Методичка по курсу «Распределенные вычисления»

Ф.С. Пеплин

30 сентября 2024 г.

Оглавление

Глава 1

Лекции

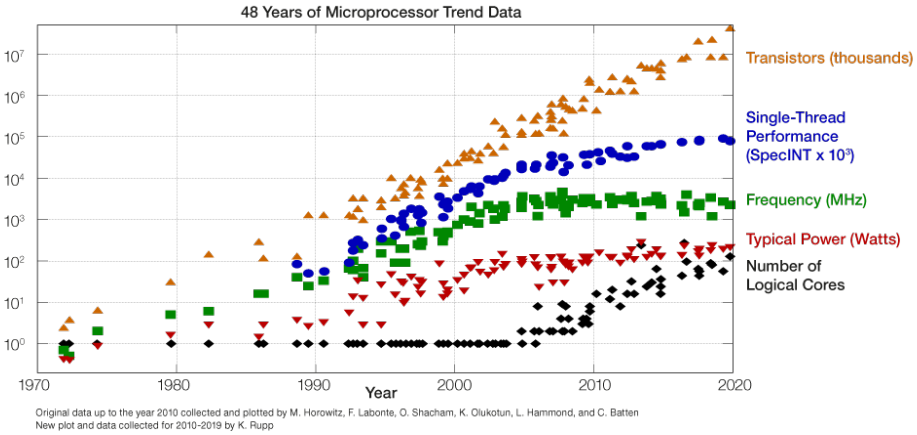

Рис. 1.1: Иллюстрация закона Мура

1.1 Введение в курс. Классификация параллельных систем

Формула оценивания

$$G = 0.5 \times \text{Exam} + 0.5 \times \text{Labs}$$

Закон Мура (1965): каждые два года количество транзисторов н интегральных схемах удваивается. Из рис. 1.1 видно, что общее количество транзисторов действительно увеличивается экспоненциально, однако производительность одноядерных процессоров, которая росла на 50% в год в период 1986 — 2003, в настоящее время стагнирует. Таким образом, рост производительности осуществляется за счет размещения нескольких самостоятельных ядер на одном процессоре.

Почему нельзя производить более мощные одноядерные системы?

- $\uparrow$ скорость $\Rightarrow \uparrow$ мощность $\Rightarrow \uparrow\ ^\circ C$ (при выполнении закона Мура в 2010 году тепловыделение было бы как в атомном реакторе)
- Скорость доступа к памяти не может быть выше скорости света $\Rightarrow$ при частоте 1 ТГц и памяти 8 Тб на 1 байт будет приходиться площадь атома

Решения проблемы

1. Квантовые вычисления
2. Развитие алгоритмов

3. Использовать многоядерные процессоры

Сферы применения HPC

1. Биотехнологии, биомедицина
   - Фолдинг белка
   - Персонализированная медицина, моделирование действия препаратов
2. Анализ данных
3. Вычислительное материаловедение
4. Расчеты в промышленности
   - Прочность, жесткость, устойчивость
   - Гидроаэродинамика, газовая динамика
   - Тепломассоперенос
5. Моделирование климата
6. Системы реального времени

Классификация параллельных систем

1. По используемой памяти
   - Общая память (Shared memory, рис. 1.2)
   - Распределенная память (Distributed memory, рис. 1.3)
2. По потокам данных и инструкций (таксономия Флинна)
   - Single Instruction — Multiple Data (SIMD) (параллелизм данных): общее устройство управления, разные операционные автоматы
   - Multiple Instruction — Multiple Data (MIMD) (параллелизм задач): разные устройства управления, разные операционные автоматы

Программы для систем с общей памятью называются многопоточными: в них один процесс распадается на несколько потоков. Программы для систем с распределенной памятью состоят из нескольких процессов.

Процесс

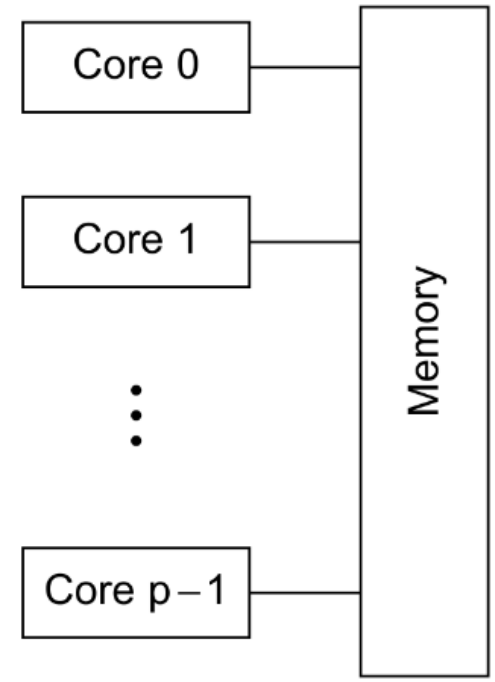

Рис. 1.2: Общая память

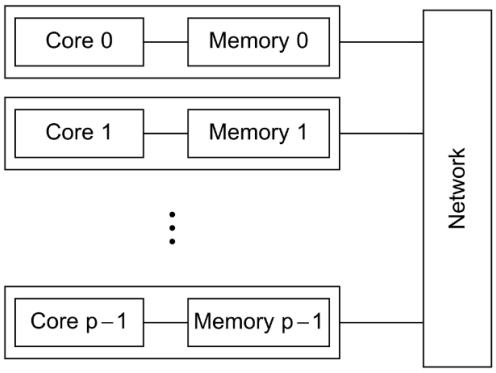

Рис. 1.3: Распределенная память

- Исполняемый код
- Дескрипторы ресурсов
- Стэк вызовов и куча
- Информация о статусе процесса (запущен, ожидает)
- Информация о правах доступа

Поток

- Тот же исполняемый файл, что и у процесса
- Те же ресурсы, кроме счетчика команд и стека вызовов

Одновременные вычисления (Concurrent computing) Различные задачи могут быть запущены одновременно.

Параллельные вычисления (Parallel computing) Различные процессы работают одновременно на разных вычислительных устройствах и взаимодействуют друг с другом для решения задачи.

Распределенные вычисления (Distributed computing) Программы взаимодействуют с другими программами для решения задачи.

Таким образом, параллельные и распределенные вычисления являются частными случаями одновременных вычислений. К одновременным вычислениям также можно отнести работу многозадачной ОС, даже на одноядерной системе.

1.2 Архитектура компьютера. Параллельное железо

1.2.1 Архитектура фон Неймана

В рамках данной модели компьютер представляется как центральный процессор, соединенный шиной с оперативной памятью.

В памяти единообразно хранятся данные и команды.

Центральный процессор состоит из двух частей: устройства управления (control unit) и операционного автомата (datapath). Устройство управления отвечает за определение инструкций, которые нужно выполнять. Операционный автомат занимается выполнением инструкций.

Процессор содержит регистры, в которых хранятся данные, с которыми в данный момент работает процессор. Один из регистров — счетчик команд, который содержит адрес следующей выполняемой процессором инструкции.

von Newmann bottleneck: Процессор выполняет инструкции в $\approx$ 100 раз быстрее, чем может получить доступ к памяти.

1.2.2 Улучшения архитектуры фон Неймана

Кэш-память

Кэш-строка. Объем данных, копируемый в кэш из оперативной памяти за одну операцию чтения. Конкретное содержание кэш-строки, формирующейся при доступе к ячейке памяти, определяется принципом пространственной и временной локальности. Размер $\approx$ 64 байт.

Попадание в кэш (cache hit). Требуемые процессору инструкции или данные находятся в кэше.

Кэш-промах (cache miss). Требуемые процессору инструкции или данные отсутствуют в кэше.

Неконсистентность кэша (cache inconsistency). Состояние, когда данные в основной памяти и в кэше отличаются.

Механизмы обеспечения когерентности кэшей:
Write-through — сквозная запись.
Write-back — обратная запись.

Ассоциативность кэшей. При необходимости записать новую кэш-строку в уже заполненный кэш, возникает вопрос: какую строку кэша удалить?

Кэш прямого отображения (direct mapped cache). Кэш-строка помещается в единственно возможное место в кэше (например, если кэш

состоит из четырех строк, то можно брать остаток от деления номера кэш-строки на 4 и записывать в соответствующее место в кэше).

Полностью ассоциативный кэш (fully associative cache). Кэш-строка может быть помещена в любое место в кэш-памяти.

n-ассоциативный кэш. Кэш-строка может быть помещена в одну из n локаций.

Если кэш полностью ассоциативный или n-ассоциативный, то для решения вопроса о том, в какое место записать новую строку, используется принцип `LRU (Least Recently Used)`.

Виртуальная память

Единое адресное пространство для оперативной памяти и файла подкачки (свопа).

В целом та же идея, что и с кэшем, но есть два отличия:

1. Используется только write-through подход (по причине медленного доступа к жесткому диску)
2. Виртуальная память управляется комбинацией железа и ОС (кэш управляется только аппаратно).

Аналог кэш-строки — страница виртуальной памяти размером 4 — 16 Кб.

Чтобы обеспечить работу многозадачной операционной системы и исключить обращение нескольких программ по одному физическому адресу, при компиляции программы оперируют виртуальными адресами, которые преобразуются в физические средствами ОС.

Таблица страниц. Структура данных, используемая для преобразования виртуального адреса в физический.

TLB (Translation Lookaside Buffer). Быстрая память, содержащая 16 — 512 строк таблицы страниц. Строки выбираются исходя из принципа локальности.

TLB hit, TLB miss.

page fault — обращение к странице, которой нет в основной памяти (только на диске)

Параллелизм на уровне инструкций

ILP (Instruction Level Parallelism). Бывает двух типов.

Конвейерная обработка данных (pipelining) Выполнение команды разбивается на этапы, на каждом из которых используются отдельные функциональные устройства.

Множественная выдача команд (multiple issue) Одновременное выполнение команд различными функциональными устройствами.

Если функциональные устройства назначаются компилятором, то **статическая** м.в.к. Если функциональные устройства назначаются в процессе работы программы, то **динамическая** м.в.к. (**суперскалярность**).

М.в.к. осуществляется с помощью **спекуляции**.

1.2.3 SIMD

Одно управляющее устройство — много операционных автоматов.
Векторные процессоры

- Скорость и простота использования
- Рациональное использование данных
- Ограниченная масштабируемость
- Плохо приспособлены для работы с более сложными структурами данных
- Плохо приспособлены для работы с векторами малого размера

GPU (см. рис. 1.4)

- Большое количество функциональных устройств (≈ 128) на одном ядре.
- Не чистый SIMD/MIMD: на каждом SM несколько варпов, каждый из которых работает как SIMD.
- Комбинация общей и распределенной памяти.
- Неэффективны для небольших задач.

1.2.4 MIMD

Общая память

Все процессоры имеют приватный кэш первого уровня (L1 cache). Кэши второго и третьего уровней могут быть как приватными, так и общими.

MIMD-системы с общей памятью можно разделить на UMA (Uniform Memory Access) и NUMA (Non-Uniform Memory Access).

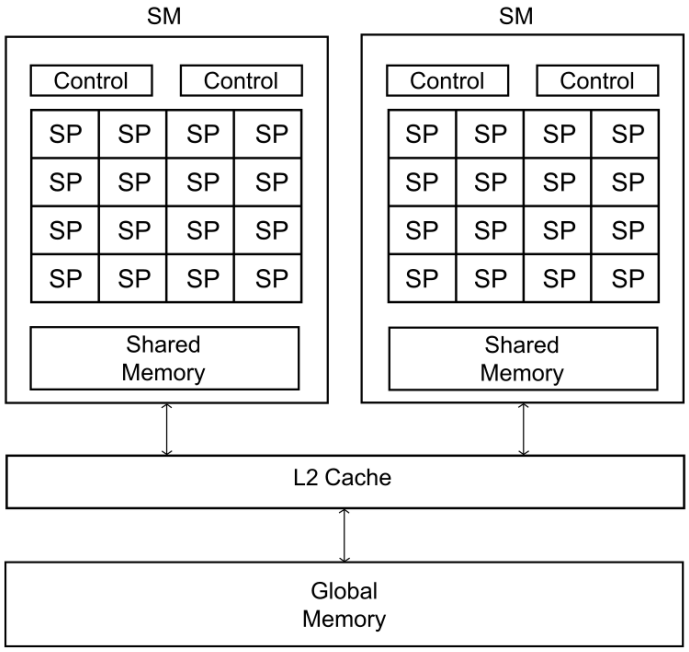

Рис. 1.4: Архитектура GPGPU

В случае UMA доступ ко всей памяти осуществляется напрямую: интерконнект соединяет все процессоры и все модули оперативной памяти. Поэтому время доступа к любой локации в памяти будет примерно одинаковым.

В случае NUMA интерконнект напрямую связывает процессор только со своей памятью. Доступ к «чужой» памяти осуществляется путем обращения к другому чипу, что занимает больше времени.

Распределенная память

MIMD-система с распределенной памятью — это кластер или грид-система.

Кластер. Состоит из отдельных узлов с общей памятью, соединенных сетью Ethernet.

Грид-система.

- Объединение сетей, находящихся друг от друга на большой расстоянии, в единую систему с распределенной памятью.

- В общей случае система гетерогенная.

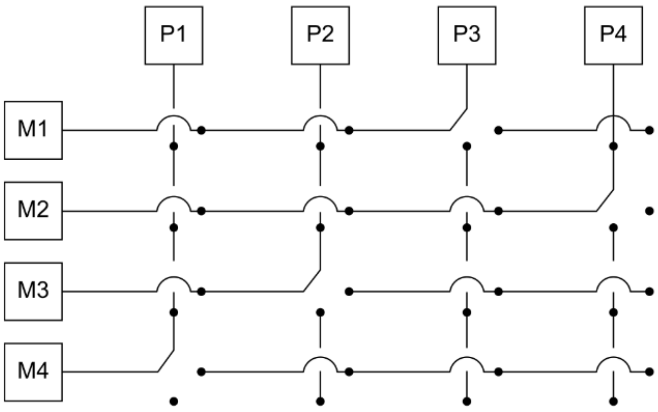

Рис. 1.5: Crossbar

1.2.5 Интерконнекты

Интерконнект — связь процессора с памятью.

Общая память

Для систем с общей памятью в качестве интерконнекта традиционно использовалась **шина** — параллельные провода, соединяющие устройства. Минус шины в том, что она не допускает подключение большого числа устройств. В настоящее время в системах с общей памятью используются переключатели-кроссбары (см. рис. 1.5), которые допускают одновременные коммуникации между сколь угодно большим количеством устройств. Однако стоимость такого интерконнекта слишком высока. В том числе по этой причине для большого количества процессоров чаще используется распределенная память.

Распределенная память

Ширина бисекции — два эквивалентных определения.

- Разбить систему на два множества с равным количеством узлов. Определить количество потоков данных, которые могут одновременно пересекать границу.
- Определить минимальное количество связей, которое нужно удалить, чтобы разбить систему на два несвязанных множества с одинаковым количеством узлов.

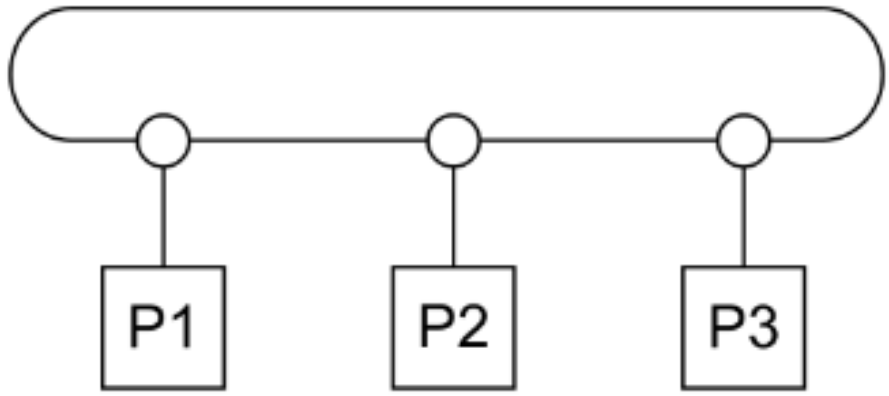

Рис. 1.6: Кольцо

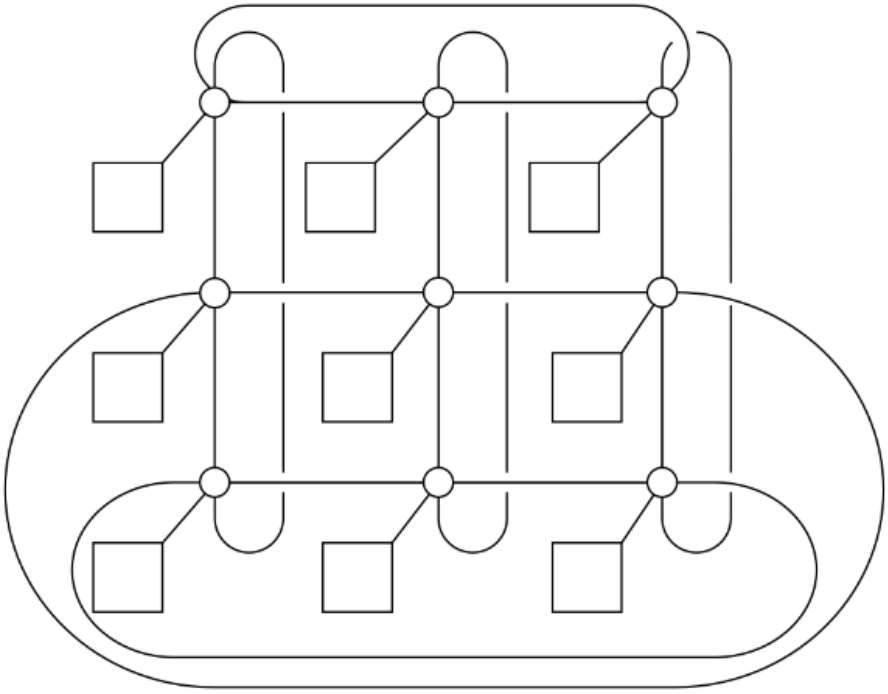

Рис. 1.7: Тороидальная сетка

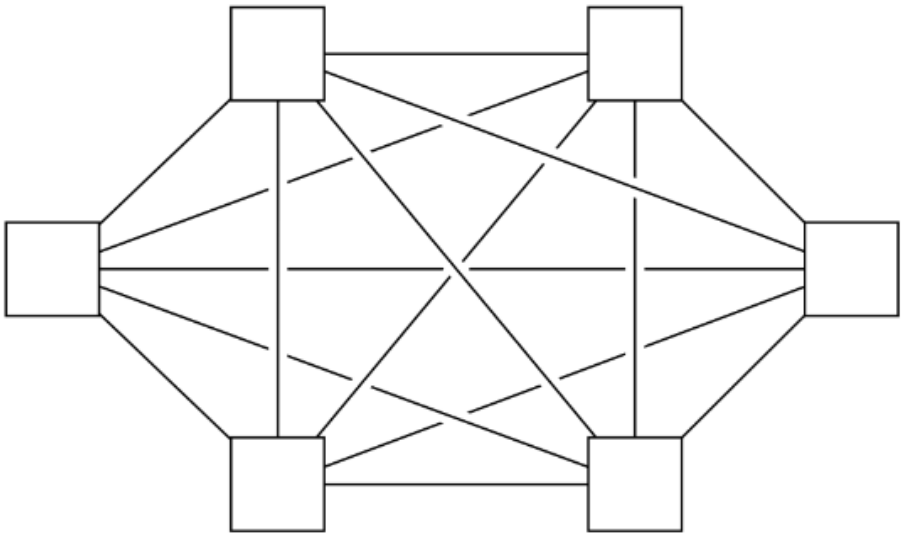

Рис. 1.8: Полностью связанный интерконнект

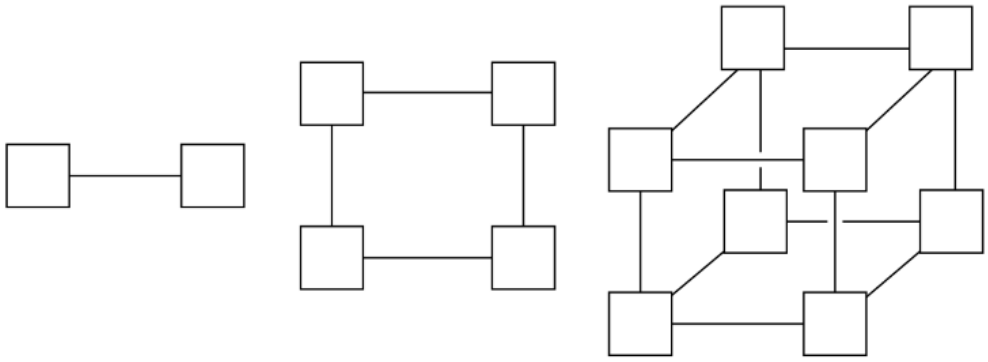

Рис. 1.9: Гиперкуб

Название	Ширина бисекции	Количество связей	Мощность адаптера
Кольцо	2	p	3
Тороидальная сетка	$2\sqrt{p}$	$2p$	5
Полносвязная сеть	$\frac{p^2}{4}$	$\frac{p^2}{2} - \frac{p}{2}$	p
Гиперкуб	$\frac{p}{2}$	$\frac{1}{2}p\log_2 p$	$1 + \log_2 p$

Таблица 1.1: Сравнение прямых интерконнектов

1.3 Оценка производительности параллельных программ. Получение и отправка сообщений в MPI

1.3.1 Оценка производительности параллельных программ

Ускорение параллельной программы S:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \tag{1.1}$$

где T_{serial} — время работы программы без распараллеливания, T_{parallel} — время работы параллельной программы.

Эффективность параллельной программы E:

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{pT_{\text{parallel}}}, \tag{1.2}$$

где p — количество потоков/процессов.

Ясно, что

$$E \cdot T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}. \tag{1.3}$$

Формула (1.3) подталкивает к интерпретации понятия эффективности как доли времени работы параллельной программы, которая тратится максимально эффективно. Иными словами, общее время работы программы можно представить как

$$T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}} \tag{1.4}$$

Источники возникновения T_{overhead}:

- Критические секции
- Синхронизация
- Коммуникация

Выведем оценку сверху для ускорения S:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \overset{(1.4)}{=} \frac{T_{\text{serial}}}{\frac{T_{\text{serial}}}{p} + T_{\text{overhead}}} < \frac{T_{\text{serial}}}{T_{\text{overhead}}} \tag{1.5}$$

Обозначим через r долю инструкций программы, которые не могут быть распараллелены. Предположим, что остальные $1 - r$ инструкций допускают распараллеливание без накладных расходов. Тогда $T_{\text{overhead}} = rT_{\text{serial}}$ и формула (1.5) примет вид

$$S < \frac{1}{r} \tag{1.6}$$

Формула (1.6) выражает собой **закон Амдала** (1967): ускорение выполнения программы за счет распараллеливания ее инструкций на множестве вычислителей ограничено временем, необходимым для выполнения ее последовательных инструкций.

Отметим, что при выводе закона Амдала мы зафиксировали количество инструкций и задались вопросом: «во сколько раз быстрее мы можем выполнить данное множество команд»? Полученная оценка является довольно пессимистичной. Другую, более радостную, зависимость, можно получить, если зафиксировать время выполнения последовательной программы и задать себе вопрос: «во сколько раз больше инструкций мы можем выполнить за то же самое время?». Чтобы ответить на этот вопрос, введем понятие *ускорения масштабирования* параллельной программы S_p:

$$S_p = (1 - r)p + r \tag{1.7}$$

Величина S_p может трактоваться как отношение количества инструкций, выполненных за единицу времени на параллельной архитектуре к количеству инструкций, выполненных за тот же промежуток времени программой без распараллеливания.

Из (1.7) следует, что объем работы, выполненный параллельной программой, почти в p раз больше, чем последовательной (если r мало). Объем инструкций, которые мы успеем выполнить на параллельной архитектуре, растет линейно при росте количества ядер. Данное утверждение является формулировкой **закона Густавсона** (1988).

1.3.2 Отправка и получение сообщений в системах с распределенной памятью

Листинг 1.1: Первая `MPI`-программа

```
1 #include <mpi.h>
2 #include <string.h>
3 #include <stdio.h>
4 #define MAX_SIZE 100
```

```
5
6  int main()
7  {
8
9      int comm_sz;
10     int my_rank;
11     char greeting[MAX_SIZE];
12
13     MPI_Init(NULL, NULL);
14
15     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
16     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
17
18     if (my_rank != 0)
19     {
20         sprintf(greeting,
21             "Greetings from process %d of %d", my_rank, comm_sz);
22         MPI_Send(greeting, strlen(greeting) + 1,
23             MPI_CHAR, 0, 0, MPI_COMM_WORLD);
24     }
25     else
26     {
27         printf("Greetings from process %d of %d\n",
28             my_rank, comm_sz);
29         for (int i = 1; i < comm_sz; i++)
30         {
31             MPI_Recv(greeting, MAX_SIZE, MPI_CHAR,
32                 i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
33             printf("%s\n", greeting);
34         }
35     }
36
37     MPI_Finalize();
38
39     return 0;
40 }
```

Все команды `MPI` должны помещаться между вызовами функций `MPI_Init` и `MPI_Finalize`. Первая из них принимает указатели на аргументы командной строки и выделяет место для буферов передаваемых и получаемых сообщений, решает, какой ранг будет у какого процесса и т.д.

Вторая функция освобождает выделенные ресурсы.

Коммуникатором называется множество процессов. Процессы могут отправлять и получать сообщения только в рамках одного коммуникатора. Коммуникатор `MPI_COMM_WORLD` состоит из всех процессов, которые запущены пользователем. В строках 15 и 16 происходит запись в переменные `comm_sz` и `my_rank` размера коммуникатора и порядкового номера текущего процесса в коммуникаторе соответственно.

Отправка и получение сообщений происходит путем вызова функций `MPI_Send` и `MPI_Recv`.

Сигнатура `MPI_Send`:

```
MPI_Send(void* buf,
         int size,
         MPI_Datatype,
         int dest,
         int tag,
         MPI_Comm communicator);
```

Первые три аргумента описывают передаваемое сообщение: где оно хранится, сколько элементов какого типа содержит. Четвертый аргумент — это номер процесса-получателя, пятый — тег, то есть условное обозначение, которое позволит получателю понять, какая информация содержится в этом сообщении. Последний аргумент — это имя коммуникатора.

Сигнатура `MPI_Recv`:

```
MPI_Recv(void* buf,
         int size,
         MPI_Datatype,
         int source,
         int tag,
         MPI_Comm communicator,
         MPI_Status* status_p);
```

Первые три аргумента характеризуют получаемое сообщение: в какой буфер его записать, каковы максимальный размер и тип элементов сообщения. Далее указывается ранг процесса-отправителя и тег сообщения, а также имя коммуникатора. Последний аргумент предназначен для вывода информации о ранге отправителя и теге сообщения. Зачем это нужно? Дело в том, что в качестве третьего аргумента функции `MPI_Recv` можно передать макрос `MPI_ANY_SOURCE`, а в качестве четвертого — `MPI_ANY_TAG`. Если указан хотя бы один из этих макросов, то мы можем не знать тега сообщения или его отправителя, или и того, и другого. Тогда нужно создать переменную типа `MPI_Status` и пере-

дать указатель на нее в функцию `MPI_Recv`. Функция `MPI_Recv` запишет в эту переменную информацию о теге и отправителе, которую можно получить, обратившись к полям `MPI_SOURCE` и `MPI_TAG`.

Что происходит, когда вызывается функция `MPI_Send`? В первую очередь передаваемая информация упаковывается в «конверт», который помимо содержательной части включает в себя служебные данные. Затем этот «конверт» может либо начать передаваться по сети получателю, когда тот вызовет соответствующую функцию `MPI_Recv`, либо происходит кэширование сообщения (сохранение во внутренней памяти) без его отправки. Обычно первый вариант развития событий происходит, когда размер передаваемого значения выше некоторого порогового значения, а второй вариант характерен для маленьких сообщений. Таким образом, когда программа возвращается из функции `MPI_Send`, мы не знаем точно, состоялась ли коммуникация с процессом-получателем.

Что же касается функции `MPI_Recv`, то возвращение из нее, напротив, говорит о том, что коммуникация состоялась.

Пусть процесс q вызывает функцию

```
MPI_Send(send_buf_p, send_buf_sz,
    send_type, dest, send_tag, send_comm);
```

Процесс r осуществляет вызов

```
MPI_Recv(recv_buf_p, recv_buf_sz,
    recv_type, source, recv_tag, recv_comm, &status);
```

Для того, чтобы случился акт коммуникации между процессами, нужно, чтобы

- `recv_comm=send_comm`
- `recv_tag=send_tag`
- `dest=r, source=q`

Для того, чтобы сообщение было передано успешно, нужно, чтобы дополнительно выполнялись еще условия

- `recv_type=send_type`
- `recv_buff_sz` $\geq$ `send_buff_sz`

1.4 Коллективные коммуникации в MPI-программах

Рассмотрим задачу нахождения значения определенного интеграла

$$\int_a^b f(x)dx \tag{1.8}$$

с использованием численных методов.

Согласно методу трапеций

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f(a)+f(b)\right)+h\left(f(x_1)+f(x_2)+\ldots f(x_{n-1})\right), \tag{1.9}$$

где n — количество отрезков, на которые делится отрезок $[a,b]$, $x_0 = a, x_1 = x_0 + h, x_2 = x_0 + 2h, \ldots, x_n = b$ — узлы разбиения, $h = \frac{b-a}{n}$ — шаг разбиения.

Первая попытка решить задачу приведена в листинге 1.2.

Листинг 1.2: Вычисление определенного интеграла

```
1  #include <mpi.h>
2  #include <string.h>
3  #include <stdio.h>
4
5  int n;
6  double a;
7  double b;
8
9  int comm_sz;
10 int my_rank;
11
12 double f(double x)
13 {
14   return x * x;
15 }
16
17 void Input()
18 {
19   if (my_rank == 0)
20   {
21     printf("Enter a, b, n\n");
22     scanf("%lf %lf %d", &a, &b, &n);
23     for (int i = 1; i < comm_sz; i++)
24     {
```

```
25          MPI_Send(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
26          MPI_Send(&b, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
27          MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
28        }
29      }
30      else
31      {
32        MPI_Recv(&a, 1, MPI_DOUBLE, 0, 0,
33          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
34        MPI_Recv(&b, 1, MPI_DOUBLE, 0, 0,
35          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36        MPI_Recv(&n, 1, MPI_INT, 0, 0,
37          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
38      }
39  }
40
41  double Trap(double a, double b, int n, double h)
42  {
43    double res = 0.0;
44    res = 0.5 * (f(a) + f(b));
45    for (int i = 1; i <= n - 1; i++)
46    {
47      double x = a + i * h;
48      res += f(x);
49    }
50    return res * h;
51  }
52
53  int main()
54  {
55
56    MPI_Init(NULL, NULL);
57
58    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
59    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
60
61    Input();
62
63    double h = (b - a) / n;
64
65    int local_n = n / comm_sz;
```

```
66    double local_a = a + my_rank * local_n * h;
67    double local_b = local_a + local_n * h;
68    double local_int = Trap(local_a, local_b, local_n, h);
69
70    double global_int;
71
72    if (my_rank != 0)
73    {
74      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
75    }
76    else
77    {
78      global_int = local_int;
79      for (int i = 1; i < comm_sz; i++)
80      {
81        MPI_Recv(&local_int, 1, MPI_DOUBLE, i, 0,
82          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
83        global_int += local_int;
84      }
85    }
86
87    if (my_rank == 0)
88    printf("Int = %lf\n", global_int);
89
90    MPI_Finalize();
91
92    return 0;
93  }
```

Основной недостаток программы 1.2 заключается в том, что нахождение глобальной суммы из локальных сумм целиком и полностью выполняется нулевым процессом. Выглядит более разумным использовать древовидную структуру (см. рис. 1.10). Это позволит снизить время с $O(p)$ до $O(\log(p))$ (p — количество процессов). Реализовать этот алгоритм можно с помощью `MPI_Reduce`:

```
MPI_Reduce(
      void*          input_buf_p,      /*in*/
      void*          output_buf_p,     /*out*/
      int            count,            /*in*/
      MPI_Datatype   datatype,         /*in*/
      MPI_Op         operation,        /*in*/
```

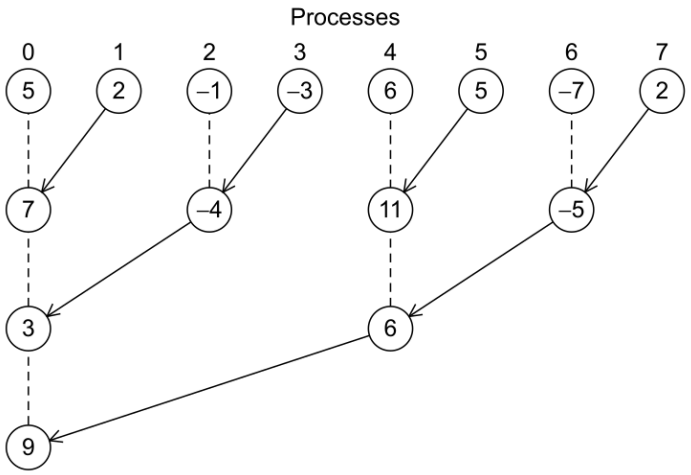

Рис. 1.10: Нахождение глобальной суммы с помощью `MPI_Reduce`

```
    int             dest,           /*in*/
    MPI_Comm        Communicator    /*in*/
);
```

Функция `MPI_Reduce` реализует **коллективную коммуникацию**. Эта функция берет из каждого процесса значения, на которые указывают `input_buf_p`, и суммирует их, так, чтобы результат оказался в буфере `output_buf_p` процесса `root`. Помимо суммирования, можно выполнять и другие операции, если передать соответствующие аргумент типа `MPI_Op`. Возможно вычисление произведения, нахождение минимума или максимума, логические и побитовые операции. Также можно определить свою собственную операцию.

Примеры других функций, реализующих коллективную коммуникацию

```
MPI_Allreduce(
    void*           input_buf_p,    /*in*/
    void*           output_buf_p,   /*out*/
    int             count,          /*in*/
    MPI_Datatype    datatype,       /*in*/
    MPI_Op          operation,      /*in*/
    MPI_Comm        Communicator    /*in*/
);
```

Работает также, как и `MPI_Reduce`, однако полученное значение затем передается всех процессам коммуникатора (см. рис. 1.11).

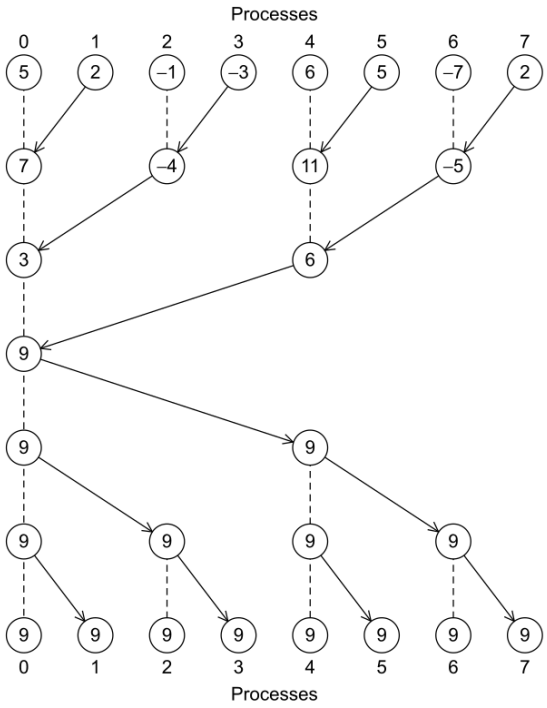

Рис. 1.11: Механизм работы `MPI_Allreduce`

```
MPI_Bcast(
    void*           input_buf_p,    /*in/out*/
    int             count,          /*in*/
    MPI_Datatype    datatype,       /*in*/
    int             source_proc,    /*in*/
    MPI_Comm        Communicator    /*in*/
);
```

Данная функция берет значение, которое хранится в буфере `input_buf_p` на процессе `source_proc`, и отправляет его всем остальным процессам в коммуникаторе, записывая результат в переменную `input_buf_p`.

Работа функции `MPI_Allreduce` сводится к последовательному вызову `MPI_Reduce` и `MPI_Bcast`.

Перепишем программу из листинга 1.2, используя возможности коллективных коммуникаций. Результат см. в листинге 1.3

Листинг 1.3: Демонстрация силы коллективных коммуникаций

```
1 #include <mpi.h>
2 #include <string.h>
3 #include <stdio.h>
```

```
4
5  int n;
6  double a;
7  double b;
8
9  int comm_sz;
10 int my_rank;
11
12 double f(double x)
13 {
14   return x * x;
15 }
16
17 void Input()
18 {
19   if (my_rank == 0)
20   {
21     printf("Enter a, b, n\n");
22     scanf("%lf %lf %d", &a, &b, &n);
23   }
24   MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
25   MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
26   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
27 }
28
29 double Trap(double a, double b, int n, double h)
30 {
31   double res = 0.0;
32   res = (f(a) + f(b)) / 2.0;
33   for (int i = 1; i <= n - 1; i++)
34   {
35     double x = a + i * h;
36     res += f(x);
37   }
38   return h * res;
39 }
40
41 int main()
42 {
43
44   MPI_Init(NULL, NULL);
```

```
45
46    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
47    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
48
49    Input();
50
51    int local_n = n / comm_sz;
52    double h = (b - a) / n;
53
54    double local_a = a + my_rank * local_n * h;
55    double local_b = local_a + local_n * h;
56    double local_int = Trap(local_a, local_b, local_n, h);
57    double global_int = 0.0;
58
59    MPI_Reduce(&local_int, &global_int, 1, MPI_DOUBLE,
60      MPI_SUM, 0, MPI_COMM_WORLD);
61
62    if (my_rank == 0)
63      printf("Int = %lf\n", global_int);
64
65    MPI_Finalize();
66
67    return 0;
68 }
```

Свойства функций, относящихся к классу коллективных коммуникаций.

1. Все процессы внутри коммуникатора должны вызывать эту функцию. Если хотя бы один процесс в коммуникаторе не вызовет функцию, то результат будет непредсказуемым.

2. Выходной буфер `output_buffer_p` используется только на процессе с рангом `root`. На других процессах можно передавать что угодно. Так, вызов в строке 59 листинга 1.3 можно заменить на

```
if(my_rank == 0)
  MPI_Reduce(&local_int, &global_int, 1, MPI_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);
else
  MPI_Reduce(&local_int, &global_int, 1, MPI_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);
```

3. Точечные коммуникации (`MPI_Send, MPI_Recv`) стыкуются нга основе совпадения коммуникатора и тега. Коллективные коммуникации не используют тегов, поэтому их стыковка происходит на основе коммуникатора и порядка, в котором вызываются функции. Так, если запустить код из листинга **??** на трех процессах, то значения b и d будут 4 и 5, не 3 и 6.

Листинг 1.4: Стыковка коллективных коммуникаций

```
int a = 1;
int c = 2;
int b = 0;
int d = 0;

if (my_rank == 0 || my_rank == 2)
{
  MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
else
{
  MPI_Reduce(&c, &d, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Reduce(&a, &b, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}

if (my_rank == 0)
printf("b = %d, d = %d\n", b, d);
```

1.5 Коллективные коммуникации (продолжение). Производные типы данных

1.6 Параллельная сортировка в MPI

1.7 OpenMP

1.8 OpenMP

1.9 OpenMP

1.10 CUDA

1.11 CUDA

1.12 CUDA

Глава 2

Семинары

2.1 Pthreads: создание и завершение потоков

Pthreads (POSIX Threads) — стандарт POSIX-реализации потоков. Реализации данного API существуют для большого числа UNIX-подобных ОС (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X).

Для использования в ОС Windows рекомендуется установить WSL (Windows Subsystem for Linux) — тонкий слой виртуализации, который позволяет запускать приложения Linux, не покидая привычного окружения Windows. Для установки WSL нужно выполнить следующую команду в PowerShell (запуск от имени администратора):

```
wsl --install
```

После завершения установки запустите программу Ubuntu — это и есть терминал Linux.

Логика управления потоками в pthreads в целом соответствует таковой при работе с библиотекой thread в C++. Отметим, что реализация потоков в C++ в Unix-системах существенно опирается на вызовы функций из pthreads. Библиотека pthreads предоставляет более низкоуровневый доступ к управлению процессом выполнения потоков, что позволяет добиться более высокой производительности. Поэтому данная библиотека широко используется для создания систем реального времени, высокочастотной торговли (HFT) и пр.

Напишем многопоточную версию «Hello, world!».

Листинг 2.1: Многопоточный «Hello, world»

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<pthread.h>
4
5  int thread_count;
6
7  void* routine(void* rank)
8  {
9      long my_rank = (long)rank;
10     printf("Hello from thread %ld of %d\n",
11         my_rank, thread_count);
12     return NULL;
13 }
14
15 int main(int argc, char** argv)
16 {
17     thread_count = strtol(argv[1], NULL, 10);
```

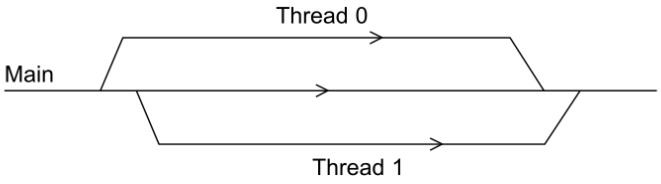

Рис. 2.1: Создание и завершение потоков

```
18      pthread_t* thread_handles =
19          malloc(thread_count * sizeof(pthread_t));
20
21      for(long i = 0; i < thread_count; i++){
22          int err = pthread_create(&thread_handles[i], NULL,
23              routine, (void*) i);
24          if(err != 0)
25                  perror("Error");
26      }
27
28      printf("Hello from main thread\n");
29
30      for(int i= 0; i < thread_count; i++)
31          pthread_join(thread_handles[i], NULL);
32
33      free(thread_handles);
34
35      return 0;
36  }
```

Создание потока.

```
pthread_create(
    pthread_t*            thread_p,             /*out*/
    const pthread_attr_t* attr_p,               /*in*/
    void*                 (*routine) (void*),   /*in*/
    void*                 args                  /*in*/
);
```

Первый аргумент — это дескриптор потока, куда функция pthread_create запишет данные конкретного создаваемого потока. Гарантируется, что в объекте типа `pthread_t` содержится достаточно информации, чтобы однозначно идентифицировать поток.

Дескрипторы потока можно использовать, например, чтобы понять, соответствуют ли два дескриптора одному или разным потокам

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Можно получить дескриптор текущего потока

```
pthread_t pthread_self(void);
```

Второй аргумент функции `pthread_create` — атрибуты потока. Эта структура содержит параметры планировщика, размер стека и другие свойства (подробности что и как задать см. тут).

Третий аргумент — это указатель на функцию, запускаемую на потоке.

Четвертый аргумент — параметры, передаваемые в функцию.

Завершение потока

```
int pthread_join(
    pthread_t           thread_p,           /*in*/
    void**              p__arg              /*out*/
);

int pthread_detach(
    pthread_t           thread_p,           /*in*/
);

int pthread_kill(
    pthread_t           thread_p,           /*in*/
    int                 sig                 /*in*/
);

int pthread_cancel(
    pthread_t           thread_p            /*in*/
);

void pthread_exit(
    void**              retval              /*out*/
);
```

Функция `pthread_join` ждет, когда все потоки завершат свою работу. Далее она помещает в `&p_arg` возвращаемое потоком значение и очищает выделенные потоку ресурсы. Таким образом, когда программы покидает строку 31 программы 2.1, то мы можем быть уверены в том, что все потоки завершили свою работу, ресурсы, занятые потоками, освобождены,

а возвращаемые значения успешно сохранены (если возникла ошибка в процессе, то функция вернет ее код).

Функция `pthread_detach`, напротив, ничего не будет ждать и возвращать из потока. Она просто пометит поток как «отсоединенный» и больше ничего не будет делать. Поток сам завершит свою работу и выделенные на него ресурсы будут возвращены операционной системе. Отличие `pthread_detach` от `pthread_join` можно проиллюстрировать, сравнив результат работы программ 2.1 и 2.2 при большом количестве потоков (например, 100000 или больше). Первая программа выдаст ошибку «Cannot allocate memory», т.к. все создаваемые потоки «живут» вплоть до 31 строчки, поэтому память рано или поздно исчерпается. Что же касается второй программы (листинг 2.2), то каждый вновь созданный поток функцией `detach` тут же «отсоединяется», и по завершению своей работы тут же освобождает ресурсы. Поэтому вторая программа будет работать при любом количестве потоков.

Листинг 2.2: Отличие join от detach

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

int thread_count;

void* routine(void* rank)
{
    long my_rank = (long)rank;
    printf("Hello from thread %ld of %d\n",
        my_rank, thread_count);
    return NULL;
}

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

    thread_count = strtol(argv[1], NULL, 10);

    pthread_t* thread_handles = malloc(thread_count *
        sizeof(pthread_t));

    for(long i =0; i<thread_count; i++){
        int err = pthread_create(&thread_handles[i], NULL,
```

```
            routine, (void*) i);
        pthread_detach(thread_handles[i]);
        if(err != 0)
            perror("Error");
    }

    printf("Hello from main thread\n");

    free(thread_handles);

    return 0;
}
```

Функция `pthread_kill` отправляет потоку сигнал, который передается ей в качестве второго аргумента. Константы, соответствующие передаваемым кодам, содержатся в заголовочном файле `signal.h`. Для завершения потока можно передавать константу `SIGQUIT`, при этом поток будет «убит» незамедлительно.

Функция `pthread_exit` завершает текущий поток (тот, из которого она была вызвана), возвращая из него значение `&retval`.

Функция `pthread_cancel` отправляет сигнал завершения потоку, дескриптор которого передается в качестве аргумента. При этом поток может как отреагировать на этот сигнал и завершиться (если он отменяемый), либо проигнорировать его (если он неотменяемый). По умолчанию процесс является отменяемым. Чтобы задать отменяемость процесса, используется функция

```
int pthread_setcancelstate(
    int    state,            /*in*/
    int    *oldstate         /*out*/
);
```

Первый аргумент — это константа, значение которой
`PTHREAD_CANCEL_ENABLE` или
`PTHREAD_CANCEL_DISABLE`.
Во втором аргументе можно сохранить текущий статус потока.

Если поток отменяемый, то посылаемый сигнал отмены будет исполнен либо асинхронно, либо отложено (по умолчанию). Асинхронное исполнение означает исполнение в любой момент (как правило, немедленно). Отложенное исполнение означает, что поток завершится, как только встретит одну из команд, являющихся точками отмены. Полный список таких команд см. здесь.

Конкретный тип отмены задается аналогично, с помощью функции

```
int pthread_setcanceltype(
    int     type,               /*in*/
    int     *oldtype            /*out*/
);
```

Первый аргумент — это одна из двух констант
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_ASYNCHRONOUS
Во втором аргументе сохраняется текущее значение.

Следующая программа (листинг 2.3) демонстрирует передачу структуры в качестве аргумента для потока, а также получение возвращаемого потоком значения.

Листинг 2.3: Передача и возвращение структур из потока

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>

int thread_count;

struct S
{
    long rank;
    char* title;
};

void *routine(void *s)
{
    struct S* my_s = (struct S*)s;
    printf("Hello from thread %ld of %d, Title: %s\n",
    my_s->rank, thread_count, my_s->title);
    struct S* ret = malloc(sizeof(struct S));
    ret->rank = my_s->rank;
    ret->title = "return";
    return ret;
}

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

```
{

    thread_count = strtol(argv[1], NULL, 10);

    pthread_t *thread_handles = malloc(thread_count * sizeof(pthread

    for (long i = 0; i < thread_count; i++)
    {
        struct S* s = malloc(sizeof(struct S));
        s->rank = i;
        s->title = "Title";
        int err = pthread_create(&thread_handles[i],
            NULL, routine, (void *)s);
        if (err != 0)
            perror("Error");
        }

    printf("Hello from main thread\n");

    for (int i = 0; i < thread_count; i++){
        struct S* s;
        pthread_join(thread_handles[i], (void**)&s);
        printf("Return: rank=%ld, title=%s\n", s->rank, s->title);
        free(s);
    }

    free(thread_handles);

    return 0;
}
```

Время	Поток 1	Поток 2
0	Взять операнды	—
0	Передвинуть точку	Взять операнды
0	Сложить мантиссы	Передвинуть точку
0	Округлить	Сложить мантиссы
0	Записать результат	Округлить
0	—	Записать результат

Таблица 2.1: Механизм возникновения гонки

2.2 Pthreads: мьютексы и семафоры

2.2.1 Состояние гонки. Критические секции

Разложим функцию arctg в ряд Тейлора в окрестности точки $x = 0$:

$$\operatorname{arctg} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \tag{2.1}$$

Подставляя в (2.1) $x = 1$ и учитывая, что $\operatorname{arctg} 1 = \frac{\pi}{4}$, получаем следующее выражение для вычисления числа π:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right) \tag{2.2}$$

В листинге 2.4 представлена неудачная попытка посчитать сумму (2.2). Результат работы программы 2.4 при запуске с несколькими потоками оказывается недетерминированным. Причина в состоянии гонки в строке 22.

Продемонстрируем механизм возникновения состояния гонки на примере. Предположим, что начальное значение переменной `sum` равно 10. Пусть работают два потока и первый увеличивает значение `sum` на 2, а второй уменьшает на 5. Тогда после выполнения строки 22 обоими потоками в переменной `sum` должно храниться число 7. Однако если первый поток начнет выполнять присваивание раньше, чем второй, а закончит после того, как второй начнет, но до того, как закончит, то второй поток «затрет» результат работы первого потока (см. таблицу 2.1. Тогда легко видеть, что в переменной `sum` в рассматриваемом случае будет лежать число 5.

Листинг 2.4: Неудачая попытка посчитать число π

```
1
2  double sum = 0;
```

```
3
4  void *routine(void *rank){
5
6      long my_rank = (long)rank;
7      double factor;
8      int my_n = n / thread_count;
9      int my_first_i = my_rank * my_n;
10     int my_last_i = (my_rank + 1) * my_n - 1;
11     if(my_first_i % 2 == 0)
12         factor = 1.0;
13     else
14         factor = -1.0;
15
16     for (int i = my_first_i; i < my_last_i; i++, factor *= -1)
17     {
18         sum += factor / (2 * i + 1);
19     }
20
21     return NULL;
22
23 }
```

Таким образом, строка 22 в листинге 2.4 является **критической секцией**. Так называется фрагмент программы, в случае одновременного выполнения которого несколькими потоками может возникнуть состояние гонки. Следовательно, необходим механизм, исключающий одновременное выполнение несколькими потоками критической секции. Далее рассмотрим два таких механизма: холостой цикл и мьютексы.

Использование холостого цикла показано в листинге 2.5.

Листинг 2.5: Холостой цикл

```
int flag = 0;
void *routine(void *rank)
{

    long my_rank = (long)rank;
    double factor;
    int my_n = n / thread_count;
    int my_first_i = my_rank * my_n;
    int my_last_i = (my_rank + 1) * my_n - 1;
    if (my_first_i % 2 == 0)
        factor = 1.0;
```

```
    else
        factor = -1.0;

    double my_sum = 0.0;

    for (int i = my_first_i; i < my_last_i; i++, factor *= -1)
    {
        my_sum += factor / (2 * i + 1);
    }
    while (flag != my_rank);
    sum += my_sum;
    flag++;

    return NULL;
}
```

Второй механизм исключения одновремменного выполнения критической секции — мьютекс — показан в листинге 2.6. Мьютекс — это примитив синхронизации, которому в `pthreads` соответствует тип данных `pthread_mutex_t`.

Мьютекс инициализируется с помощью функции

```
int pthread_mutex_init(
        pthread_mutex_t*        mutex,    /*out*/
        pthread_mutexattr_t*    attr_p    /*in*/
);
```

Второй аргумент — указатель на атрибуты мьютекса, которые позволяют задать его тип с помощью функции `pthread_mutexattr_settype`:

```
int pthread_mutexattr_settype(
        pthread_mutexattr_t     *attr,    /*out*/
        int                     type      /*in*/
);
```

Некоторые типы мьютексов (второй аргумент `pthread_mutexattr_settype`):
`PTHREAD_MUTEX_NORMAL`
`PTHREAD_MUTEX_ERRORCHECK`
`PTHREAD_MUTEX_RECURSIVE`

Две основные функции для работы с мьютексами:

```
int pthread_mutex_lock(
        pthread_mutex_t* mutex /*in/out*/
);
```

Время	Поток 1	Поток 2
0	`pthread_mutex_lock(&m1)`	`pthread_mutex_lock(&m2)`
1	`pthread_mutex_lock(&m2)`	`pthread_mutex_lock(&m1)`

Таблица 2.2: Дэдлок

```
int pthread_mutex_unlock(
        pthread_mutex_t* mutex /*in/out*/
);
```

Когда происходит выполнение функции `pthread_mutex_lock`, то переданный ей в качестве аргумента мьютекс будет помечен как «закрытый», а его владельцем будет вызвавший функцию поток. Если поток 1 закрыл мьютекс, а поток 2 попытается его еще раз закрыть, то поток 2 будет ожидать в функции `pthread_mutex_lock`, пока поток 1 не откроет его вызовом `pthread_mutex_unlock`. После этого поток 2 закроет мьютекс и продолжит выполнение кода, став новым бладателем мьютекса. Отметим, что**открыть мьютекс может только тот поток, который его закрыл** (который им владеет в данный момент). При попытке открыть мьютекс, закрытый другим потоком, поведение программы не определено.

Дэдлок (deadlock) возникает, когда потоки вынуждены ждать получения мьютекса бесконечно долго. Пример см. в таблице 2.2.

Освободить ресурсы, выделенные для мьютекса, можно с помощью функции

```
int pthread_mutex_destroy(
        pthread_mutex_t*        mutex     /*in, out*/
);
```

Листинг 2.6: Использование мьютекса

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int thread_count;

int n;
double sum = 0;

int flag = 0;
```

```
pthread_mutex_t mutex;

void *routine(void *rank)
{

    long my_rank = (long)rank;
    double factor;
    int my_n = n / thread_count;
    int my_first_i = my_rank * my_n;
    int my_last_i = (my_rank + 1) * my_n - 1;
    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    double my_sum = 0.0;

    for (int i = my_first_i; i < my_last_i; i++, factor *= -1)
    {
        my_sum += factor / (2 * i + 1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main(int argc, char **argv)
{
    thread_count = strtol(argv[1], NULL, 10);
    n = strtol(argv[2], NULL, 10);

    pthread_t *thread_handles =
        malloc(thread_count * sizeof(pthread_t));

    pthread_mutex_init(&mutex, NULL);

    for (long i = 0; i < thread_count; ++i)
        pthread_create(&thread_handles[i],
```

```
            NULL, routine, (void *)i);

    for (int i = 0; i < thread_count; ++i)
        pthread_join(thread_handles[i], NULL);

    printf("pi=%lf\n", 4 * sum);

    pthread_mutex_destroy(&mutex);
    free(thread_handles);

    return 0;
}
```

Отличия типов мьютексов друг от друга:

1. `PTHREAD_MUTEX_NORMAL`. Мьютекс по умолчанию. Из соображений скорости работы не осуществляется контроль ошибок. Так, попытка потока залочить уже закрытый им же мьютекс, равно как и попытка открыть закрытый кем-то другим (или открытый) мьютекс, приведет к неопределенному поведению.

2. `PTHREAD_MUTEX_ERRORCHECK`. Мьютекс со встроенной проверкой на ошибки. Если поток попытается закрыть мьютекс, который он же закрывал, то вернется ошибка. Также ошибка будет возвращена, если поток попытается открыть мьютекс, который он не закрывал (или который открыт).

3. `PTHREAD_MUTEX_RECURSIVE`. В отличие от предыдущих двух мьютексов, данный тип допускает повторное закрытие тем же потоком без ошибок или неопределенного поведения. При этом происходит подсчет закрытий и открытий.

Отличие первых двух типов мьютексов между собой демонстрирует листинг 2.7. Программа выведет сообщения `Operation not permitted`. Если в строке 35 заменить тип мьютекса на `PTHREAD_MUTEX_NORMAL`, то сообщений об ошибках не будет.

Листинг 2.7: Демонстрация отличий мьютекса с провркой ошибок от обычного

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
```

```
5  #include <errno.h>
6  #define MSG_SIZE 100
7
8  pthread_mutex_t m;
9
10 int thread_count;
11
12 char **messages;
13
14 void *routine(void *rank)
15 {
16
17     long my_rank = (long)rank;
18     errno = pthread_mutex_unlock(&m);
19     if (errno != 0)
20     perror("Error");
21
22     return NULL;
23 }
24
25 int main(int argc, char **argv)
26 {
27     thread_count = strtol(argv[1], NULL, 10);
28
29     messages = malloc(thread_count * sizeof(char *));
30
31     pthread_t *thread_handles =
32         malloc(thread_count * sizeof(pthread_t));
33
34     pthread_mutexattr_t attr;
35     pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
36
37     pthread_mutex_init(&m, &attr);
38     pthread_mutex_lock(&m);
39
40     for (long i = 0; i < thread_count; ++i)
41         pthread_create(&thread_handles[i], NULL, routine, (void *)i);
42
43     for (int i = 0; i < thread_count; ++i)
44         pthread_join(thread_handles[i], NULL);
45
```

```
46      free(thread_handles);
47      pthread_mutex_destroy(&m);
48
49      return 0;
50  }
```

2.2.2 Семафоры

Рассмотрим задачу синхронизации типа «производитель — потребитель», простейшая формулировка которой требует от потоков передавать сообщения друг другу по кругу. Первая попытка такой программы показана в листинге 2.8. Результат работы программы предсказуем: некоторые потоки попадут в 20 строку раньше, чем предшествующие потоки выполнят присваивание в 19 строке. Поэтому некоторые потоки не смогут вывести предназначенные для них сообщения.

Листинг 2.8: Первая попытка потоков оставлять сообщения друг другу

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  int thread_count;
6
7  #define MSG_SIZE 100
8
9  char**  messages;
10
11 void *routine(void *rank)
12 {
13
14      long my_rank = (long)rank;
15      int dest = (my_rank + 1) % thread_count;
16      int source = (my_rank + thread_count - 1) % thread_count;
17      char* mes = malloc(MSG_SIZE * sizeof(char));
18      sprintf(mes, "Hello from %ld to %d", my_rank, dest);
19      messages[dest] = mes;
20      if(messages[my_rank] != NULL)
21          printf("Thread %ld -> %s\n", my_rank, messages[my_rank]);
22      else
23          printf("Thread %ld -> No messages\n", my_rank);
24      return NULL;
```

```
25 }
26
27 int main(int argc, char **argv)
28 {
29     thread_count = strtol(argv[1], NULL, 10);
30
31     messages = malloc(thread_count * sizeof(char*));
32
33     pthread_t *thread_handles =
34         malloc(thread_count * sizeof(pthread_t));
35
36     for (long i = 0; i < thread_count; ++i)
37         pthread_create(&thread_handles[i], NULL,
38             routine, (void *)i);
39
40     for (int i = 0; i < thread_count; ++i)
41         pthread_join(thread_handles[i], NULL);
42
43     free(thread_handles);
44
45     for(int i=0; i<thread_count; i++)
46         free(messages[i]);
47     free(messages);
48
49     return 0;
50 }
```

Корректная реализация алгоритма возможна с использованием семафоров.

Семафор — это переменная типа `sem_t`, которая представляет собой беззнаковое целое число.

Инициализация семафора

```
int sem_init(
    sem_t*       s_p,     /*out*/
    int          shared,  /*int*/
    int          ini_v    /*int*/
);
```

Второй аргумент — 0, если семафор общий для всех потоков, или 1, если нет. Третий аргумент — начальное значение семафора.

Освобождение ресурсов семафора

```
int sem_init(
    sem_t*       s_p,      /*in/out*/
);
```

Две основные команды для работы с семафорами — это поднять семафор, то есть увеличить на единицу (`post`), или опустить, то есть уменьшить на единицу (`wait`).

```
int sem_init(
sem_t*       s_p,      /*in/out*/
);

int sem_wait(
sem_t*       s_p,      /*in/out*/
);
```

Если текущее значение семафора единица или выше, то вызов функции `wait` просто уменьшает его на единицу. Если значение семафора равно 0, то вызов функции `wait` приведет к блокировке потока. Поток не покинет функцию `wait`, пока другой поток не выполнит `post` для этого же семафора. Тогда значение семафора станет равный единице, функция `wait` в заблокированном потоке опустит семафор и поток продолжит выполнять дальнейшие инструкции.

Пример использования семафоров для решения вышепоставленной задачи см. в листинге 2.9.

Листинг 2.9: Использование симафоров

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MSG_SIZE 100

sem_t *sems;

int thread_count;

char **messages;

void *routine(void *rank)
{
```

```
    long my_rank = (long)rank;
    int dest = (my_rank + 1) % thread_count;
    int source = (my_rank + thread_count - 1) % thread_count;
    char *mes = malloc(MSG_SIZE * sizeof(char));
    sprintf(mes, "Hello from %ld to %d", my_rank, dest);
    messages[dest] = mes;
    sem_post(&sems[dest]);
    sem_wait(&sems[my_rank]);
    printf("Thread %ld -> %s\n", my_rank, messages[my_rank]);
    return NULL;
}

int main(int argc, char **argv)
{
    thread_count = strtol(argv[1], NULL, 10);

    messages = malloc(thread_count * sizeof(char *));

    pthread_t *thread_handles =
        malloc(thread_count * sizeof(pthread_t));

    sems = malloc(thread_count * sizeof(sem_t));

    for (int i = 0; i < thread_count; i++)
        sem_init(&sems[i], 0, 0);

    for (long i = 0; i < thread_count; ++i)
        pthread_create(&thread_handles[i],
            NULL, routine, (void *)i);

    for (int i = 0; i < thread_count; ++i)
        pthread_join(thread_handles[i], NULL);

    free(thread_handles);

    for (int i = 0; i < thread_count; i++)
        sem_destroy(&sems[i]);

    for (int i = 0; i < thread_count; i++)
        free(messages[i]);
    free(messages);
```

```
    free(sems);

    return 0;
}
```

Бинарный семафор можно использовать в качестве мьютекса. Основное отличие заключается в том, что для семафоров нет понятия владения. Поднять и опустить семафор может любой поток. Что же касается мьютекса, то открыл его может только тот поток, который закрыл (завладел).

Небинарный семафор можно использовать, например, для контроля ограниченных ресурсов.

2.3 Pthreads: потокобезопасность

2.4 Pthreads: барьеры

2.5 MPI

2.6 MPI

2.7 MPI

2.8 OpenMP

2.9 OpenMP

2.10 OpenMP

2.11 CUDA

2.12 CUDA

2.13 CUDA

2.14 Отладка и профилирование параллельных программ

Глава 3

Лабораторные работы

3.1 Pthreads

!!!Дедлайн 22 октября 23:59 МСК!!!

3.1.1 Вычисление числа π

Описание задачи. Требуется определить число π по методу Монте-Карло.

Именно, пусть в квадрат со стороной $2r$ бросается дротик (монетка, кошка или другой предмет). Нарисуем внутри квадрата вписанную окружность с радиусом r. Ясно, что отношение количества попаданий в окружность к общему количеству бросков будет равно $\frac{\pi}{4}$. Следовательно, число π можно представить как

$$\pi \approx 4\frac{\text{Количество попаданий в окружность}}{\text{Общее количество бросков}} \tag{3.1}$$

Формат входных данных. Запуск программы из командной строки
`./program nthreads ntrials`,
где `nthreads` — количество потоков,
`ntrials` — общее количество попыток.

Формат выходных данных. Вывести на экран получившееся значение числа π.

3.1.2 Множество Мандельброта

Описание задачи. Множество Мандельброта — это совокупность всех чисел $c \in \mathbb{C}$, для которых последовательность

$$z_{n+1} = z_n^2 + c \tag{3.2}$$

является ограниченной для всех значений $n \in \mathbb{N}$. Иными словами, число c принадлежит множеству Мандельброта тогда и только тогда, когда

$$(\exists A \in \mathbb{R}) \quad (\forall n \in \mathbb{N}) \quad |z_n| < A$$

Можно доказать, что в последнем равенстве A можно взять равным двум, т.е. число c принадлежит множеству Мандельброта тогда и только тогда, когда

$$(\forall n \in \mathbb{N}) \quad |z_n| < 2, \quad z_1 = 0$$

Формат входных данных. Запуск из командной строки
`./program nthreads npoints`,

где `nthreads` — количество потоков,
`npoints` — количество точек.

Формат выходных данных. Файл формата `csv` с координатами точек множества Мандельброта.

3.1.3 Реализация read-write lock

Описание задачи.

В `pthreads` есть тип данных `pthread_rwlock_t`. Инициализация и уничтожение осуществляются по знакомой схеме:

```
int pthread_rwlock_init(
        pthread_rwlock_t*       lock,     /*out*/
        pthread_rwlockattr_t    attr      /*in*/
);

int pthread_rwlock_destroy(
        pthread_rwlock_t*       lock     /*out*/
);
```

Данный тип похож на мьютексы, но допускает две фунции `lock`:

```
int pthread_rwlock_rdlock(
    pthread_rwlock_t*     lock     /*in/out*/
);

int pthread_rwlock_wrlock(
    pthread_rwlock_t*     lock     /*in/out*/
);
```

Как и для мьютексов, есть лишь одна функция типа `unlock`:

```
int pthread_rwlock_unlock(
    pthread_rwlock_t*     lock     /*in/out*/
);
```

Когда поток получает блокировку чтения, то сколько угодно других потоков могут также получить блокировку на чтение, но ни один поток не получит блокировку на запись, пока все читающие потоки не вызовут функцию `unlock` для данной переменной.

Когда поток получает блокировку на запись, то ни один другой поток не сможет получить блокировку ни на чтение, ни на запись.

Как это можно организовать?.

Создать структуру, включающую в себя:

- Мьютекс
- Две условные переменные (одну для читателей, другую для писателей)
- Счетчик читателей (сколько потоков в данный момент читают)
- Счетчик количества потоков, ожидающих получения блокировки на чтение
- Счетчик количества потоков, ожидающих получения блокировки на запись
- Флаг, показывающий, получил ли блокировку хотя бы один писатель в данный момент

Что нужно сделать

1. Создать свою реализацию типа `rwlock` и функций `rdlock, wrlock`.
2. Апробировать эту реализацию на примере односвязного списка. См. файл `pth_ll_rwl.c`. Сравнить производительность своей реализации с библиотечной.

Обратите внимание, что в программе `pth_ll_rwl` используется свой генератор случайных чисел, т.к. стандартный `rand()` не является потокобезопасным: `seed` там один на все потоки.

3.1.4 Критерии оценивания

На оценку 5 достаточно выполнить первую или вторую задачу.
На оценку 7 достаточно выполнить первую и вторую задачи.
На оценку 10 нужно выполнить все три задачи.
Необходимо предоставить исходные коды программ и текст отчета, который бы содержал описание алгоритма и оценки времени работы, ускорения и эффективности разработанных программ в зависимости от объема входных данных и количества потоков. Провести анализ полученных результатов, сделать выводы.
За что может быть снижена оценка:

- Неоптимальность или ошибочность алгоритма.
- Неполные или недостоверные данные в отчете, отсутствие анализа результатов.