называемая *флажком*. Процессам разрешается считывание всех флажков, но изменять значение лишь своего флажка. Основу данного механизма составляет правило: процесс получает доступ к общему ресурсу (входит в критический участок) тогда, когда он “выставил” свой флажок, в то время как флажки всех остальных процессов не “выставлены”. Соответственно после завершения работы в критическом участке данный процесс “снимает” свой флажок и освобождает дорогу для других процессов. Такой механизм работает хорошо до тех пор, пока несколько процессов одновременно не подходят к критическому участку. Тогда они одновременно “выставляют” свои флажки. Каждый проверяет флажки соседей и обнаруживает, что критический интервал занят. Возникает режим deadlock. Это главный недостаток подобного механизма. Борьба с этим недостатком возможна за счет введения случайного запаздывания между выставлением и снятием флажка в различных процессах. В общем случае такой механизм требует значительных расходов ресурсов, как с точки зрения объема оперативной памяти, так и малоэффективного использования времени процессора, занятого опросом состояния флажков других процессов. Преимуществом этого механизма является его простота.

Развитие механизма флажков связано с введением механизма семафоров, введенных Дейкстрой.. Семафор – это неотрицательная переменная, для которой определены две операции P (пропустить) и V (освободить). Обычно семафор S принимает два значения 0 и 1 и называется двоичным. Операция P на семафоре S выполняется следующим образом. Проверяется значение S. Если S>0, т.е. S=1, то P(S) – истинно и это означает пропуск процесса в критический участок и S=S-1. Таким образом, если в этот момент будет предпринята попытка войти в этот критический участок, то P(S) для других процессов будет ложно и вход в критический участок будет задержан. В конце критического участка выполняется операция V, которая изменяет значение семафора S=S+1 и тем самым открывает возможность входа в критическую область другого процесса.

Недостатками семафоров является плохая структуризация и выразительность. В этом механизме не предусмотрены никакие специальные конструкции выделения общих (разделяемых) данных. Поэтому программист может включить использование общих данных как в критическом интервале, так и вне их, что приводит к ошибкам. Кроме этого возникают ошибки, связанные с пропуском одной из операций (P или V) или включение P в один семафор, а V в другой – это приводит к нарушению принципа взаимного исключения. Ведение семафоров

ухудшает прозрачность текста программы, затрудняет отладку и возможную последующую модификацию.

Определенным улучшением семафоров можно считать предложенные В. Хансеном средства выделения критических интервалов в программе. Программисту необходимо только отметить эти зоны, а компилятор автоматически вводит необходимые семафоры и требуемые операции, регулирующие взаимное исключение. Снимая с программиста определенную нагрузку, эта техника все же оставляет за ним процедуру выделения критических интервалов по тексту. Таким образом, при работе с семафорами высока вероятность ошибки, которую трудно найти и исправить.

Для повышения надежности механизмов взаимного исключения были предложены два типа конструкций: условные критические участки и мониторы.

При использовании подхода условных критических участков общие ресурсы (данные) представлены переменными, которые имеют специальный оператор описания SHARED. В контексте языка Pascal данный оператор имеет вид:

VAR A: SHARED T
где A – идентификатор переменной;
T – тип переменной.

Процесс может использовать общие переменные только внутри оператора, называемым условным критическим интервалом., который имеет такой вид:

REGION A WHEN B DO S
где B – условное выражение;
S – оператор (-ры) критического участка .

Таким образом процесс может войти в критический участок только, если B – истинно, иначе этот процесс откладывается и задерживается до тех пор, пока другой процесс не завершит своих вычислений в критическом участке. Задержанные процессы образуют очередь, повторные вхождения в условные критические участки реализуются оператором REGION. Условные критические участки «разбросаны» по всей программе. Поэтому проследить за ходом использования разделяемых ресурсов и убедиться в правильности их использования достаточно сложно. Так же трудно правильно определить значение B.

Эта трудность преодолевается языковой конструкцией централизующей управление асинхронными процессами, которая

получила название монитора. Идея монитора заключается в создании механизма, который бы соответствующим образом унифицировал взаимодействие параллельных процессов по синхронизации, разделяемым данным и программам, которые обрабатывают эти данные. Монитор защищает данные и доступ к ним может быть осуществлен только с помощью программы, включенной в тело монитора. В мониторе имеется два типа процедур: входные и выходные. Обращение к монитору происходит через входные процедуры. Обращение к внутренним процедурам возможно только из тела монитора. Монитор гарантирует, что в каждый момент времени процедуры монитора может использовать только один процесс. Этот процесс называется процессом в мониторе. Остальные процессы, обращающиеся в этот момент к монитору, задерживаются и ставятся в очередь. Монитор «отторгает» от процессов все их критические интервалы, связанные с данными ресурсами, превращая их в свои процедуры.

Обращение к монитору (вызов монитора) производится с помощью указания имени процедуры и имени монитора. Такое обращение доступно всем процессам в том смысле, что любой из процессов может попытаться вызвать любую программу, но только один из них может войти в процедуру, остальные же должны ждать, когда будет завершен предыдущий вызов. Процедуры монитора могут содержать только переменные, локализованные в теле монитора. Это ограничение предостерегает от ошибок возникающих при использовании семафоров.

Синтаксически описание монитора начинается со слова MONITOR. Затем следует описание процедуры и данных. Функционально монитор объединяет некоторые данные, в том числе типа «событие», используемые параллельными процессами, и все операторы обрабатывающие эти данные, в единый блок.

Переменная типа «событие» используется для управления доступом к разделяемым ресурсам. В мониторах она называется «условная переменная». К переменной типа «событие» создается очередь процессов, ожидающих выполнения определенного события. Над этой переменной разрешены операции двух типов: WAIT и SIGNAL. Оператор WAIT задерживает выполнение процесса, вызвавшего монитор, и открывает доступ процесса к монитору. Выполнение задержанного процесса может быть инициировано оператором SIGNAL другого процесса. Оператор SIGNAL выполняется следующим образом. Если очередь к переменной «событие» не пуста, то из очереди выбирается один из процессов и инициируется его выполнение, а если очередь пуста, то не производится никакого действия.

Пример монитора:

```
RESOURSE: MONITOR
    BEGIN LOGICAL: BUSY;
        CONDITIONAL X;        -переменная, типа «событие»
        PROC ACQUIRE;                   -захват ресурса
    BEGIN IF BUSY THEN WAIT X;
        BUSY:=TRUE;
    END;
    PROC RELEASE;                       -освобождение ресурса
    BRGIN BUSY:=FALSE;
        SIGNAL X;
    END;
    BUSY:=FALSE;
    END;
```

В мониторе может быть введено логическое условие в виде условного оператора WAIT (L), где L – булевское выражение. Выражение WAIT (L) задерживает инициализацию процесса до тех пор, пока значение L не станет истинным.

Третий подход синхронизации доступа к разделяемым ресурсам может быть реализован в управляющем выражении тапа PATH управляющим выражением END.

Так, запись PATH P1 END означает, что если несколько процессов стоят в очереди к процедуре P1, то один из них получит к ней доступ. После завершения выполнения P1 по первому обращению к ней P1 получает доступ второй процесс и т.д. Возможны комбинации управляющих выражений этого типа с другими. Запись{P1} означает что P1 может быть выполнена несколькими процессами одновременно. Если один процесс начал выполнение P1, то другой может получить к ней доступ без задержки. Этот подход достаточно специализирован и используется в языках, где реализованы конструкции PATH.

6.4. Языковые средства синхронизации сообщений для МКМД–систем с распределенной памятью

В системах класса МКМД с распределенной памятью (MPP) для организации обмена сообщениями требуются новые языковые средства и серьезная системная поддержка.

Особенность обмена информацией в данном случае заключается в том, что тут действуют две автономные стороны: *отправитель (отправители)* и *адресат (адресаты)*. Причем моменты времени, когда процессы готовы к отправлению и приему могут не совпадать. Поэтому возникает необходимость синхронизации.

Существует три основных способа синхронизации пересылки сообщений:

1) *Слабосвязанные системы (loosly coupled).*

Обмен сообщениями производится через «почтовый ящик» в виде буферного накопителя.

2) *Сильно-связанные системы (tightly coupled).*

Обмен сообщениями производится в отдельных точках синхронизации («рандеву»).

3) *Полностью связанные системы (completely couple).*

Существует единый централизованный синхронизирующий источник (send, receive). Централизованная синхронизация необходима, когда происходит общение трех и более процессов.

В слабосвязанных системах в обмене участвуют два процесса и используется асинхронная модель пересылки данных. Смысл такой модели заключается в следующем. Когда готовы данные в процессе-источнике, они передаются сразу же либо в процесс-приемник, если он готов к приему данных, либо в промежуточную буферную память («почтовый ящик»), после чего процесс-источник продолжает свою работу независимо от того, переданы ли данные процессу-приемнику. Процесс-приемник принимает данные в момент готовности и о полученных данных не информирует процесс-источник. Примером использования является язык BBC.

Преимущество: высокая эффективность использования процессорных ресурсов.

Недостатки:

1. Наличие буферной памяти для временного хранения сообщений, объем которой может достигать больших размеров с ростом числа процессоров КС.
2. Низкая надежность доставки сообщений, т.к. процесс-приемник не информирует процесс-источник о своем состоянии. Возможна ситуация, когда процесс-источник заканчивает свое существование до приема его сообщения процессом-приемником, и, вообще, может никогда не узнать о том, принято ли его сообщение приемником.

В сильно связанных системах так же, как и в слабосвязанных, используется два процесса, которые участвуют в обмене, но буферная память отсутствует. Для синхронизации используется механизм «рандеву». Данный подход обеспечивает синхронную модель обмена. Примеры использования - языки ADA, OCCAM.Смысл механизма «рандеву» состоит в следующем. При взаимодействии двух процессов, процесс-источник приостанавливается до тех пор, пока процесс-приемник не будет готов к приему. Когда процесс-приемник готов к приему, он посылает процессу-источнику сигнал готовности, после чего происходит пересылка данных. Далее процесс-приемник информирует источник о приеме его

сообщения (служебное слово – reply message), после чего процесс-источник и процесс-приемник продолжают свою работу. Осуществляется постоянная проверка надежности доставки. Ожидания ведут к снижению эффективности работы процессоров. Однако для мультипрограммного режима это лишь относительное снижение производительности.

Преимущества:

1) Высокая надежность доставки сообщений.
2) Отсутствие необходимости дополнительных ресурсов (буферной памяти).

Недостаток: возможна низкая эффективность использования отдельных процессорных ресурсов.

Для преодоления указанных недостатков, имеет смысл объединить первый и второй подходы. В качестве буферной памяти можно организовать дополнительный промежуточный процесс, который будет принимать и передавать данные, и он всегда будет готов к таким действиям (рис.6.4.1). Таким образом, программист вводит буферную память, но использует механизм «рандеву».

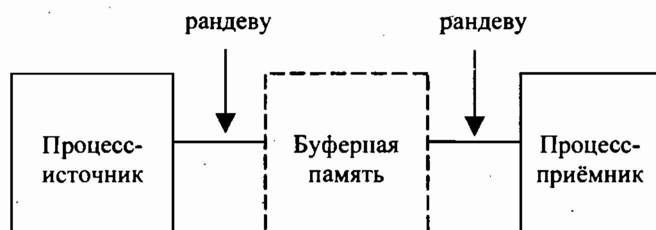


Рис.6.4.1. Комбинированный способ пересылки сообщений

В полностью связанных системах одновременно обмениваются данными множество процессов. Один процесс должен опросить все процессы, готовы ли они к обмену (они должны быть готовы одновременно). Механизм основан на механизме «рандеву». Если процессы находятся на различных процессорах, то нужен механизм маршрутизации.

Сильно связанные системы являются наиболее применимыми в настоящее время. Рассмотрим специальные языковые конструкции, необходимые для отправления и приема информации в таких системах. Существует два основных оператора пересылки и приема, которые содержат соответствующие служебные слова языка, адреса отправителя и

адресата, и список переменных, значения которых пересылаются из одного процесса в другой:

SEND <набор значений> TO <адрес процесса-приемника> *(этот оператор находится в процессе-источнике)*

RECEIVE <набор значений> FROM <адрес процесса-источника> *(этот оператор находится в процессе-приемнике)*

Рассмотрим операторы пересылки сообщений, используемые в языке OCCAM. В данном случае, вместо ключевых слов SEND и RECEIVE используются символы “!” и “?”, а для определения адресов процесса-приемника и процесса-источника используется имя логического канала. Для организации каждого обмена в языке OCCAM используется логический канал с уникальным именем, которое применяется только одним процессом-отправителем и одним процессом-приемником.

Оператор SEND имеет следующий вид:

<имя логического канала> ! <набор значений>

Оператор RECEIVE имеет такой вид:

<имя логического канала> ? <переменная>

Пример: Пусть нам необходимо передать переменной a, которая находится в одном процессе значение b+5, находящееся в другом процессе. Тогда операторы пересылки данных имеют следующий вид:

```
chan1  !  b+5.
chan1  ?  a.
```

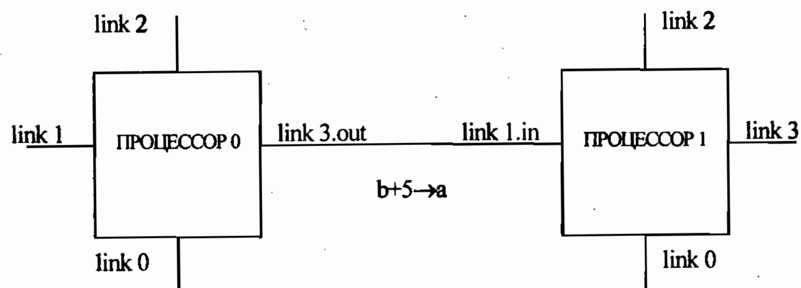
В приведенном примере для обмена данными используется имя логического канала chan1.

Чтобы пересылка данных была реализована физически, следует, прежде всего, определить на каких процессорах выполняются процессы 1 и 2. Возможны два варианта: оба процесса выполняются на одном и том же процессоре либо процессы выполняются на различных процессорах. При реализации первого варианта обмен происходит через оперативную память процессора и для этого достаточно перечисленных выше двух операторов. При реализации же второго варианта обмен осуществляется по физическим каналам, соединяющих процессоры. Поэтому, кроме перечисленных операторов, в файл конфигурации необходимо включить операторы соответствия логических каналов физическим.

Пример:

```
PLACED     F  PAR
PROCESSOR  0  T4
```

```
PLACE chan1    AT    link3.out
PROCESSOR      0     T4
PLACE chan1    AT    link1.in
```



7. Заключение

Все рассмотренные нами языковые конструкции сводятся к описанию параллельных процессов и матричных вычислений. Параллельные процессы являются реализацией параллелизма независимых ветвей, а матричные вычисления являются одним из частных случаев реализации естественного параллелизма. Другим примером естественного параллелизма в программах является параллельное выполнение циклических участков, однако, на уровне создания параллельных программ чаще всего специальных средств параллельного выполнения циклических участков нет.

Примером использования параллельно задаваемых циклов является VS FORTRAN (расширение FORTRAN77) для IBM3096 (двухпроцессорная КС с векторной обработкой, общей ОП, каждый имеет СОЗУ). Параллельный цикл задается следующим образом:

```
@DO
.
.
.
@END DO;
```

Требованием является независимость всех итераций и следующие условия:

- Индексы циклов должны быть целочисленными;

- Не должно быть переходов из тела цикла и внутрь тела;
- Начальное значение индекса должно быть меньше конечного значения индекса.

Как правило, не используется никаких языковых средств описания параллелизма смежных операций. Таким образом, распараллеливание циклов и программ для КС, реализующих параллелизм смежных операций, выполняется на этапе компиляции.

III. Особенности компиляции программ для КС

8. Традиционные этапы компиляции

Компиляция – это преобразование программы на входном языке в эквивалентную последовательность машинных команд.

Рассмотрим основные этапы процесса компиляции. Процесс компиляции делят на четыре логически обособленных этапа:

1. Лексический анализатор.
2. Синтаксический анализатор.
3. Семантический анализатор
4. Генерация кода.

Логически транслятор содержит две части – синтез и анализ. В первой части распознаются конструкции абстрактной программы и проверяют их правильность, а во второй части производится построение эквивалентной выходной программы в машинных кодах. Реализация этих этапов в виде последовательности шагов называется просмотрами.

Лексический анализ производит построение по исходному тексту программы лексической свертки. Лексический анализатор на входе имеет длинную цепочку символов (исходная программа) и должен преобразовать ее в последовательность внутренних кодировок и набор таблиц лексем, образующих лексическую свертку. Лексемы - это цепочки литер, образующих конструкции языка (идентификаторы, знаки операций, служебные слова, разделители и т. п.). Таким образом, лексическая свертка состоит из набора лексем в едином формате, которые представляются посредством дескриптора, содержащего два типа информации: тип лексемы (идентификатор, служебное слово) и местонахождение данной лексемы (вход в таблицу идентификаторов, номер в таблице служебных слов и т. п.).

Следующим этапом является синтаксический анализ, осуществляющий выявление в лексической свертке понятий (цепочки лексем – арифметические выражения, операторы присваивания, условный оператор и др.) и их структуру, а также проверку удовлетворяет ли эта структура синтаксису языка.

Третьим этапом является семантический анализ, проверяется смысловая правильность программы. Другими словами, семантический анализ – это нахождение значений атрибутов (набор характеристик) для понятий программы (лексем и типов лексем),в результате чего строится атрибутное дерево программы.

Последним этапом является получение программы в машинных кодах по входной программе (которая представлена как атрибутное дерево).

9. Особенности компиляции при использовании различных способов программирования для параллельных КС

Как отмечалось ранее, существуют два пути программирования для параллельных КС:

- Разработка новых параллельных программ.
- Адаптация программ, написанных для однопроцессорных компьютерах.

При первом подходе при трансляции стоит задача адекватного перенесения параллелизма с языка высокого уровня на машинный язык. Именно адекватного, так как «подгонка» параллелизма к архитектуре КС производится на этапе программирования. Стадии лексического и синтаксического анализов при трансляции как для матричных и векторных, так и для МКМД систем могут быть организованы традиционным способом. Необходимый "довесок" для обработки параллельных операторов состоит из процедур распознавания некоторых параллельных конструкций, что ведет к появлению новых типов семантических подпрограмм. Как было отмечено выше, практически во всех параллельных языках программирования отсутствуют конструкции для распараллеливания циклических участков программ. Поэтому дополнительной функцией компиляторов практически всех языков параллельного программирования является векторизация (для матричных и векторных КС) или распараллеливание (для МКМД систем) циклов.

Второй путь предусматривает распараллеливание не в процессе программирования, а в процессе компиляции. Таким образом, в этом случае трансляторы для любых типов КС, кроме традиционных этапов, должны выполнять также функцию автоматического распараллеливания программ. Второй путь программирования теоретически может быть использован для любых типов структур КС. Практически этот способ программирования используется для *Dataflow*, *VLIW* и ***конвейерных*** КС. Для перечисленных систем компилятор должен обеспечивать автоматическое разбиение программы на независимые операции (параллелизм смежных операций).

10. Способы распараллеливания процесса компиляции

Для ускорения самого процесса компиляции, для параллельных КС при любом способе программирования могут применяться алгоритмы параллельной трансляции.

Существует три способа распараллеливания процесса компиляции:

1. *Конвейеризация* процесса трансляции с использованием традиционных последовательных алгоритмов. В компиляторе выделяются независимые процессы, связанные очередями частичных результатов, передаваемых от одного процесса к другому (лексический, синтаксический, семантический анализаторы и генерация кода). Каждый из этих процессов может быть выполнен на отдельном процессоре (слое конвейера). Данный подход целесообразно использовать в многопроцессорных системах с разделяемой (общей) памятью, тогда как в МКМД системах с распределенной памятью возникают большие потери на пересылку частичных результатов.

2. ***Параллельное выполнение отдельных этапов трансляции*** (разработка новых параллельных алгоритмов трансляции). Для систем с векторными процессорами эти алгоритмы являются достаточно перспективными. Существуют алгоритмы лексического анализа, использующие векторные операции. Кроме этого векторные операции могут использоваться на этапе синтаксического анализа. В этом случае осуществляется параллельное распознавание всех заданных синтаксических конструкций, например на первом этапе – все разделители, на втором – все идентификаторы и т. п. Семантический анализ и генерацию кода распараллелить сложно. Для реализации данного подхода может понадобиться множество копий программ.

3. *Параллельная трансляция частей программы.* Такой подход сводится к одновременной компиляции различных операторов или групп операторов с последующей «сшивкой» оттранслированных частей в общую программу. Данный способ имеет смысл особенно для программ с модульной структурой. Основная трудность его применения – обеспечение достаточно эффективной синхронизации при объединении параллельно откомпилированных модулей.

11. Автоматическое распараллеливание программ

Для выявления различных типов параллелизма необходимо рассмотреть типичные элементы распараллеливания в программах. Перечислим основные элементы и соответственно уровни распараллеливания:

- распараллеливание арифметических выражений (на уровне операций);
- распараллеливание линейных участков программ (на уровне оператора);
- распараллеливание разветвленных участков программ (на уровне операторов);
- распараллеливание и векторизация циклов (соответственно на уровне итераций и векторных операций);
- распараллеливание универсальных программ (на уровне ветвей).

Распараллеливание на каждом из перечисленных уровней имеет свои особенности и сложности. Наиболее формализуемыми элементами распараллеливания является арифметические выражения и линейные участки, т.к. в данном случае имеется полная определенность как с точки зрения размеров (число операторов и соответствующих им операций) элементов распараллеливания, так и с точки зрения уровней распараллеливания. Более сложно вопросы распараллеливания решаются для разветвленных участков программ и циклов, поскольку появляется некоторая неоднозначность. Так, число операторов и операций (размер), реально используемых в разветвленных участках зависит от исходных данных программы и поэтому неоднозначно, число итераций циклов также не всегда известно заранее (циклы с while). И, наконец, меньше всего поддается формализации распараллеливание программ по ветвям, поскольку само понятие ветвь – неоднозначно по размеру. Более того программа может быть разветвленной и тогда имеется неопределенность еще и с точки зрения элемента распараллеливания. Таким образом, качество автоматического распараллеливания зависит как от характеристик (структуры) последовательной программы (есть ли в ней циклы, ветвления), так и от типа структуры КС, для которой необходимо выполнить распараллеливание (а значит от типа параллелизма: параллелизм смежных операций, естественный или ветвей).

Задача автоматического распараллеливания для КС, ориентированных на параллелизм смежных операций, сводится к распараллеливанию программ до уровня операций.

Задача автоматического распараллеливания для КС, ориентированных на естественный параллелизм, сводится к распараллеливанию циклов.

И, наконец, задача автоматического распараллеливания для КС, ориентированных на параллелизм независимых ветвей, сводится к распараллеливанию циклов и к разбиению программы на независимые асинхронные процессы. Последняя функция должна присутствовать в компиляторе только в случае применения традиционных языков программирования, в случае же использования параллельных языков - данная функция отсутствует.

Результаты сказанного отражены в следующей таблице.

Тип КС	Функции транслятора, обеспечивающие автоматическое распараллеливание программ		
	Распараллеливание линейных участков и арифметических выражений	Векторизация и распараллеливание циклов	Разбиение программ на ветви (асинхронные процессы)
Потоковые, ССКС, Конвейерные	+	–	–
Векторные и матричные	–	+	–
МКМД	–	+	+ –

11.1 Распараллеливание линейных программ

Известно, что линейная программа – это программа без ветвлений. Основными понятиями при распараллеливании линейных программ являются информационный граф и граф параллельной формы. Информационный граф содержит информацию о выполняемых операторах (операциях) и о зависимости между операторами по данным, но не содержит информацию о порядке выполнения и зависимости по памяти. Всю эту информацию имеет граф параллельной формы.

Исходными данными для распараллеливания является запись программы в каком-либо виде, например в виде схемы алгоритма. Преобразование схемы алгоритма в граф параллельной формы и является задачей распараллеливания. Формально эту задачу решить несложно. Существует два необходимых и достаточных условия для параллельного выполнения операторов:

1. Информационная независимость (независимость по данным). Два оператора считаются независимыми по данным, если ни один из них не формирует данные, которые используются другим в качестве исходных.
2. Независимость по памяти. Два оператора **a** и **b** считаются независимыми по памяти, если для них выполняется условие:

$$Out(a) \cap In(b) = 0 \ \& \ In(a) \cap Out(b) = 0 \ \& \ Out(a) \cap Out(b) = 0$$

Где In – входное множество данных оператора.
Out – выходное множество данных оператора.

Рассмотрим пример. Пусть имеется схема алгоритма (рис.11.1.1). Для данной схемы алгоритма информационный граф и граф параллельной формы имеют вид, представленный на рис.11.1.2.

Необходимо стремится к уменьшению количества дуг (связей). Рассмотрим способы уменьшения числа связей в приведенных графах. Так, в информационном графе количество связей можно уменьшить только за счет применения другого алгоритма решения данной задачи. В графе параллельной формы может быть уменьшено число связей по памяти с помощью следующих двух подходов.

Первый способ связан с преобразованием линейной программы за счет увеличения объема оперативной памяти.

Для рассмотренного примера произведем следующее преобразование: заменим $a{:=}\varphi_3(b)$ на $d{:=}\varphi_3(b)$. Тогда схема алгоритма программы будет иметь вид, представленный на рис.11.1.3., а соответствующие ей информационный граф и граф параллельной формы на рис.11.1.5.

Второй способ связан с преобразованием линейной программы к *приведенной* форме, в которой уменьшение количества связей по памяти достигается без увеличение его объема. Для рассматриваемого примера преобразование к приведенной форме можно выполнить заменой $a{:=}\varphi_3(b)$ на $b{:=}\varphi_3(b)$. Схема алгоритма в приведенной форме имеет вид, представленный на рис.11.1.4. Соответствующие приведенной схеме алгоритма информационный граф и граф параллельной формы имеют такой же вид, как и в результате первого способа преобразования программы (рис.11.1.5.)

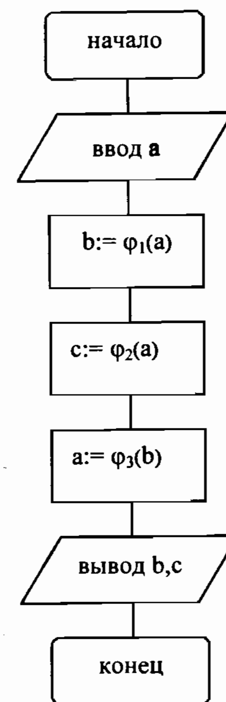


Рис.11.1.1. Пример схемы алгоритма

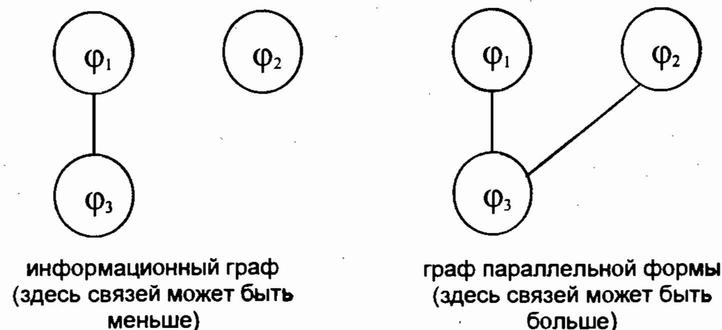


Рис.11.1.2. Информационный граф и граф параллельной формы

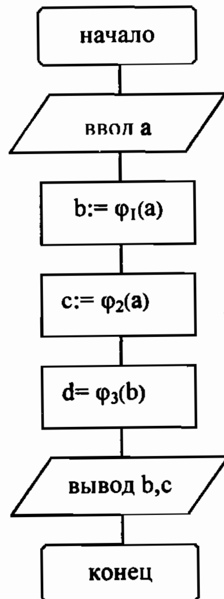


Рис.11.1.3. Схема алгоритма после первого способа преобразования

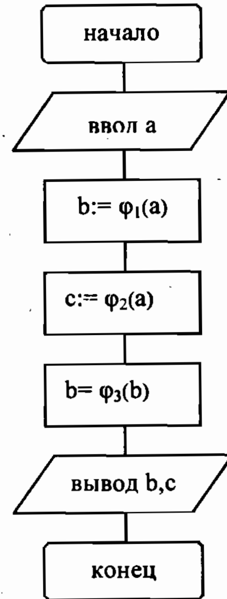
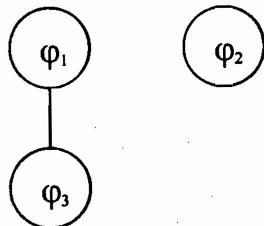
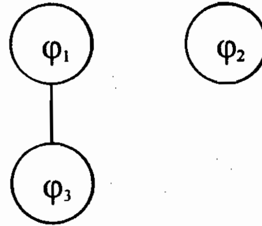


Рис.11.1.4. Схема алгоритма после второго способа преобразования



информационный граф

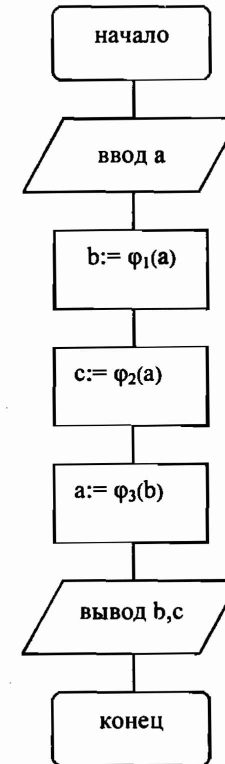


граф параллельной формы

Рис.11.1.5. Информационный граф и граф параллельной формы, соответствующие преобразованным схемам алгоритма

Рассмотрим пример реализации алгоритма распараллеливания линейной программы, основанного на построении и преобразовании матриц.

Пример: Пусть имеется линейная программа.



Out		In
B	=	$\varphi_1(a)$
C	=	$\varphi_2(a)$
A	=	$\varphi_3(b)$

Алгоритм распараллеливания состоит в следующем:

1. Начало.
2. Построение матрицы последовательности выполнения - С.
3. Построение матрицы смежности, в которой описаны информационные связи – А.
4. Построение матрицы входных и выходных данных – In, Out.
5. Построение матрицы графа параллельной формы – В на основании вышеперечисленных матриц .
6. Конец.

C

φ1	φ2	φ3
1		
1	1	

A

	φ1	φ2	φ3
φ1			
φ2			
φ3	1		

In

	a	b
φ1	1	
φ2	1	
φ3		1

Out

	a	b	c
φ1		1	
φ2			1
φ3	1		

Формируем результирующую матрицу B на основании исходных матриц:

$\mathbf{B_{ij} = (A_{ij} \cup IO_{ij}) \cap C_{ij}}$, где $\mathbf{IO_{ij}}$ – функция связи по памяти между i-ым и j-ым элементом.

$\mathbf{IO_{ij} = In_i \cap Out_j = 0 \ \& \ Out_i \cap In_j = 0 \ \& \ Out_i \cap Out_j = 0}$ (условие независимости по памяти)

B

	φ1	φ2	φ3
φ1			
φ2	0		
φ3	1	1	

Данная матрица B соответствует графу параллельной формы (рис.11.1.2).

11.2 Распараллеливание программ с ветвлениями

Существует два подхода распараллеливания разветвленных участков программ:

1. Преобразование разветвленного участка программы к линейному виду с последующим распараллеливанием его, как линейного.
2. Распараллеливание разветвленного участка программы по операторам.

Первый подход связан с укрупнением схемы алгоритма с целью ее приведения к линейной программе. В этом случае может быть использован механизм «гамаков».

Гамаком называется такой подграф **g** графа **G**, для которого обязательно существуют две вершины: входная A и выходная B, обладающие следующими свойствами:

1. Все дуги из A ведут в гамак.
2. Любой путь в B гамак ведет через A.
3. Все дуги в B ведут из гамака.
4. Любой путь, идущий из гамака вовне, проходит через B.

Гамак интересен тем, что его можно стянуть в одну вершину, не нарушая отношений смежности между другими вершинами графа, однако – это полный отказ от параллельной обработки.

Пример:

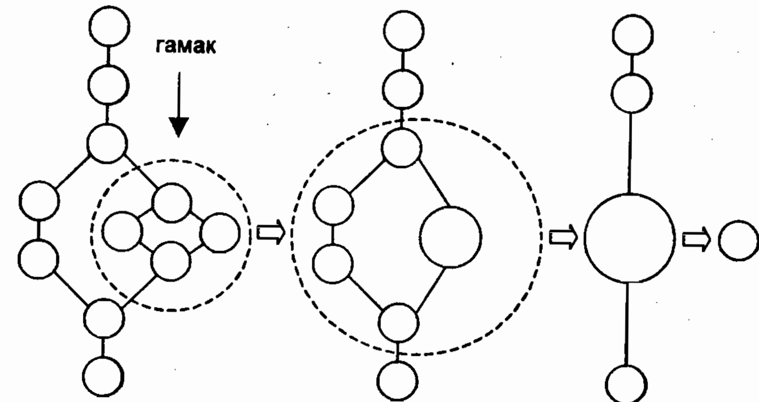


Рис.11.2.1. Последовательность применения механизма гамаков для разветвленной программы

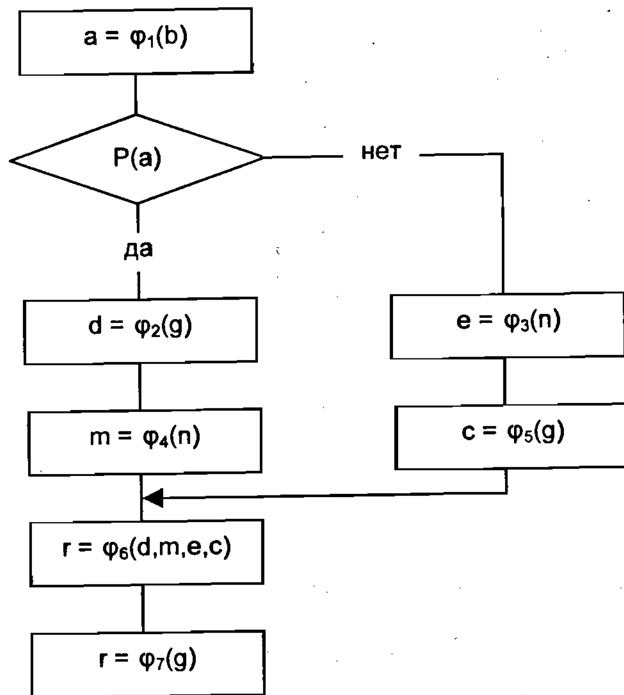
В результате применения механизма гамаков (рис.11.2.1) получаем линейную программу с операторами различной сложности. Далее эту программу можно распараллелить способом указанным выше. Преимущество такого подхода заключается в однозначности получаемого результата. Недостатками данного подхода являются сложность определения оптимальных границ гамаков, количества итераций применения механизма гамаков, а также различная сложность получаемых операторов (элементов параллельной обработки), которая может привести к низкой эффективности использования процессорных ресурсов.

Второй подход связан с распараллеливанием разветвленной программы по операторам без какого-либо предварительного ее преобразования. В отличие от линейных программ, для параллельного выполнения операторов разветвленной программы, необходимо и достаточно соблюдение трех следующих условий:

1. Независимость по данным.
2. Независимость по памяти.
3. Независимость по управлению, т.е. операторы не должны быть связаны между собой оператором проверки условия.

Рассмотрим алгоритм реализации данного подхода распараллеливания разветвленной программы, основанного на построении и преобразовании матриц.

Пусть имеется разветвленный участок алгоритма.



Алгоритм распараллеливания состоит в следующем:

1. Начало.
2. Построение матрицы последовательности выполнения - C.
3. Построение матрицы смежности, в которой описаны информационные связи – S.
4. Построение матрицы входных и выходных данных – I и O.
5. Построение матрицы логических связей – L.
6. Построение матрицы неполного параллелизма P' на основе матриц C, I, O, S.

$$P'_{ij} = (IO_{ij} \cup S_{ij}) \cap C_{ij}$$

$$IO_{ij} = In_i \cap Out_j = 0 \ \& \ Out_i \cap In_j = 0 \ \& \ Out_i \cap Out_j = 0$$

Данная матрица объединяет зависимости по данным и по памяти.

7. Построение матрицы графа параллельной формы – R на базе матриц L, P', C.
8. Конец.

	C							
	φ_1	P	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p	1							
φ_2	1	1						
φ_3	1	1						
φ_4	1	1	1					
φ_5	1	1		1				
φ_6	1	1	1	1	1	1		
φ_7	1	1	1	1	1	1	1	

	S							
	φ_1	p	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p	1							
φ_2								
φ_3								
φ_4								
φ_5								
φ_6			1	1	1	1		
φ_7								

	l							
	b	a	g	N	d	e	M	c
φ_1	1							
p		1						
φ_2			1					
φ_3				1				
φ_4				1				
φ_5			1					
φ_6					1	1	1	1
φ_7			1					

	O					
	A	d	e	m	c	r
φ_1	1					
p						
φ_2		1				
φ_3			1			
φ_4				1		
φ_5					1	
φ_6						1
φ_7						1

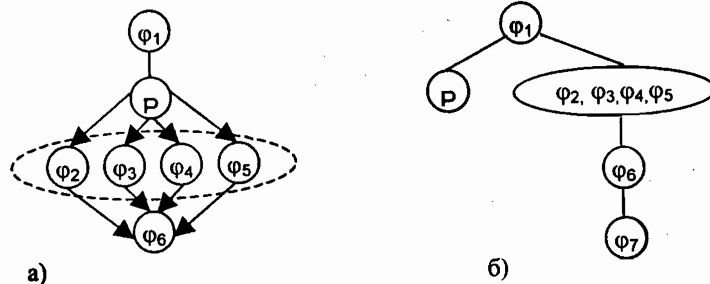
	I							
	φ_1	p	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p								
φ_2		1						
φ_3		1						
φ_4		1						
φ_5		1						
φ_6								
φ_7								

9 — 2-3006

P'	φ1	p	φ2	φ3	φ4	φ5	φ6	φ7
φ1	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0
φ2	0	1	0	0	0	0	0	0
φ3	0	1	0	0	0	0	0	0
φ4	0	1	0	0	0	0	0	0
φ5	0	1	0	0	0	0	0	0
φ6	0	0	1	1	1	1	0	0
φ7	0	0	0	0	0	0	1	0

R	φ1	p	φ2	φ3	φ4	φ5	φ6	φ7
φ1								
p	1							
φ2		1						
φ3		1						
φ4		1						
φ5		1						
φ6			1	1	1	1		
φ7							1	

Результирующей матрице R соответствует граф, изображенный на рис.11.2.2.а. Данный граф параллельной формы включает вершину P, которой соответствует операция проверки условия. Это означает, что мы получили некоторый объединенный граф параллельной формы, который включает множество реальный подграфов (в зависимости от исходных данных). Такой граф неоднозначен с точки зрения определения и оценки таких его характеристик, как число вершин, ширина ярусов, критический путь, общая трудоемкость выполнения, связность и др. Указанная неоднозначность является **недостатком** данного способа распараллеливания. С другой стороны, данный способ легко поддается формализации и в этом его **преимущество.**



а)

φ1
P
φ2, φ3,φ4,φ5
φ6
φ7

б)

Рис.11.2.2. Результирующие графы параллельной формы

Определенным усовершенствованием выполнения разветвленных участков программ является параллельное выполнение логически связанных операторов (рис.11.2.2.б). Это означает, что одновременно с проверкой условия организовывается выполнение всех, зависящих от него ветвей вычислений. Когда же результат проверки условия становится известным, выполнение ненужных ветвей приостанавливается. Таким образом, может быть достигнуто значительное повышение реальной производительности КС при выполнении разветвленных программ. Конечно же это возможно за счет дополнительной аппаратной и программной поддержки вычислительных средств. Такой подход получил распространение в настоящее время. Следует, однако, заметить, что он не вписывается в классические понятия параллелизма (см. необходимые и достаточные условия параллельного выполнения операторов в разветвленных программах).

11.3 Распараллеливание и векторизация циклических участков программ

Основная работа, которую выполняет любой компьютер – обработка циклов и других повторяющихся участков. Это объясняется тем, что ациклические участки вычисляются один раз, а фрагменты внутри циклов повторяются сотни и тысячи раз. Поэтому удельный вес таких фрагментов в вычислительных задачах очень высок и распараллеливание циклических участков имеет особое значение. В зависимости от типов КС применяется распараллеливание либо векторизация циклов.

Рассмотрим основные отличия распараллеливания и векторизации (рис.11.3.1). Распараллеливание применяется для асинхронных МКМД систем и предусматривает одновременное выполнение процессорами различных итераций (проходов) цикла. Векторизация применяется для синхронных ОКМД систем и предусматривает преобразование цикла в последовательность векторных команд, причем КС, для которой производится векторизация, должна иметь соответствующий набор векторных команд. Как распараллеливание, так и векторизация возможны при соблюдении некоторых условий и ограничений на структуру циклических участков. Прежде чем их перечислить, рассмотрим препятствия к распараллеливанию и векторизации циклов.

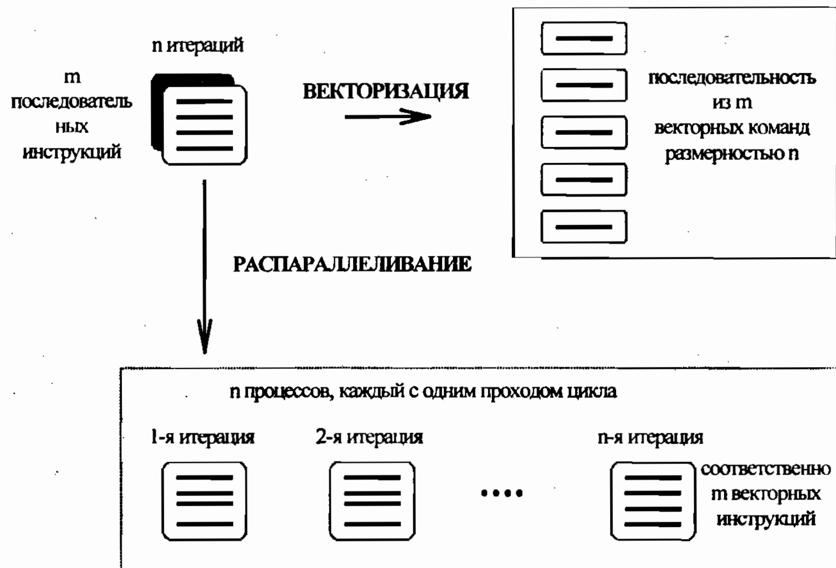


Рис.11.3.1. Распараллеливание и векторизация

11.3.1 Препятствия к распараллеливанию и векторизации циклов

Рассмотрим различные конструкции, которые затрудняют или даже блокируют процесс распараллеливания и векторизации. Другими словами, некоторые из ниже перечисленных конструкций «фундаментально» не распараллеливаемы и (или) не векторизуемы, а некоторые являются трудными для распараллеливания и (или) векторизации с точки зрения анализа и требуемого преобразования.

Наличие операторов условного перехода в теле цикла.

Процесс распараллеливания в данном случае может производится беспрепятственно, в то время как процесс векторизации может сдерживаться циклами, которые содержат операторы IF и передачу управления. Для того, чтобы такие циклы векторизовать, в компиляторах используются два метода:

1. Метод маскирования.
2. Метод укороченных векторов.

Пример: Рассмотрим цикл с условным оператором на языке FORTRAN

```
      DO 100 I = 1,N
      IF (A(I).NE.0) B(I) = C(I) + 10
100   CONTINUE
      END.
```

Если применить к этому циклу *метод маскирования*, то получится такая программа:

```
VECTOR CODE
LENGTH
    N      COMPARE A(1:N), 0  M(1:N)         - маска
    N      ADD C(1:N), 10     B(1:N)         WHERE M(1:N)   }
           (векторные команды длинной N)
```

Вначале, с помощью операции сравнения векторов, формируется разрядный вектор-маска. В следующей команде выполняется векторная операция под маской, то есть складываются только те элементы, для которых в соответствующих разрядах маски стоят единицы.

В *методе с использованием укороченных векторов* главную роль играет операция сжатия, выполняемая под маской. Существует два способа сжатия. Вначале рассмотрим программу, полученную в результате применения первого способа:

```
N      COMPARE A(1:N), 0   M(1:N)
N      COMPRESS C(1:N),M(1:N) CC(1:N*d)
N*d    ADD CC(1:N*d), 10         BB(1:N*d)
N      DECOMPRESS BB(1:N*d), M(1:N)    B(1:N)
```

где **N** – размерность команд (длина); ***d*** – удельный вес «1» в маске; $(0 \le d \le 1)$

Вновь вначале выполняется сравнение векторов, в результате чего формируется маска. Затем следует команда сжатия, дающая временный укороченный вектор, т.е. такой, в котором оставлены элементы, соответствующие единицам в векторе-маске. Далее над укороченным вектором выполняется основная операция, результатом которой является также временный вектор в сжатом виде. Процесс завершается превращением сжатого вектора-результата в нормальный вид с помощью операции расширения, которая выполняется под управлением той же самой маски. Теперь рассмотрим программу, полученную в результате второго способа сжатия:

```
N      COMPARE A(1:N), 0    M(1:N)
N      GENERATE 1:N I(1:N)
N      COMPRESS I(1:N),M(1:N) II(1:N*d)
N*d    GATHER C(II(1:N*d)) CC(1:N*d)
N*d    ADD CC(1:N*d), 10          BB(1:N*d)
N*d    SCATTER BB(1:N*d)   B(II(1:N*d))
```

Вначале генерируется вектор-маска. Затем формируется вектор индексов, элементами которого являются числа 1,2,..., N , где N – длина искомого вектора. Третий шаг состоит в том, что этот вектор индексов сжимается с помощью сформированной ранее маски и превращается во временный укороченный вектор, элементы которого представляют собой индексы элементов исходного вектора, удовлетворяющих условию. Затем исходный вектор с помощью операции сборки, выполняемой под управлением укороченного индексного вектора, превращается во временный вектор, как это было в предыдущем примере. На следующем шаге над сжатым таким образом вектором выполняется основная операция. Наконец, сжатый вектор-результат после операции рассыпания, также выполняемой под управлением индексного вектора, преобразуется в нормальный вид.

Какой из двух способов, связанных с использованием укороченных векторов, эффективнее, зависит от длин векторов, с которыми работают команды, используемые в том или ином цикле. Метод маскирования позволяет обходиться меньшим числом команд, но зато каждый из них оперирует с векторами длины N. В обоих случаях использования метода укороченных векторов число команд больше, но основная операция производится над вектором длиной N*d ($0 \le d \le 1$), где d – относительное число единиц в векторе-маске. Накладные расходы (временные), связанные с обработкой укороченных векторов, зависят от того, как реализованы операции сжатия данных. В первом случае длина каждого вектора, участвующего в операции сжатия, равна N. Во втором случае для формирования вектора индексов используются три операции, каждая над векторами длины N, зато последующие операции сборки и рассыпания выполняются над векторами, имеющими длину N*d.

Таким образом, метод укороченных векторов выгоднее тогда, когда в векторе-маске имеется небольшое число единиц и число операций сжатия мало, по сравнению с числом арифметических операций.

В КС CRAY X-MP, 4-MP, 2, 3, Fujitsu (VP-100, VP-200), NEC (SX-1, SX-2) использован метод укороченных векторов.

В КС ETA Systems (ETA-10), Control Data (Cyber-205) использован метод маскирования.

Зависимости по данным в циклах.

С точки зрения распараллеливания и векторизации существует три группы зависимостей в циклах:

1. Зависимости, не препятствующие векторизации и распараллеливания.
2. Зависимости, препятствующие векторизации и распараллеливанию. Для их устранения необходима трансформация тела цикла.
3. Рекурсивные зависимости, которые векторизацию и параллельное выполнение цикла делают невозможным.

Рассмотрим примеры, иллюстрирующие зависимости каждой из перечисленных групп.

Примеры зависимостей, не препятствующих векторизации и распараллеливания:

a) Зависимость по данным внутри одной итерации цикла.

```
FOR i:=1 TO n DO
  A(i):=C(i);
  B(i):=A(i);
END;
```

Как видно из приведенного примера, такая зависимость не препятствует как векторизации, так и распараллеливанию.

b) Зависимости по данным между операторами, выполняемыми на различных итерациях цикла.

```
FOR i:=2 TO n DO
  A(i):=C(i);
  B(i):=A(i-1);
END;
```

В данном случае векторизация возможна, а распараллеливание нет, поскольку существует связь предыдущей и последующей итераций.

c) Зависимости по данным между операторами для вложенных циклов.

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    A(i,j):=C(i,j);
    B(i,j):=A(i,j-1);
  END;
END;
```

В данном примере по внешнему циклу (по i) распараллеливание и векторизация возможны, а по внутреннему (по j) векторизация возможна, а распараллеливание нет.

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    A(i,j):=C(i,j);
    B(i,j):=A(i-1,j-1);
```

```
    END;
  END;
```

В приведенном примере векторизация возможна, а распараллеливание по внутреннему циклу возможно, а по внешнему нет.

Примеры зависимостей, препятствующих векторизации и распараллеливанию:

а) Зависимости по данным между операторами, выполняемыми на различных итерациях цикла (для одиночного цикла).

```
FOR i:=1 TO n DO
  A(i):=C(i);
  B(i):=A(i+1);
END;
```

В данном примере имеется зависимость между текущей и следующей итерациями цикла, то есть между i и i+1 итерациями цикла, которая препятствует как его распараллеливанию, так и векторизации. Однако векторизация может стать возможной, если выполнить трансформацию исходного цикла путем переупорядочивания его операторов:

```
FOR i:=1 TO n DO
  B(i):=A(i+1);
  A(i):=C(i);
END;
```

Теперь векторизация возможна.

Рассмотрим пример, когда существует зависимость по памяти (циркуляционная):

```
FOR i:=1 TO n DO
  A(i):=B(i);
  B(i):=A(i+1);
END;
```

В приведенном примере трансформация цикла путем переупорядочения операторов на даст желаемого результата. В данном случае проблема разрыва зависимости может быть решена с помощью введения дополнительного массива .

```
FOR i:=1 TO n DO
  D(i):=B(i);
  A(i):=D(i);
  B(i):=A(i+1);
END;
```

Введение дополнительного массива **D** делает возможным векторизацию данного цикла.

а) Зависимости по данным между операторами, выполняемыми на различных итерациях для вложенных циклов.

```
FOR i:=1 TO n DO
  FOR j:=1 TO n-1 DO
    A(i+1,j):= A(i,j)+10;
  END;
END;
```


Рассмотрим данный пример подробнее Пусть имеется массив А вида

$$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Тогда в соответствии с приведенным выше циклом новые элементы массива А формируются в таким порядке:

$$A(2,1) \to A(1,1)+10=11 \quad (1)$$
$$A(3,1) \to A(2,1)+10=21 \quad (2)$$
$$A(2,2) \to A(1,2)+10=12 \quad (3)$$
$$A(3,2) \to A(2,2)+10=22 \quad (4)$$
$$A(2,3) \to A(1,3)+10=13 \quad (5)$$
$$A(3,3) \to A(2,3)+10=23 \quad (6)$$

Анализируя представленную последовательность операторов, можно сделать вывод, что сначала могут выполняться одновременно (1),(3) и(5) операторы , а затем (2),(4) и (6) операторы. Другими словами, полная векторизация и распараллеливание невозможны, однако возможна частичная по j. Для этого заменим местами заголовки цикла **FOR** и получим:

```
FOR j:=1 TO n-1 DO
  FOR i:=1 TO n DO
    A(i+1,j):= A(i,j)+10;
  END;
END;
```

Рекурсивные зависимости в циклах.

Рассмотрим пример рекурсивной зависимости в цикле:

```
FOR j:=1 TO n DO
  A(j):= A(j-1)+10;
END;
```

По своей природе такие циклы являются «чисто» последовательными, потому не векторизуются и не распараллеливаются.

Основные условия векторизации и распараллеливания циклических участков программ

Все существующие методы векторизации и распараллеливания работают при соблюдении следующих условий и ограничений:

1. В цикле не должно быть недопустимых зависимостей по данным или по памяти.
2. Цикл должен быть тесно-гнездовым, то есть все операторы **DO** завершаются одним оператором и между операторами **DO** не должно быть никаких других операторов.
3. Минимальные и максимальные значения параметров циклов должны быть целочисленными выражениями, а шаг (приращение) должен быть целой константой.
4. В теле цикла не должно быть операторов ввода/вывода, передачи управления из цикла, вызова подпрограмм и функций.
5. Не должна применяться нелинейная и косвенная индексация массивов в теле цикла.

11.3.2 Методы распараллеливания и векторизации циклов

Методы распараллеливания и векторизации циклов имеют различную степень универсальности и ориентированы на различные типы КС. Так, метод гиперплоскостей ориентирован на векторно-конвейерные и МКМД КС, метод координат - на матричные и векторные КС а метод пирамид – на МКМД системы. Рассмотрим каждый из перечисленных методов.

Метод гиперплоскостей Общая постановка задачи распараллеливания циклов состоит в следующем. Пусть имеется тесно-гнездовой цикл вида:

```
FOR  I1 = 1 TO r1 DO
  FOR I2 = 1 TO r2 DO
    FOR In = 1 TO rn DO
      T(I1, I2,…, In) -тело цикла представляет собой
функцию от индексов.
    END;
  END;
END;
```

где r_k, k=1,n - конечные границы индексов I_k.

Все множество итераций такого цикла называется пространством итераций. Задача распараллеливания цикла ставится как задача разбиения пространства итераций на такие подпространства (подмножества), что

итерации каждого из них могут быть выполнены одновременно, при этом сохраняется порядок информационных связей исходного цикла .Между подпространствами действия выполняются последовательно. Решение этой задачи распадается на две. Первая задача – выявление зависимостей (информационных и по памяти) между итерациями, вторая – сборка на основе выявленных зависимостей итераций в отдельные ветви.

Поясним смысл метода гиперплоскостей на примере. Пусть дано гнездо цикла вида:

```
FOR  I = 1 TO l DO
 FOR  J = 2 TO m DO
  FOR K = 2 TO n DO
   X(J,K) = f((J+1,K), X(J,K+1), X(J-1,K), X(J,K-1))
  END;
 END;
END.
```

Каждый элемент матрицы **X** является функцией от 4-х соседних элементов. Элемент **X(J,K)** переопределяется l раз.

Пространство итераций этого гнезда циклов трехмерно и все итерации являются целочисленными точками параллелепипеда [1,l]*[1,m]*[1,n] (см. рис.11.3.2.).

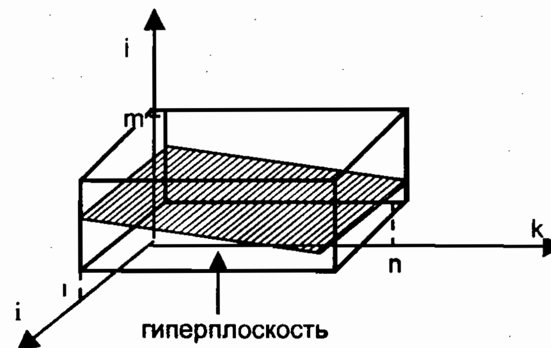


Рис.11.3.2. Геометрическое представление пространства итераций, применяемое в методе гипреплоскостей

Целью данного метода является поиск в пространстве итераций такого семейства гиперплоскостей, то есть в нашем случае обычных двумерных плоскостей, чтобы точки итераций, лежащие на ней, удовлетворяли бы условиям независимости. Идеальным вариантом является нахождение плоскостей, параллельных одной из осей координат. Однако, анализ зависимостей точек **X** показывает, что в таких плоскостях мы не можем одновременно просчитывать множество входящих в них точек **X** (между ними существует информационная зависимость). Таким образом, необходимо найти какие-либо наклонные плоскости (гиперплоскости), которые бы в результате покрыли все точки нашего параллелепипеда (пространства итераций). С помощью специального математического аппарата на основании зависимости между координатами выводятся уравнения плоскостей. Циклам с различной структурой соответствуют различные уравнения.

Для нашего случая возьмем плоскость, определяемую равенством:

$$2I + J + K = const$$

Значение **const** должно изменяться до тех пор, пока не будут найдены все точки пространства итераций.

В связи с введением уравнения плоскости должны быть введены новые индексные переменные в гнездо циклов.

$$I' = 2I + J + K$$
$$J' = I' - 2J' - K'$$
$$K' = K$$

Далее переопределяем наши координаты:

```
FOR  I' = 6 TO 2l + m + n DO
  FOR J', K' DO PAR (параллельно для всех J', K')
    X(J',K') = f(X(J'+1,K'), X(J',K'+1), X(J'-1,K'),
X(J,K'-1))
  END;
END.
```

В этом случае на каждом шаге внешнего цикла **I'** (для заданной гиперплоскости), изменяющемся от **6** до **2l + m + n**, параллельно выполняются операции над всеми элементами массива, входящими в данную гиперплоскость. Другими словами внешний цикл "перебирает" все гиперплоскости, пересекающие пространство итераций [1,l]*[1,m]*[1,n], а внутри каждого обращения этого цикла все итерации принадлежащие плоскости, выполняются одновременно и независимо. Минимальное значение **I'** определяется подстановкой минимальных значений **I, J, K** в равенство **I'=2I + J + K**. Множество итераций, принадлежащих каждой плоскости, определяется перебором координат **I, J, K** из заданного множества, удовлетворяющим значениям констант гиперплоскостей. Так,

например, плоскости **2I + J + K = 6** принадлежит единственный набор координат

I	J	K
1	2	2

и, соответственно, единственная итерация: при **I = 1** определяется **X(2,2)**.

Следующей плоскости **2I + J + K = 7** принадлежит два набора координат

I	J	K
1	2	3
1	3	2

и, соответственно, две независимые итерации при **I=1 : X(2,3)** и **X(3,2)**, а плоскости **2I + J + K = 10** уже принадлежат восемь различных наборов координат

I	J	K
1	2	6
1	6	2
1	3	5
1	5	3
1	4	4
2	2	4
2	4	2
2	3	3

что соответствует одновременному выполнению восьми итераций: при I=1: X(2.6); X(6.2); X(3.5); X(5.3); X(4.4) и при I=2: X(2.4); X(4.2); X(3.3).

Метод координат. Данный метод является модификацией метода гиперплоскостей и дает эффективные результаты для векторных и матричных систем. Суть его состоит в том, чтобы используя перестановки операторов тела цикла и переименования некоторых переменных, добиться возможности расположения гиперплоскости параллельно одной из координат.

В этом методе все операторы тела цикла выполняются строго последовательно, но каждый из них выполняется не над одним элементом массива, а над всем массивом или его частью (т.е. преобразуется в

векторную команду (оператор)). Проиллюстрируем это на примере. Пусть задан последовательный цикл

```
FOR i = 2 TO m DO
  FOR j = 1 TO n DO
    x(i,j) = y(i,j) + z(i);
    z(i) = y(i-1,j);
    y(i,j) = x(i+1,j);
  END;
END.
```

Для того, чтобы применить метод координат, необходимы следующие преобразования гнезда цикла: поменять между собой местами два оператора заголовка цикла (1 и 2 операторы), ввести дополнительный вектор **u(i)** для хранения **x(i+1,j)**, поменять местами операторы **z(i)** и **y(i,j)**. В результате получим такой цикл

```
FOR j = 1 TO n DO
  FOR i ∈ 2 TO m синхронно DO
    u(i)= x(i+1,j)
    x(i,j) = y(i,j) + z(i);
    y(i,j) = u(i);
    z(i) = y(i-1,j);
  END;
END.
```

Этот цикл векторизуется по **i**, то есть каждый из операторов является векторной операцией для **i ∈ 2,m**. Данный цикл выполняется **n** раз (**j** = 1, ..., **n**).

Метод пирамид. Целью данного метода распараллеливания циклов является полного исключения пересылок в КС. Поэтому метод пирамид ориентирован на МКМД системы с распределенной памятью, в которых синхронизация и обмен данными между вычислительными ветвями приводит к большим накладным затратам. Примером его применения является компилятор для КС «Эльбрус».

В соответствии с этим методом распараллеливание цикла сводится к его преобразованию во множество автономных ветвей. Ветвь – последовательность итераций. Вся необходимая информация для каждой ветви вычисляется внутри нее.

Самый простой метод формирования автономных ветвей заключается в следующем. В цикле выбираются все результирующие итерации, т.е. итерации, вырабатывающие некоторые данные, не используемые никакими другими далее в теле цикла. Каждая итерация служит основой для

отдельной параллельной ветви и является последней в этой ветви. Затем к каждой из них присоединяются все итерации, непосредственно формирующие для нее исходные данные и т.д. до тех пор, пока каждая ветвь не будет содержать всю «историю» вычислений результирующей итерации (просмотр снизу вверх). Таким образом, мы можем построить полностью автономные ветви (без пересылок), однако некоторые итерации при таком подходе будут дублироваться в различных ветвях (рис.11.3.3.). Поэтому данный метод является эффективным тогда, когда потери времени на синхронизацию и обмен данными превышают время на дублирование вычислений. Кроме того, сформированные ветви итераций могут быть различной длины, а значит трудоемкости, что может негативно сказаться на эффективности использования процессорных ресурсов. Поэтому для повышения эффективности использования процессорных ресурсов после порождения множества ветвей некоторые из них (короткие) могут быть склеены (искусственно с помощью компилятора) для того, чтобы длина всех ветвей могла быть одинаковой.

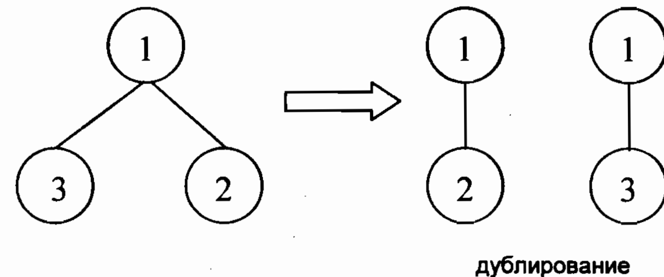


Рис. 11.3.3. Преобразорвание цикла вследствие применения метода пирамид

11.4 Разбиение программы на асинхронные параллельно выполняемые процессы

Прежде всего следует отметить, что, если методы распараллеливания и векторизация циклов широко используются при построении компиляторов, то методы автоматического распараллеливания на

асинхронно выполняемые процессы (уровень независимых ветвей) представляют в основном теоретический интерес.

В общем случае программа может быть представлена как последовательность циклов, либо разветвленная программа, включающая вычислительные операторы и циклические участки.

В первом случае суть разбиения программы на асинхронные параллельно выполняемые процессы заключается в следующем. Каждый цикл (из заданного множества m) распараллеливается с помощью метода пирамид, в результате чего формируются асинхронные ветви (рис.11.4.1.).

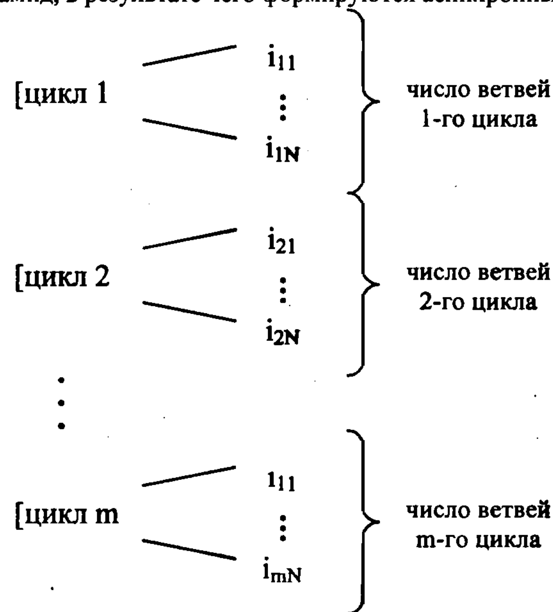


Рис.11.4.1. Последовательность циклов после применения метода пирамид

Далее формируется граф параллельной формы, с числом ярусов, равным количеству последовательных циклов. На каждом i-ом ярусе содержится число вершин, равное количеству ветвей i-ого цикла (рис.11.4.2.).

Во втором случае для разбиения программы на асинхронные параллельные процессы может быть применен метод гамаков, описанный ранее. Отличительной особенностью данного варианта является то, что элементом программы могут быть не только отдельные операторы, но и

циклические участки, которые включаются в ветви с помощью метода гамаков (рис.11.4.3.). Понятно, что метод гамаков, как и метод пирамид, не дает гарантии получения оптимального разбиения на ветви (критерий оптимизации – минимальное время выполнения программы), так как существует множество вариантов объединения в гамаки. Для того чтобы приблизить полученный результат разбиения к оптимальному целесообразно преобразовать полученный граф программы путем объединения некоторых ветвей (процессов) с целью выравнивания их трудоемкости, а также приближения количества ветвей к значению, кратному числу процессорных элементов. При этом все процессы, выполняемые до каждого из объединяемых процессов, должны быть выполнены к моменту начала выполнения объединенных ветвей. Аналогично все процессы, которые выполнялись после каждого из объединяемых процессов, могут начинать свое выполнение лишь после реализации объединенного процесса. Объединение двух процессов в один приводит к сокращению суммарного времени работы на время τ реализации команд образования и объединения процессов. Необходимо генерировать такие процессы программы, чтобы для их реализации на конечном числе процессоров Q требовалось минимальное время.

Иными словами, необходимо найти Q объединений ветвей приблизительно равной продолжительности, в которые входят все исходные ветви.

Для определения оптимальной структуры программы в общем случае предложен эвристический алгоритм, позволяющий с помощью итеративного метода синтезировать структуру программы с достаточно хорошим приближением к минимальному времени выполнения.

В поставленной задаче появляется возможность изменения структуры графа, в зависимости от которой, изменяется и время выполнения графа. Суть предложенного алгоритма заключается в проверке целесообразности объединения соседних процессов и в итеративном процессе их объединения. Резерв времени для каждого процесса определяется как разность между поздним и ранним сроками начала выполнения процессов без изменения общего времени выполнения графа. Объединение возможно производить для ветвей, исходящих или входящих в одну вершину.

Объединение проводится, если резерв времени одной ветви больше времени выполнения второй ветви, с которой она объединяется. Процесс продолжается итеративно до тех пор, пока уменьшается время выполнения программ.

На рис. 11.4.4 и рис.11.4.5 представлены фрагменты графов программ и условия объединения ветвей, для которых может рассматриваться описанный выше алгоритм.

11 — 2-3006

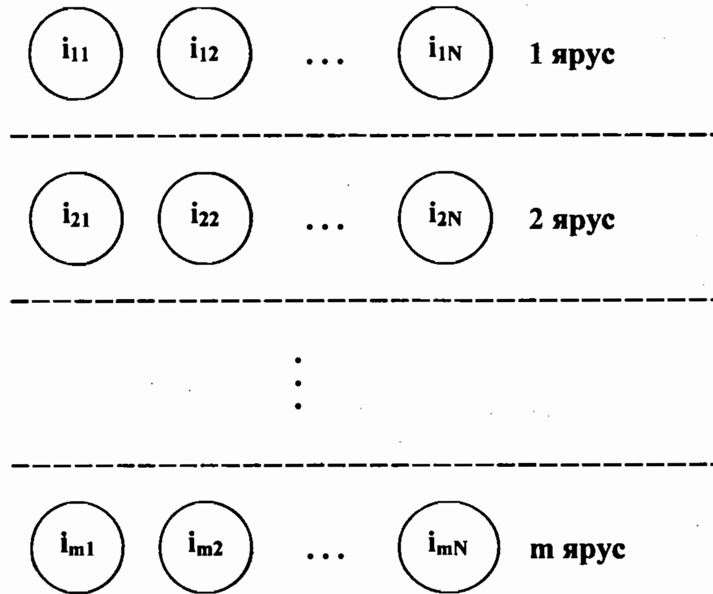


Рис.11.4.2. Граф параллельной формы, соответствующий программе, состоящей из последовательности циклов на рис.11.4.1.

Рассмотрим фрагмент графа на рис.11.4.4. Пусть для 2-й вершины ранний срок ($t_{рс2}$), поздний срок ($t_{пс2}$) и ее вес (W_2) соответственно равны: $t_{рс2'}=i$; $t_{пс2'}=j$; $W_2=Z_2$, а для 3-й вершины : $t_{рс3'}=k$; $t_{пс3'}=l$; $W_3=Z_3$ Тогда для объединенной вершины ранний, поздний сроки ее выполнения, а также вес соответственно определяются, как: $t_{рс2+3}=\max\{i, k\}$; $t_{пс2+3}=\min\{j, l\}$; $W_{2+3}=Z_2+Z_3$. Объединение имеет смысл, если резерв времени объединенной вершины не превышает ее суммарный вес, т.е. $t_{пс2+3} - t_{рс2+3} \leq Z_2+Z_3$. Аналогичным образом определяются ранние, поздние сроки выполнения вершин, а также условия их объединения для фрагмента графа, изображенного на рис.114.5. Рассмотрим пример объединения ветвей фрагмента программы на рис. 11.4.6. В данном случае объединение возможно, так как $t_{пс2+3} - t_{рс2+3}=8-3=5$, $W_{2+3}=5$, то есть резерв в 5 тактов достаточен для выполнения дополнительных тактов объединенной вершины.

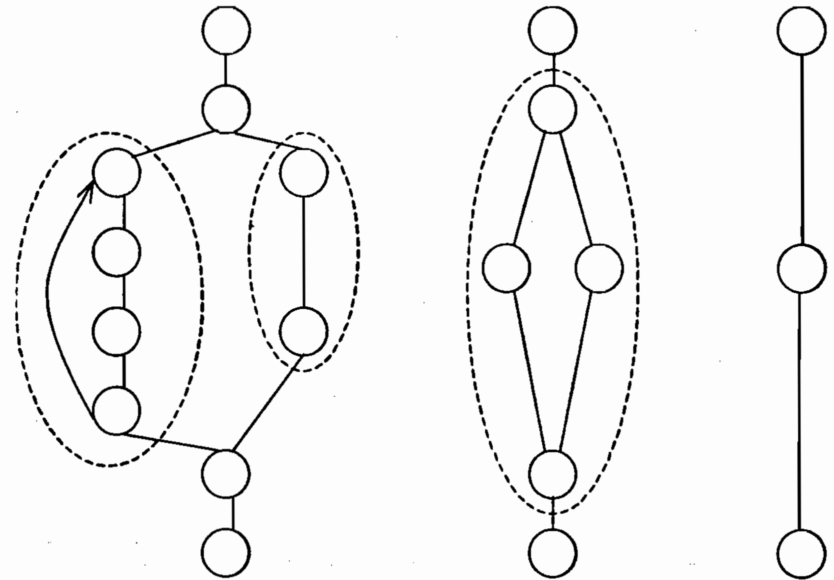


Рис.11.4.3. Преобразование разветвленной программы с циклами к линейной форме посредством метода гамаков

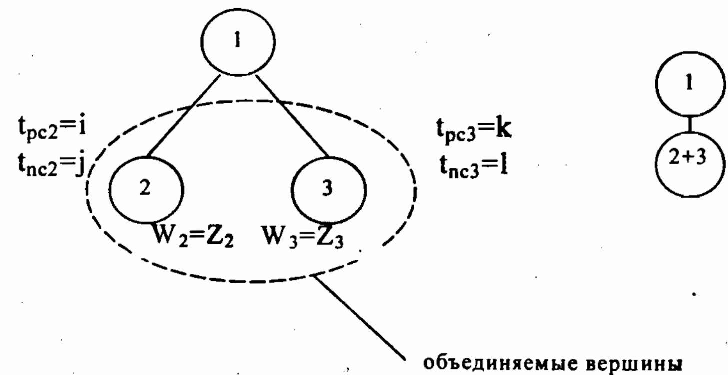


Рис.11.4.4. Фрагмент графа программы до и после объединения

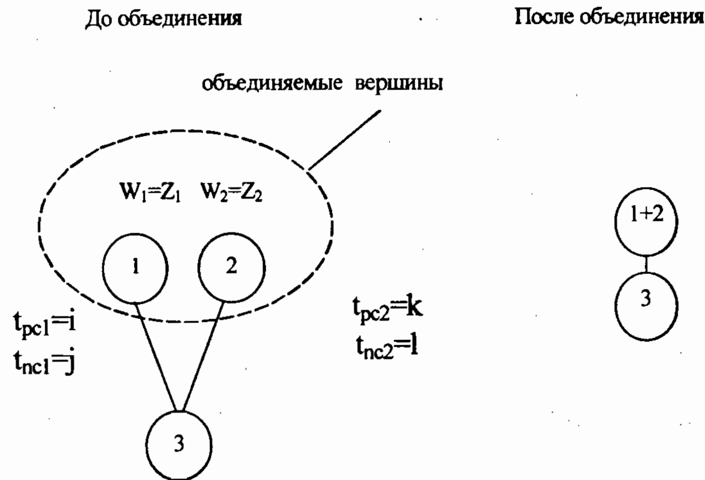


Рис.11.4.5. Фрагмент графа программы до и после объединения

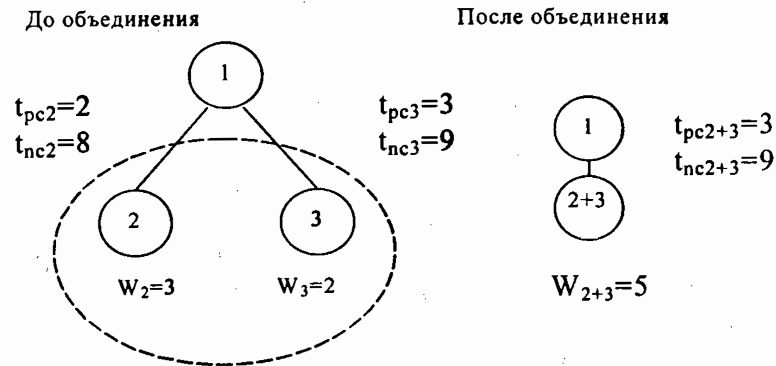


Рис.11.4.6. Пример фрагмента графа программы с возможностью объединения ветвей

11.5 Распараллеливание программ для КС, управляемых потоком данных

11.5.1 Граф потока данных

Как отмечалось выше, в КС, управляемых потоком данных (Dataflow системах) порядок выполнения операторов в программе определяется динамически (выполняются те операторы, для которых подготовлены данные). В каждый момент времени осуществляется проверка условий готовности операторов к работе и выполнение тех из них, которые удовлетворяют условиям готовности. Такой способ управления позволяет сохранить при составлении программы и легко использовать при выполнении естественный параллелизм, присущий любой задаче. При этом полностью исключается проблема синхронизации, так как выполняются всегда только те операторы, для которых уже закончился процесс вычисления необходимых данных. Такой подход сильно отличается от традиционного, когда обычные программы считаются по умолчанию последовательными, использующими «естественный» порядок выполнения команд. Для потоковых систем используется противоположный подход – любая программа считается по умолчанию параллельной, несмотря на то, что для составления таких программ применяются традиционные (последовательные) языки программирования. Распараллеливание же производится динамически, вначале на стадии компиляции, а затем в процессе выполнения программ.

В Dataflow системах в качестве модели вычислений используется граф потока данных. Вершинам такого графа соответствуют команды (и соответствующие им операции), а ребрам – данные для них. При таком представлении связь между командами программы осуществляется только через данные. Операции, указанные в вершинах, выполняются только над данными на входах вершин и не зависят от соседних вершин. Ребра графа имеют обозначения, соответствующие значениям данных. Эти значения данных представляют собой промежуточный результат вычислений, который еще не был использован. Появление необходимых для выполнения данной команды операндов приводит к выполнению этой команды. Для графа потока данных нет понятия арифметического выражения, программы линейной или с ветвлениями, а есть поток команд унарных или бинарных. Граф строится слева направо. Верхний вход – левый операнд, нижний вход – правый операнд. Перед началом выполнения программы определены только исходные данные и константы. Команда становится активной, когда все ребра, входящие в соответствующую данной команде вершину графа, имеют значение. Команда становится активизированной, когда хотя бы одно из ребер,

входящих в вершину, имеет значение. Команда неактивна, когда ни одно из входящих в нее ребер не имеет значения.

Рассмотрим пример графа потока данных. Пусть имеется система, состоящая из двух линейных алгебраических уравнений вида.

$$a_{11} * x_1 + a_{12} * x_2 = b_1;$$

$$a_{21} * x_1 + a_{22} * x_2 = b_2;$$

В случае, если

$$\Delta = a_{11} * a_{22} - a_{12} * a_{21} \neq 0$$

то данная система имеет решение

$$x_1 = \frac{1}{\Delta} * (a_{22} * b_1 - a_{12} * b_2)$$

$$x_2 = \frac{1}{\Delta} * (a_{11} * b_2 - a_{21} * b_1)$$

которому соответствует схема алгоритма на рис.11.5.1.

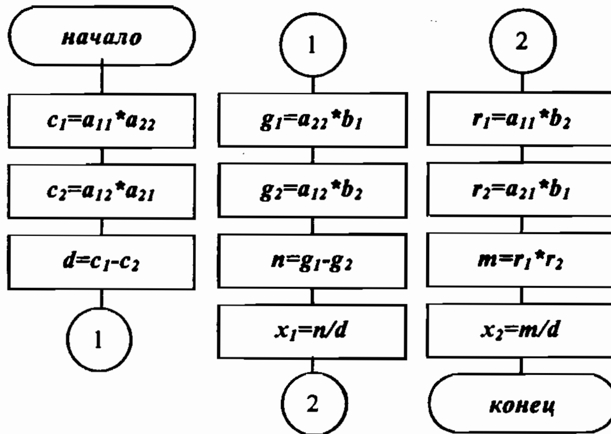


Рис.11.5.1. Схема алгоритма решения системы из двух линейных алгебраических уравнений

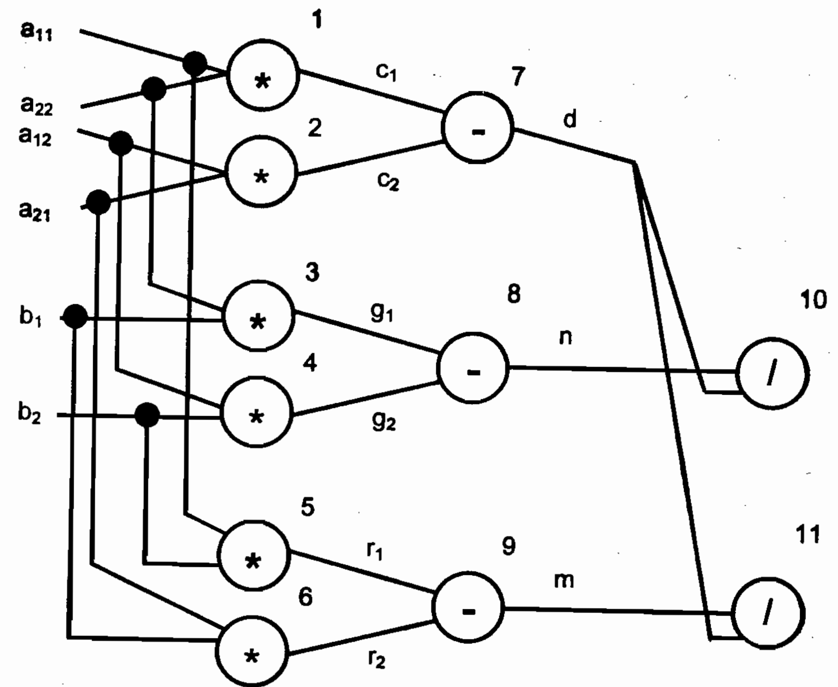
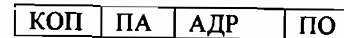


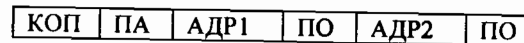
Рис 11.5.2. Граф потока данных, соответствующий решению системы двух линейных алгебраических уравнений

11.5.2 Форматы команд и данных для потоковых систем

Рассмотрим структуру команд и данных в потоковой системе, реализующей программы, представленные в виде графа потока данных. Как правило, такие команды состоят из четырех или шести полей (рис.11.5.3).



КОП	ПА	АДР	ПО



КОП	ПА	АДР1	ПО	АДР2	ПО

Рис. 11.5.3. Форматы команд для потоковой системы

Поле КОП содержит код операции, поле ПА – признак активности заданной команды (0 – неактивная , 1 – активная или активизированная).

Поле АДР содержит адрес (адреса) последующей команды (команд), поле ПО – признак очередности операндов (0 – левый операнд , 1 – правый операнд). Например, для команд, показанных на рис.11.5.2 команда 1 будет выглядеть следующим образом:

*	1	7	0

а команда 7 будет иметь вид:

–	0	10	1	11	1

Значения полей АДР1 и АДР2 –10 и 11 – это номера следующих (за командой 7) команд.

Формат данных имеет следующий вид:

РЗ	АДР	ПО

Поле РЗ содержит результат выполнения команды, а поля АДР и ПО, соответственно, адрес команды, в которой используется это данное и признак очередности операндов. Таким образом, данное, вычисленное командой 1, будет выглядеть так:

$a_{11} * a_{22}$	7	0

Выполнение команды осуществляется в два этапа: на первом этапе по входным значениям данных формируется результат. При этом в поле АДР и ПО результата помещается содержимое полей АДР и ПО команды, а в поле РЗ помещается результат выполнения указанной в поле КОП операции Признак активности в команде устанавливается в состояние 0 (команда становится неактивной). На втором этапе осуществляется вызов и передача значения результата команде (командам), адрес (а) которой (-ых) указан (-ы) в поле АДР.

Например, до выполнения команда 3 (рис.11.5.2) выглядит так:

*	0	8	0

а результат команды отсутствует (не имеет значения). После выполнения команды 3 появится результат

$a_{22} * b_1$	8	0

До выполнения команда 7 выглядит так:

-	0	10	0	11	0

В результате выполнения команды 7 формируются данные (результаты):

$a_{11} * a_{22}$	10	0

$a_{11} * a_{22}$	11	0

11.5.3 Структура потоковой системы

Рассмотрим один из возможных вариантов структурных решений потоковых КС (рис.11.5.4.).

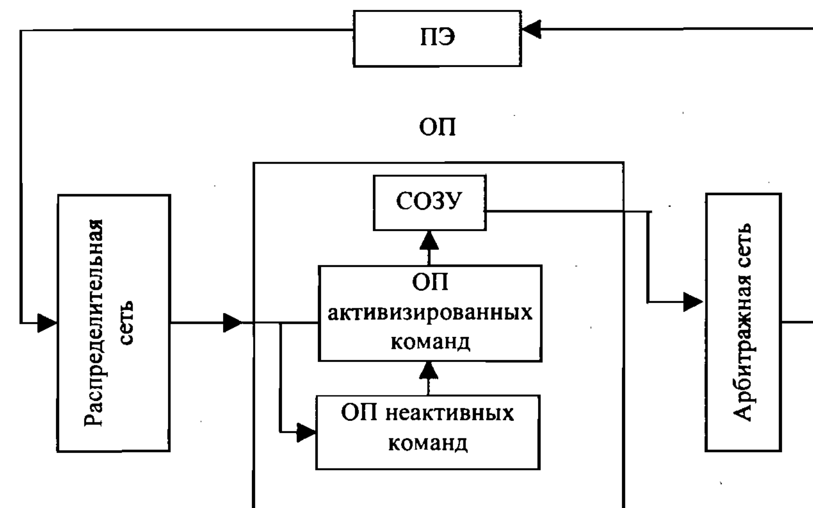


Рис.11.5.4. Структура системы, управляемой потоком данных

Оперативная память (ОП) такой системы состоит из трех компонент: ОП неактивных команд, для которых ни один операнд не готов; ОП активизированных команд, для которых готов только один операнд и СОЗУ для хранения активных команд (готовы все операнды). Собственно операции выполняются с помощью набора процессорных элементов (ПЭ).

Связь между ОП и ПЭ осуществляется с помощью двух сетей пересылок: арбитражная сеть служит для передачи пакетов готовых команд ПЭ, а распределенная сеть – для пересылки результатов в те командные ячейки ОП, в которых эти результаты используются уже как операнды. Начальное заполнение компонент оперативной памяти производится по результатам компиляции программы. Вначале выполняются команды, зависящие только от исходных данных и констант. Дальнейшее выполнение программы осуществляется следующим образом: после вычисления результата очередной команды вначале происходит обращение к командным ячейкам ОП активизированных команд. В качестве ключа поиска в данной ассоциативной памяти выступает значение поля адреса выполненной команды. В случае, если данная команда находится в памяти активизированных команд, осуществляется ее передача вместе с необходимыми для ее выполнения операндами в СОЗУ. В случае отсутствия этой команды в памяти активизированных команд, производится ее поиск в ОП неактивных команд и последующее ее перемещение в память активизированных команд.

Максимальная производительность системы определяется быстродействием и числом ПЭ, а также пропускными способностями памяти и сетей пересылок.

11.5.4 Компиляция программ для потоковых систем

Компиляция программ для потоковой системы означает преобразование программы на традиционном языке в последовательность потоковых команд, представленных в виде графа потока данных. Такой процесс компиляции состоит из следующих этапов:

1). По исходному тексту получение программы в тетрадном виде (стандартная компиляция).

2). Формирование таблиц операндов.

3). По таблице операндов получение последовательности команд в формате потоковых систем.

Рассмотрим этапы компиляции программы для потоковой КС на примере решения системы линейных алгебраических выражений. Схема алгоритма данной задачи представлена на рис.11.5.1. , а ее граф потока данных – на рис.11.5.2.

Последовательности команд в тетрадном виде в данном случае имеет следующий вид:

№ ком.	КОП	Операнд 1	Операнд 2	Результат
1	ввод	-	-	a_{11}
2	ввод	-	-	a_{12}
3	ввод	-	-	b_1
4	ввод	-	-	a_{21}
5	ввод	-	-	a_{22}
6	ввод	-	-	b_2
7	*	a_{11}	a_{22}	c_1
8	*	a_{12}	a_{21}	c_2
9	*	a_{22}	b_1	g_1
10	*	a_{12}	b_2	g_2
11	*	a_{11}	b_2	r_1
12	*	a_{21}	b_1	r_2
13	–	c_1	c_2	D
14	–	g_1	g_2	N
15	–	r_1	r_2	M
16	/	N	d	x_1
17	/	M	d	x_2
18	вывод	x_1		
19	вывод	x_2		

Формирование таблицы операндов производится следующим образом. Для каждого идентификатора операнда осуществляется запись номера команды, использующей этот операнд, в соответствующее поле таблицы операндов. Заполненная таким образом таблица имеет такой вид:

Операнд	Ком. 1	ПО	Ком. 2	ПО
a_{11}	7	0	11	0
a_{12}	8	0	10	0
a_{21}	8	1	12	0
a_{22}	7	1	9	0
b_1	9	1	12	1
b_2	10	1	11	1
c_1	13	0		
c_1	13	1		
g_1	14	0		
g_2	14	1		
r_1	15	0		
r_2	15	1		
n	16	0		
d	16	1	17	1
m	17	0		
x_1	18	0		
x_2	19	0		

Формирование последовательности команд в формате потоковых систем производится следующим образом. По идентификаторам результатов осуществляется обращение в таблицу операндов и поля, содержащие номера команд, переписываются в просматриваемую строку программы. Например, в команде 13 стоит идентификатор результата d Обращаемся к таблице операндов по этому идентификатору. В таблице операндов в строке, содержащей d, находятся номера команд 16 и 17. Эти номера переписываем в строку команды 13 после идентификатора d. Таким образом, откомпилированный текст программы имеет следующий вид:

№ ком.	КОП	ПА	Адрес1	ПО	Адрес2	ПО
1	ввод	1	7	0	11	0
2	ввод	1	8	0	10	0
3	ввод	1	9	1	12	1
4	ввод	1	8	1	12	0
5	ввод	1	7	1	9	0
6	ввод	1	10	1	11	1
7	*	1	13	0		
8	*	1	13	1		
9	*	1	14	0		
10	*	1	14	1		
11	*	1	15	0		
12	*	1	15	1		
13	–	0	16	1	17	1
14	–	0	16	0		
15	–	0	17	0		
16	/	0	18	0		
17	/	0	19	0		

Как видно из полученной таблицы, команды 1-12 – активные, а 13-17 – неактивные. Активные команды могут быть выполнены параллельно в различных процессорах потоковой системы. Таким образом, в процессе компиляции выполняется начальный этап распараллеливания программы, который затем динамически продолжается во время ее выполнения.

Литература

1. H. El-Rewini, T.G.Lewis Distributed and Parallel Computing.-Manning Publications Co.,1998.-447 p.
2. А. Ахо, Р. Сети, Дж.Ульман Компиляторы: принципы, технологии и инструменты,-М.:Изд. Дом “Вильямс”,2001.-768 с.
3. Бройнль Томас. Паралельне програмування: Початковий курс: Навч.посібник.-К.:Вища школа,1997.-358 с.
4. В.А. Вальклвский, В.Е.Котов, А.Г.Марчук, Н.Н.Миренков. Элементы параллельного программирования.-М.:Радио и связь,1983.-240 с.
5. Высокоскоростные вычисления. Архитектура, производительность, прикладные алгоритмы и программы суперЭВМ. Под ред. Я,Ковалика.-М.:Радио и связь,1988.- 432 с.
6. Системы параллельной обработки / Под ред.Д.Ивенса.-М.:Мир,1985-413 с.
7. Корнеев В.В. Параллельные вычислительные системы.-М.: “Нолидж”, 1999.-320 с.
8. СуперЭВМ. Аппаратная и программная организация / Под ред. С.Фернбаха.-М.:Радио и связь,1991.-320 с.
9. Б.Ю.Сырков, С.В.Матвеев. программное обеспечение мультитранспьютерных систем.-М.: “ ДИАЛОГ-МИФИ ”,1992.-224 с.
10. Транспьютеры, Архитектура и программное обеспечение / Под ред. Г.Харпа.-М.:Радио и связь,1993.-304 с.
11. Э.А.Трахтенгерц Программное обеспечение параллельных процессов.М.:Наука,1987.-272 с.
12. Хокни Р., Джессхоуп К. Параллельные ЭВМ. Архитектура, программирование и алгоритмы.-М.:Радио и связь,1985.-358 с.