**Вопрос 4 (**[**ссылка**](http://www.hpcc.unn.ru/files/HTML_Version/part4.html)**)**

**Вычисление частных сумм числовой последовательности.**

Рассмотрим для первоначального ознакомления со способами построения и анализа параллельных методов вычислений сравнительно простую *задачу нахождения частных сумм* последовательности числовых значений

,

где   есть количество суммируемых значений (данная задача известна также под названием *prefix sum problem* – см. п. 3.3).

Изучение возможных параллельных методов решения данной задачи начнем с еще более простого варианта ее постановки – с *задачи вычисления общей суммы* имеющегося набора значений (в таком виде задача суммирования является частным случаем общей *задачи редукции* – см. п. 3.3.)

.

**Последовательный алгоритм суммирования**

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



Вычислительная схема данного алгоритма может быть представлена следующим образом (см. рис. 4.1):

,

где  есть множество операций суммирования (вершины  обозначают операции ввода, каждая вершина , , соответствует прибавлению значения   к накапливаемой сумме ), а



есть множество дуг, определяющих информационные зависимости операций.

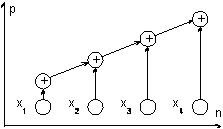


Рис. 4.1. Последовательная вычислительная схема алгоритма суммирования

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

**Каскадная схема суммирования**

Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования (известный в литературе как *каскадная схема*) состоит в следующем (см. рис. 4.2):

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

Данная вычислительная схема может быть определена как граф (пусть )

,

Изображение выглядит как ночное небо

Автоматически созданное описание

Рис. 4.2. Каскадная схема алгоритма суммирования

где   есть вершины графа ( - операции ввода,  - операции первой итерации и т.д.), а множество дуг графа определяется соотношениями:

.

Как можно оценить, количество итераций каскадной схемы оказывается равным величине

,

а общее количество операций суммирования



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

.

Как результат, можно оценить показатели ускорения и эффективности каскадной схемы алгоритма суммирования





где   есть необходимое для выполнения каскадной схемы количество процессоров.

Анализируя полученные характеристики, можно отметить, что время параллельного выполнения каскадной схемы совпадает с оценкой для паракомпьютера в теореме 2 (см. раздел 2). Однако при этом эффективность использования процессоров уменьшается при увеличении количества суммируемых значений

.

**Модифицированная каскадная схема**

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы [18]. В новом варианте каскадной схемы все проводимые вычисления подразделяется на два последовательно выполняемых этапа суммирования (см. рис. 4.3):

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

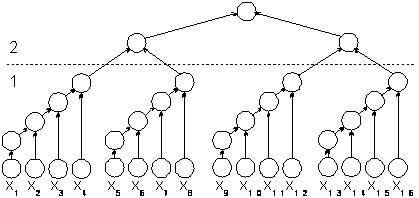


Рис. 4.3. Модифицированная каскадная схема суммирования

Для упрощения построения оценок можно предположить . Тогда для выполнения первого этапа требуется выполнение  параллельных операций при использовании  процессоров. Для выполнения второго этапа необходимо



параллельных операций для  процессоров. Как результат, данный способ суммирования характеризуется следующими показателями:

, .

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





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

 .

Можно отметить также, что данные значения показателей достигаются при количестве процессоров, определенном в теореме 5 (см. раздел 2).

**Вычисление всех частных сумм**

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

.

При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемым результатам; достижение эффективного распараллеливания требует привлечения новых подходов (может даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения задач. Так, для рассматриваемой задачи нахождения всех частных сумм алгоритм, обеспечивающий получение результатов за  параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (см. рис. 4.4) [18]:

* перед началом вычислений создается копия вектора суммируемых значений ();
* далее на каждой итерации суммирования  , формируется вспомогательный вектор   путем сдвига вправо вектора  на  позиций (освобождающие при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов  и :

.

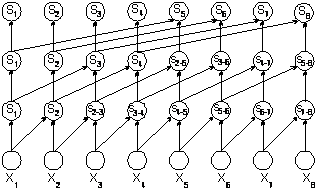


Рис. 4.4. Схема параллельного алгоритма вычисления всех частных сумм (величины  означают суммы значений от  до  элементов числовой последовательности)

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



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

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



.

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

**Вопрос 6 (**[**ссылка**](https://pro-prof.com/forums/topic/parallel-programming-paradigms)**,** [**ссылка**](https://cc.dvfu.ru/ru/361-2/)**,** [**ссылка**](https://studfile.net/preview/16404441/page:6/)**)**

**Этапы разработки параллельных программ (**[**ссылка**](https://cc.dvfu.ru/ru/361-2/)**)**

### **Декомпозиция**

Этот шаг обычно выполняется первым. Задача, которую необходимо решить, разбивается на порции, которые можно выполнить параллельно. Разбить на порции можно двумя способами:

1. разбиваем на порции все функции, которые надо выполнить (Functional);
2. разбиваем на порции, все данные, которые надо обработать или получить (Domain).

#### **Декомпозиция по функциям**

Здесь анализируются функции, которые необходимо выполнить, и разбитие выполняется по этим функциям, например, если надо вычислить определенный интеграл и корень квадратный, то эти 2 функции могут выполняться параллельно. В один класс относят функции, которые можно выполнить параллельно.

Допустим, у нас есть 2 функции. Первая читает данные с диска, а вторая – их обрабатывает. Очевидно, что такие функции не могут выполняться параллельно. Для обеспечения возможности их параллельного выполнения можно каждой из функций читать порцию данных и обрабатывать порцию после ее чтения (конвейер), в этом случае функции можно будет выполнять параллельно.

#### **Декомпозиция по данным**

Каждая параллельная задача работает с порцией данных.

Например, при вычислении определенного интеграла каждая задача обрабатывает свой интервал интегрирования.

Здесь возможно использовать несколько вариантов:

* каждая задача выполняет обработку всей порции данных целиком;
* задача выполняет обработку одного данного, а затем берет очередное необработанное данное;
* комбинация первых двух вариантов: каждая задача выполняет работу над порцией данных заданной длины, а после завершения получает очередную порцию.

Примерами такой декомпозиции являются параллельные алгоритмы вычисления суммы (каскадный и комбинированный), вычисления полинома, рассмотренные выше.

Если обрабатывается 2-х мерный массив, здесь вариантов значительно больше, т.е. обработка по строкам, столбцам и прямоугольникам.

Например, для умножения матриц можно использовать порцию строк одной матрицы и соответствующую порцию столбцов другой матрицы. Этот метод имеет недостаток, так как для второй матрицы используются несмежные данные. Если исходные матрицы и результирующая матрица вместе не помещаются в

Кеше, то в этом случае данный метод не эффективен. Определим размерность матрицы, при которой все 3 матрицы помещаются в Кеше из расчета размера Кеша 32 кБ. В этом случае на одну матрицу примерно приходится 10 кБ, т.е. 10000 байтов, если под один элемент отводится 4 байта, то одна матрица может иметь размер 2500 элементов или 50 \* 50 строк – столбцов.

Алгоритм умножения матриц может быть преобразован таким образом, чтобы во второй матрице использовались тоже строки, т.е. смежные элементы, что значительно улучшит использование Кеша. В этом случае на каждом шаге мы будем не накапливать значение одного элемента матрицы, а сразу вычислять значения всех элементов дайной строки результирующей матрицы.

### **Планирование параллельных программ**

Анализируются данные, которые используются отдельными задачами. Если есть зависимости по данным, параллельное выполнение исключается (определение зависимостей).

Если используются общие данные, то для них определяются режимы использования и выбираются необходимые элементы синхронизации. Если используется неразделяемая память, то планируются средства коммуникации между процессами.

Анализируются функции и их временная диаграмма. Если необходимо, то планируются средства синхронизации для функций с точки зрения их завершения.

На этапе планирования решаются следующие задачи:

* исследуется наличие зависимостей для данных и функций;
* в случае наличия зависимостей определяются необходимые способы синхронизации. Может быть сделан вывод о нецелесообразности параллельного выполнения;
* так как в случае параллельного выполнения время определяется временем выполнения самой длинной ветви, то рассматриваются параллельные ветви с точки зрения балансировки;
* так как создание и уничтожение потоков связано с дополнительными расходами, необходимо, чтобы время обработки параллельного потока превосходило накладные расходы (гранулярность);
* так как время ввода-вывода, как правило, не предсказуемо, необходимо проанализировать отдельно все такие операции;
* параллельные алгоритмы часто приводят к увеличению требуемых ресурсов, особенно памяти. Необходимо проанализировать реальность требований;
* будущие процессоры с большим числом ядер –масштабируемость;
* определение ожидаемых показателей параллельной программы.

#### **Зависимости данных**

Введем наборы входных и выходных переменных программы. Обозначим набор входных переменных программы R(P)R(P) (RR от слова read) – это набор входных переменных для всех ее операторов. Аналогично, набор выходных переменных программы W(P)W(P) (WW от слова write) – набор выходных переменных для всех ее операторов. Например, для программы



получаем R(P)=u,vх,w, W(P)=х,у. Заметим, что переменная х присутствует как в R(P)R(P) так и в W(Р)W(Р).

Теперь сформулируем условия Бернстайна.

Если для двух участков программ РР и QQ:

* пересечение W(P)W(P) и W(Q)W(Q) пусто,
* пересечение W(P)W(P) с R(Q)R(Q) пусто,
* пересечение R(P)R(P) и W(Q)W(Q) пусто,

тогда зависимостей гарантированно нет, участки программ можно выполнять параллельно. Таким образом, условия Бернстайна являются достаточными, но не необходимыми.

Случай двух участков программ естественным образом обобщается на их большее количество.

Условия Бернстайна проверить просто, но, к сожалению, они исключают параллельное выполнение даже в тех случаях, когда оно возможно. Так, запись в одну и ту же область памяти возможна двумя параллельными потоками, если используются соответствующие объекты синхронизации. Напоминаем, что в случае конкурентного использования общей памяти говорят, что имеет место race condition (состояние гонки). В приведенном выше примере процессы состязаются за вычисление значений переменных х и у.

#### Типы зависимостей

##### Зависимость типа «Чтение после записи»

Возникает, если оператор с номером i формирует значение некоторой переменной (записывает ее), а оператор с номером j (j>i) использует значение этой переменной.

##### Зависимость типа «Запись после записи»

Возникает, если оператор с номером i формирует значение некоторой переменной (записывает ее), и оператор с номером j (j>i) записывает значение этой же переменной ((j>i)). Значение переменной между операторами i, j не используется.

##### Зависимость типа «Запись после чтения»

Возникает, если оператор с номером i читает значение некоторой переменной, и это значение меняется, оператором j (j>i). Последние 2 зависимости называются антизависимостями, их легко избежать, если для предыдущего и последующего значений переменной использовать разные переменные.

#### **Способы синхронизации**

Барьеры (Barrier). Обычно используются для всех параллельных задач. Устанавливаются, если необходимо обеспечить продолжение кода в данной точке после завершения всех параллельных участков кода — например, всех итераций цикла, которые выполняются параллельно.

Блокировки (Семафоры). Используются для защиты общих данных (критические секции). Первая задача устанавливает «замок». Остальные задачи ждут, пока собственник «замка» не откроет его.

Если требовались операции коммуникации, то может потребоваться синхронизация для этих операций. Необходимо обеспечить, чтобы до завершения передачи данных, они не начали обрабатываться. После завершения обработки порции данных результат должен быть передан приемнику до того, как начнется обработка новой порции данных.

#### **Балансировка**

Под балансировкой понимаем равномерность загрузки процессоров при выполнении задач.

Понятно, что если параллельно выполняются итерации цикла и необходимо ждать завершения цикла, то время выполнения этого цикла будет равно времени выполнения самой длинной итерации.

Для выполнения балансировки необходимо знать время выполнения каждой итерации и затем распределить эти итерации таким образом, чтобы каждый процессор был загружен примерно равномерно. Если время выполнения итерации изменяется по предсказуемому закону, это можно сделать статически, т.е. указать распределение нагрузки на этапе компиляции программы. Если время изменяется по случайному закону, например, зависит от данных, которые рассчитываются, распределение нагрузки надо выполнять динамически, т.е. во время выполнения программы.

При распределении нагрузки не следует привязываться к конкретному числу ядер на этапе трансляции программы. Следует определять это число в процессе выполнения программы, это позволит создать масштабируемую программу.

#### **Гранулярность**

При выделении параллельных участков следует определить, что выполнять параллельно. Например, можно параллельно выполнять каждую команду, т.е. фактически потоковая функция состоит из одной команды. Но тогда накладные расходы, связанные с созданием и уничтожением параллельных ветвей, будут добавляться к каждой команде, и программа будет выполняться медленнее, а не быстрее последовательной. Особенно это важно в случае использования систем с распределенной памятью, так как время обмена сообщениями обычно намного больше времени выполнения отдельных команд.

Поэтому следует определить минимальный блок, который имеет смысл выполнять параллельно. Для этого после определения предполагаемых параллельных участков следует провести вычислительный эксперимент по определению времени выполнения этого участка в последовательном () и параллельном () режиме.

Минимальный размер блока определяется значением, при котором и не менее ожидаемого ускорения.

#### **Учет операций ввода-вывода**

Операции ввода-вывода могут существенно замедлить выполнение итерации. Асинхронный ввод-вывод является платформенно-зависимым, и использовать его для программ, которые должны работать на разных платформах, не рекомендуется.

Необходимо помнить, что операции записи в один и тот же файл разными параллельными ветвями должны выполняться в эксклюзивном режиме, операции чтения могут выполняться одновременно.

Если ввод-вывод данных выполняется с удаленных ЭВМ, то нельзя прогнозировать время, необходимое для доступа к таким данным.

Есть параллельные файловые системы, которые на уровне файловой системы поддерживают параллельный файловый доступ. Примеры таких систем: PVFS/PVFS2, Linux; GPFS: General Parallel File System для AIX (IBM). Но, к сожалению, ни одна из файловых систем, которые используются с Windows, не является параллельной.

Исходя из рассмотренного, рекомендуется операции ввода-вывода выполнять в последовательном режиме. Если это не получается, то каждая ветвь пусть использует свой файл, а потом результаты работы можно объединить.

#### **Определение необходимых ресурсов**

Использование параллельных вычислений обычно приводит к увеличению необходимых ресурсов и особенно памяти. Так, вычисление суммы последовательным методом не требует дополнительной памяти. Если для вычисления используется метод последовательного деления пополам, то необходимо n/2 ячеек для хранения промежуточных значений сумм. Поэтому необходимо оценить объем требуемых данных, возможность их размещения в оперативной памяти, так как хранение промежуточных данных на диске может свести к 0 все преимущества параллельных вычислений.

Среди ресурсов необходимо выделить ресурсы общего доступа (память, файлы), определить режимы доступа к этим данным и необходимые объекты синхронизации.

#### **Масштабируемость**

Необходимо исследовать масштабируемость, т.е. поведение программы при увеличении числа параллельных ветвей за счет увеличения числа процессоров. Для обеспечения масштабируемости необходимо динамически определять необходимое число потоков и между этими потоками равномерно распределять нагрузку.

### **Реализация программы и анализ ее производительности**

В процессе реализации программы получают более точные значения для соотношения параллельного и последовательного кодов, далее уточняются все параметры, полученные при проектировании. Если случится, что ускорение и стоимость не удовлетворяют требованиям к программе, то заново выполняется пункт Декомпозиция. Такое спиральное выполнение всех этапов может выполняться многократно.

Для анализа производительности программы следует использовать встроенные средства анализа производительности программной среды, которая используется для разработки параллельной программы.

Необходимо обязательно использовать средства проверки правильности программы с точки зрения обработки тупиков и критических секций.

**Парадигмы параллельных приложений (**[**ссылка1**](https://pro-prof.com/forums/topic/parallel-programming-paradigms)**,** [**ссылка2**](https://studfile.net/preview/16404441/page:6/)**)**

Существует ряд схем взаимодействия самостоятельных частей параллельного приложения. Кратко опишем основные.

***Итеративный параллелизм***используется для реализации параллелизма в итеративной программе (чаще всего в циклах). Такой параллелизм характерен для распараллеливания по данным в согласованных параллельных вычислениях.

***Рекурсивный параллелизм***используется в программах с одной или несколькими рекурсивными процедурами, вызов которых независим. Каждый рекурсивный вызов порождает один или несколько новых процессов, которые независимо работают над решением задачи. В рамках такой парадигмы часто реализуются технологии «разделяй и властвуй» или «перебор с возвращением».

Итеративный и рекурсивный параллелизм основан на приемах, известных в последовательном программировании. Следующие схемы взаимодействия характерны именно для параллельных программ.

**«*Производители и потребители*»** – модель взаимодействия неравноправных процессов по поводу общих данных. Одни процессы «производят» данные, другие – их «потребляют». Часто такие процессы организуются в *конвейер*, через который проходит информация. Каждый процесс конвейера потребляет выход своего предшественника и производит входные данные для своего последователя. Другой распространенный способ организации потоков – древовидная структура, на ней основан, в частности,

принцип *дихотомии*.

**«*Клиенты и серверы*»** – наиболее распространенная модель взаимодействия процессов в распределенных системах. Клиентский процесс запрашивает (возможно, неоднократно) данные у сервера и ожидает ответа, затем использует полученные данные по своему усмотрению. Серверный процесс ожидает запроса от клиента, далее в соответствии с поступившим запросом обрабатывает данные и возвращает запросившему их процессу клиенту. Таким образом, в отличие от предыдущего случая, между клиентом и сервером необходимо установить двустороннюю связь. Сервер может быть реализован как одиночный процесс, обслуживающий одновременно несколько клиентских процессов. Сервер может быть многопоточной программой, каждый поток которой обслуживает своего клиента. Если клиент и сервер выполняются на одном компьютере, то они представляют собой параллельное программное обобщение процедур: сервер исполняет роль процедуры, а клиент ее вызывает. Однако если коды клиента и сервера разнесены в пространстве, то для синхронизации используются специальные технологии, такие как удаленный вызов процедур или рандеву.

**«*Управляющий и рабочие*»** – модель организации вычислений, при которой существует поток, координирующий работу всех остальных потоков. Как правило, управляющий поток распределяет данные, собирает и анализирует результаты. Эта парадигма часто применяется в задачах оптимизации и статистической обработки информации, при обработке изображений и других научных вычислениях с итеративными алгоритмами.

**«*Взаимодействующие равные*»** – модель, в которой исключен не занимающийся непосредственными вычислениями управляющий поток. Распределение работ в таком приложении либо фиксировано заранее, либо динамически определяется во время выполнения. Одним из распространенных способов динамического распределения работ при создании программ для ВС с общей памятью является *портфель задач*. Портфель задач, как правило, реализуется с помощью разделяемой переменной, доступ к которой в один момент времени имеет только один процесс. Если же память ВС распределенная, то такая схема распределения работ превращается в схему «управляющий – рабочий», поскольку портфель задач оформляется отдельным процессом. Основными примерами в этой области являются научные вычисления с итеративными алгоритмами и системы, требующие децентрализованного принятия решений.