Тема 1. Синхронизация процессов, обмен сообщениями, семафоры, барьеры

1. Каналы передачи данных и методы обмена сообщениями

1.1. Свойства канала передачи данных

Для взаимодействия программ, запущенных на системах с распределённой памятью, используются каналы передачи данных. Основными характеристиками канала являются *пропускная способность* (время передачи одного байта) и *латентность* (инициализация передачи). Тогда время передачи сообщения объёма n байт:

$$T(n) = n \cdot T_{\text{байта}} + T_{\text{латентности}},$$

где $T_{\text{латентности}}$ может быть значительным (для Gbit Ethernet $\approx 50\,\mu$s) .

1.2. Синхронный и асинхронный методы передачи данных

Синхронный метод. `Send(proc, addr, size)` и `Recv(proc, addr, size)` блокируют процессы до начала обмена — оба конца должны вызвать соответствующую функцию, иначе процесс ждёт.
Асинхронный метод. `ASend(proc, addr, size)`, `ARecv(proc, addr, size)` и `ASync(proc)` возвращают управление сразу, ставя в очередь запрос на передачу/приём. Существуют небуферизованные и буферизованные (например, `ABSend`) варианты.

2. Семафоры

Определение Семафор — целочисленная неотрицательная переменная, над которой выполняются атомарные операции $P(S)$ и $V(S)$. Атомарность гарантирует, что при одновременных вызовах операции выполняются поочерёдно.

Операция $P(S)$

- Если $S > 0$, уменьшает S на 1.
- Иначе блокирует процесс до выполнения $V(S)$ другим процессом.

Операция $V(S)$ Увеличивает S на 1, разблокируя при этом один ожидающий процесс (если есть).

Пример использования

```
Semaphore Sem = 1;
...
P(Sem);
// Критическая секция
V(Sem);
```

Для более надёжного управления разделяемым ресурсом рекомендуется использовать *монитор* — набор процедур, инкапсулирующих все обращения к ресурсу и скрывающих детали работы с семафором.

3. Барьерная синхронизация

Барьер — функция, вызываемая всеми процессами; ни один не выйдет из неё, пока все не войдут. Реализуется двухэтапно:

1. `RecvOneFromAll()` — вход в барьер (запуск на дереве приёма).
2. `SendOneToAll()` — выход из барьера (рассылка ответа по дереву).

Соответствующий алгоритм (A14):

```
Barrier() {
 RecvOneFromAll();
 SendOneToAll();
}
```

Количество тактов для этого барьера оценивается как $2\log_2 p$, где p — число процессов. Альтернативно может применяться «бабочка»-барьер с $\log_2 p$ шагами при условии одновременного выполнения всех обменов.

Тема 2. Базовые алгоритмы

В этой теме рассмотрены основные параллельные методы статической и динамической балансировки нагрузки процессов.

1. Статические методы

1.1. Метод сдваивания

Делит задачу длины $n = 2^q$ на пары, суммы которых вычисляются за $\log_2 n$ этапов.

$$T_p(n) = \tau\,\log_2 n, \quad S_p(n) = \frac{n\,\tau}{\tau\log_2 n} = \frac{n}{\log_2 n}, \quad E_p(n) = \frac{S_p(n)}{p}\,,$$

где $p = n/2$.

1.2. Метод геометрического параллелизма

Разбиение двумерной сетки $n \times n$ на p прямоугольных блоков. Каждый процессор работает над своим блоком, обмены данных нужны лишь на границах соседних блоков.

$$S_p \approx p, \quad E_p \to 1 \text{ при больших } n, \quad E_p \text{ падает при малых размерах блоков.}$$

1.3. Метод конвейерного параллелизма

Вычисления организованы по «стадам» (слоям стены Фокса). Процессоры обрабатывают слои последовательно в конвейере:

$$T_p = \underbrace{(p-1)\tau}_{\text{запуск}} + \frac{nk}{p}\mu + \underbrace{(p-1)\tau}_{\text{завершение}},$$

где k — число рядов, μ — время обработки одного узла, τ — время обмена между соседями. Эффективность близка к 1 только если $p \ll k$.

2. Динамические методы

2.1. Метод коллективного решения

Есть единый управляющий процесс, координирующий распределение множества заданий непредсказуемой трудоёмкости между рабочими процессами.

$$T_p = T_1 + 2\,t_s \frac{n}{p}, \quad E_p \approx \frac{T_1}{T_1 + 2t_s(n/p)} \to 1 \text{ при } T_1 \gg \tfrac{n}{p} t_s,$$

где t_s — латентность обмена данными с управляющим :contentReference[oaicite:6]index=6.

2.2. Метод диффузной балансировки

(Описан в главе 8; здесь упомянем для полноты) реакции «диффузии» избыточной нагрузки между соседями итеративно до достижения баланса.

3. Причины потери эффективности

- Недостаток внутреннего параллелизма.
- Дисбаланс нагрузки.
- Дополнительные затраты:
  - передача данных,
  - синхронизация (барьеры, семафоры),
  - дублирование вычислений,
  - спекулятивные операции.

Эти факторы подробно обсуждаются в разделах 4.5 и 3.6 монографии :contentReference[oaicite:10]index=

Тема 3. Параллельные алгоритмы сортировки данных

5.1 Постановка задачи

Сортировкой называется упорядочение элементов массива в неубывающем порядке. Исходный массив из n целых чисел распределён по p процессорам порциями A_i, по завершении сортировки каждая порция B_i должна быть упорядочена, причём $|B_i| = n/p$. Определим

$$T(n,p) \;=\; \text{время сортировки массива из } n \text{ элементов на } p \text{ процессорах.}$$

Требуется разработать алгоритмы, обеспечивающие $T(n,p) \to T(n,1)/p$ при больших n и p.

5.2 Последовательные алгоритмы сортировки

В таблице приведены асимптотические оценки числа операций при $p = 1$:

Алгоритм	Среднее число операций	Худшее число операций
Быстрая (qsort)	$11{,}7\,n\log_2 n$	$O(n^2)$
Слияние списков (lsort)	$9\,n\log_2 n$	$O(n\log n)$
Двухпутевое слияние (dsort)	$11\,n\log_2 n$	$O(n\log n)$
Пирамидальная (hsort)	$16\,n\log_2 n + 0{,}01n$	$18\,n\log_2 n + 38n$

5.3 Наилучший последовательный алгоритм (dhsort)

Комбинируя `hsort` и `dsort`, конструируют алгоритм `dhsort`: сначала малые блоки сортируют `hsort`, затем сливают методом `dsort`. Этот метод демонстрирует наименьший эмпирический коэффициент K в оценке $T(n) = 10^{-9}K(n)\,n\log_2 n$ и выбран за «наилучший» последовательный сортировщик.

5.4 Масштабируемые параллельные алгоритмы

5.4.1 Сети сортировки

Сеть сортировки — это фиксированная схема компараторов, не зависящая от значений данных. Элементы изображают горизонтальными линиями, компараторы — вертикальными отрезками. Каждый компаратор сравнивает две линии и обменивает элементы, чтобы по верхнему выходу шёл меньший, по нижнему — больший элемент.

5.4.2 Сеть четно-нечетной перестановки

На каждом из p шагов компараторы нечетной группы обрабатывают пары $(2i-1, 2i)$, четной — $(2i, 2i+1)$. Всего p шагов, время $Q(p) = p$.

5.4.3 Сеть Бэтчера (обменная сортировка со слиянием)

Рекурсивно делят p линий на две части, сортируют каждую и затем применяют (n, m)-сеть нечетно-четного слияния. Обеспечивает $O(\log^2 p)$ шагов.

5.4.4 Сортировка больших массивов

Для $n \gg p$ применяют двухфазный подход:

1. На каждом процессоре сортировка фрагмента длины $m = n/p$ алгоритмом `dhsort`.
2. Глобальное слияние фрагментов через сеть сортировки на p линиях.

Такой метод допускает сортировку массивов, размер которых ограничен лишь совокупной памятью системы.

5.4.5 Сравнение сетей сортировки

- Простая сеть вставки: $Q(p) = 2p - 3$.
- Четно-нечетная сеть: $Q(p) = p$.
- Сеть Бэтчера: $Q(p) = O(\log^2 p)$.

Хотя первая сеть и быстрее на малых p, обменные сети (особенно Бэтчера) уменьшают число шагов при росте процессоров .

5.5 Результаты численных экспериментов

На суперкомпьютере МВС 1000М (768 процессоров Myrinet) при сортировке 10^8 элементов наблюдались следующие показатели ускорения S и эффективности E:

Значительный объём коммуникаций в фазе глобального слияния снижает эффективность при росте p .

p	T, с	B,%	S	E,%	$E_{\max}$,%	$Q(p)$
1	83.51	100.0	1.00	100	100	1
2	46.40	90.0	1.80	90	100	2
4	29.68	70.4	2.81	70	96	4
8	19.95	52.3	4.19	52	90	6
16	12.36	42.2	6.75	42	82	10
27	9.32	33.2	8.97	33	64	15

Тема 4. Рациональная декомпозиция расчётных сеток

1. Модель сетки как графа

Расчётную сетку заменяем неориентированным графом $G = (V, E)$, где вершины V — узлы сетки, а ребра E — связи соседства. Каждая вершина v_i и ребро (v_i, v_j) могут иметь веса $w(v_i)$ и $w(v_i, v_j)$, отражающие вычислительную нагрузку и объём обмена данными соответственно.

2. Критерии рациональной декомпозиции

1. **Классический (равномерность + минимизация разреза).** Ищут разбиение $V = V_1 \cup \cdots \cup V_p$ с равными весами $\sum_{v \in V_k} w(v) \approx \frac{\sum w(v)}{p}$ и минимальным числом разрезанных ребер.

2. **Выделение обособленных доменов.** Домены одного типа (основные) не смежны друг с другом, все их границы соприкасаются только с поддерживающим доменом .

3. **Минимизация максимальной степени макрографа.** Строят макрограф, вершины которого — домены, а ребра взвешены суммами весов разрезанных связей. Сокращают максимальную степень (число соседей) любой вершины макрографа.

4. **Связность доменов.** Каждый домен должен образовывать связный подграф исходного графа, без «островков».

3. Простейшие методы декомпозиции

3.1 По исходной нумерации узлов

Вершины с номерами в равных интервалах $[k\frac{n}{p}, (k+1)\frac{n}{p} - 1]$ назначаются доменам. Гарантирует равномерность, но разрывает границы на множестве ребер.

3.2 Индексная бисекция для регулярных сеток

Для прямоугольной сетки рекурсивно делят по координатам (между столбцами, рядами), но баланс по числу узлов остаётся грубым и далёк от оптимального (см. алгоритм A24).

4. Рекурсивная бисекция

Общая стратегия: для k доменов строят бинарное дерево разбиения за $k - 1$ шагов. На каждом шаге одну часть делят на две (алгоритм A23) .

5. Декомпозиция произвольных графов

5.1 Иерархический подход

1. *Огрубление*: формируют вложенную последовательность уменьшенных графов.

2. *Начальная декомпозиция*: разбивают самый мелкий граф.

3. *Восстановление и локальное уточнение*: разворачивают граф, уточняя границы локальным алгоритмом перемещения соседних вершин.

5.2 Спектральная бисекция

Для двудольного разбиения решают $\min|E_c|$ при равных объёмах, используя второй собственный вектор матрицы Лапласа L. Компоненты этого вектора задают порядок вершин; меньшие попадают в первый домен, большие — во второй.

5.3 Алгоритм инкрементного роста

Начинают с p случайных «ядер», расширяют домены слоями вершин по минимальному расстоянию до границ, выполняют локальное уточнение и повторяют итеративно до связности «ядер» заданного уровня.

6. Декомпозиция больших сеток

6.1 Координатная рекурсивная бисекция

Применяют рекурсивную бисекцию по координатам без огрубления, быстро, но с худшим качеством разбиения, часто в качестве предварительного шага.

6.2 Двухуровневая стратегия хранения

Сетку разбивают на малые микродомены, строят макрограф их связей и выполняют декомпозицию макрографа, что ускоряет многократную балансировку при варьирующем числе процессов.

Тема 5. Визуализация сеточных данных

5.1. Клиент-серверная технология

Для визуализации больших сеточных данных применяется архитектура «клиент-сервер», где сервер визуализации, запущенный на многопроцессорном кластере, выполняет параллельную обработку и сокращение объёма данных, а клиент, работающий на рабочей станции, лишь отображает и интерактивно управляет готовой виртуальной сценой. **Сервер визуализации:**

- Параллельная декомпозиция и фильтрация сетки;
- Вычисление и огрубление изоповерхностей;
- Сжатие и передача минимального набора примитивов.

Клиент визуализации:

- Приём готовых треугольников (или микродоменов);
- Преобразование в графические объекты и вывод на экран;
- Манипуляции сценой (вращение, масштабирование) без обращения к серверу.

5.2. Online vs Offline визуализация

Online Визуализация «на лету» при одновременной работе расчётов и отрисовки. Позволяет быстро реагировать на промежуточные результаты, но требует высокой пропускной способности канала и синхронизации с расчётами.

Offline Полная предварительная обработка и запись огрублённых данных на диск, последующий их просмотр без нагрузки на расчётный кластер. Даёт максимальную интерактивность на клиенте и надёжность работы при низкой сетевой скорости .

5.3. Этапы визуализации

1. *Моделирование:* декомпозиция сетки и запись структуры + фрагментов сетки и функций.
2. *Ввод/декомпозиция на сервере:* чтение записанных данных, распределение фрагментов по процессорам.
3. *Фильтрация/огрубление:* аппроксимация исходных примитивов (изоповерхностей) в каждом микродомене.
4. *Сборка и иерархическое уменьшение:* объединение огрублённых фрагментов и повторное редуцирование на уровне групп процессоров.
5. *Формирование виртуальной сцены:* пакетирование треугольников и передача клиенту.
6. *Клиент:* приём, преобразование и отрисовка, интерактивное управление сценой.

5.4. Визуализация изоповерхностей

5.4.1. Определение и сеточная аппроксимация

Изоповерхность — множество точек, где скалярная функция равна заданному уровню. В сеточном подходе она строится как триангуляция: набор треугольников, вершины которых лежат на линиях между узлами исходной сетки.

5.4.2. Форматы описания триангуляции

- *Поток вершин и индексов треугольников*;
- *Списки смежности для каждой вершины*;
- *Байтовые и битовые маски для границ микродоменов.*

Выбор формата влияет на объём передаваемых данных и скорость построения сцены .

5.4.3. Метод редукции

Реализация функции редукции: в каждом микродомене отсеиваются вершины с малыми углами или малыми координатными градиентами, пока число треугольников не станет ниже порога. Алгоритм требует $O(n \log n)$ времени и легко распараллеливается по доменам .

5.4.4. Заполняющие пространство триангуляции

Используются алгоритмы Delaunay для аппроксимации «дырок» после редукции, сохраняя связность и предотвращая утечку визуальных артефактов .

5.4.5. Параллельные алгоритмы аппроксимации

- *Метод «редуцирования»:* каждый процессор работает над своим микродоменом и формирует локальную огрублённую триангуляцию.
- *Иерархический сбор и повторное редуцирование:* группы процессоров по дереву образуют более крупные домены и выполняют второй проход редукции :contentReference[oaicite:18]index=18.

5.4.6. Многоуровневое огрубление

Строится многоуровневая иерархия доменов: сначала редуцируются низкоуровневые микродомены, затем их объединения и т. д., что позволяет обрабатывать данные любого объёма на конечном числе процессоров :contentReference[oaicite:19]index=19.

5.4.7. Примеры визуализации

На примере обтекания сферы газом видно, что после редукции сохраняется ключевая форма изоповерхности, а число треугольников сокращается в десятки раз (рис. 104).

5.5. Ввод–вывод сеточных данных

5.5.1. Время чтения vs время обработки

Для трёх размеров сетки ($50\times50\times10$, $500\times500\times100$, $500\times500\times1000$) на 20 процессорах замеры показали, что ввод данных по NFS занимает до 90% общего времени, а чистое огрубление (вычисление и передача на клиент) — лишь доли секунды.

5.5.2. Распределённый ввод–вывод

Двухуровневая схема хранения микродоменов на файловых серверах и их параллельное чтение каждым процессором позволяют снизить задержки ввода-вывода и повысить масштабируемость.

5.5.3. Огрубление и сжатие скалярных функций

Стандартное безпотерьное сжатие неэффективно для вещественных данных. Предложен метод усечения младших бит мантиссы (float), позволяющий сокращать объём в сотни раз без заметных визуальных потерь (рис. 110) и ускорять ввод-вывод на порядки.

Тема 6. Динамическая балансировка нагрузки процессоров

6.1. Стратегии балансировки нагрузки

Задача балансировки нагрузки на многопроцессорных системах зависит от динамики изменения трудоёмкости отдельных подзадач. Можно выделить три класса сценариев:

1. **Статическая нагрузка.** Веса узлов (или заданий) не меняются со временем: $w_i(t) = \text{const}$. Подходят статические алгоритмы декомпозиции.
2. **Квазистационарная нагрузка.** Веса узлов медленно меняются от шага к шагу. Эффективны локальные диффузные методы перераспределения между соседями.
3. **Сильно динамическая нагрузка.** Веса узлов быстро и непредсказуемо меняются, заранее их не узнать. Требуются безприорные методы, например, *серверный параллелизм.*

6.2. Метод диффузной балансировки

Метод опирается на локальный обмен вычислительной работы между соседними процессорами, сохраняя при этом «локальность» данных и минимизируя коммуникации. В общих чертах:

- Каждый процессор периодически оценивает свою текущую нагрузку t_i и предлагает передать часть узлов тем соседям, у которых время обработки больше или меньше среднего.
- Новое количество узлов n_i' на процессоре i может быть рассчитано централизованно за $O(p)$, либо приближённо по информации от процессов $i-1, i, i+1$.
- Перераспределение выполняется редко, только когда накопившийся дисбаланс снижает эффективность ниже заданного порога.

6.3. Серверный параллелизм

При моделях с непредсказуемым порождением горячих точек (например, горение метанового факела) статическая или диффузная балансировка не работает. В методе серверного параллелизма:

- Нет единого управляющего процесса: каждый рабочий процессор сам запрашивает следующие задания из общего пула.
- Процессор в первую очередь обрабатывает свои локальные «горячие» точки, а затем по потребности запрашивает удалённые.
- Семантика пула реализуется через атомарные операции добавления/извлечения заданий и семафоры для сигнализации о наличии работы.

Такой подход устраняет узкое место единственного менеджера и обеспечивает адаптивную равномерную загрузку.

6.4. Адаптивное интегрирование на общей памяти

Задачи, в которых новые подзадачи (отрезки интегрирования) динамически порождаются на лету, решаются с помощью *пула заданий* и семафоров. Общая схема алгоритма:

1. Глобальный стек отрезков интегрирования и семафор `sem_task_present`.
2. Каждый процессор:
   - Захватывает семафор `sem_task_present` (P), извлекает отрезок.
   - Выполняет оценку и, при необходимости, разбивает отрезок на более мелкие, помещая их обратно в стек.
   - При заполнении стека терминальными отрезками сигнализирует (V) для остальных, что работа завершена.
3. Суммирование частичных результатов защищено семафором `sem_sum` для исключения гонок.

Эта схема демонстрирует высокую эффективность на малом числе процессов (2–4) даже при нерегулярной трудоёмкости задач.

Тема 7. Псевдослучайные числа для многопроцессорных систем

7.1. Требования к генераторам для МВС

Генераторы псевдослучайных чисел в многопроцессорных системах должны удовлетворять двум ключевым требованиям:

1. *Достаточно большой период*, чтобы избежать повторов в ходе долгих расчётов.
2. *Прямой доступ к любому элементу* последовательности без необходимости генерировать все предшествующие члены, иначе эффективность параллельных методов снижается из-за затрат на прокрутку последовательности.

Кроме того, важно обеспечить *воспроизводимость* генерируемых фрагментов при многократных запусках и независимость фрагментов на разных процессорах.

7.2. Линейно-конгруэнтные генераторы

ЛКГ задаётся:

$$u_{n+1} \;=\; (a\,u_n + c) \bmod m,$$

где коэффициенты подбираются так, чтобы максимизировать период при m не превосходящем разрядность процессора.

Skip-ahead и leapfrog Чтобы на p процессорах получить непересекающиеся фрагменты:

$$u_{n+p} = a^p u_n + \frac{a^p - 1}{a - 1} c \pmod m,$$

что позволяет вычислить шагом p за $O(\log n)$ операций без последовательного прохода. Однако при выборе p-разряда фрагментов (leapfrog) могут появляться корреляции (рис. 52).

7.3. M-последовательности (генераторы Фибоначчи)

M-последовательности задаются рекуррентой порядка r:

$$U_n = U_{n-r}\,\Theta\,U_{n-s} \quad (\Theta \in \{+, -, *, \bmod\}),$$

например, $U_n = U_{n-20} + U_{n-33} \pmod 2$ даёт период $2^{33} - 1$. Для skip-ahead применяют «jump-ahead» матрицы размера $r \times r$, требующие $O(r^3 \log v)$ операций, или улучшенный $O(r^2 \log v)$ метод — для фиксированного шага v :contentReference[oaicite:10]index=10.

7.4. Проверка примитивности полиномов

M-последовательности строятся на примитивных полиномах $G(x)$ над $\mathbb{F}_2$:

$$x^r \equiv 1 \pmod{G(x)}, \quad G(x) \text{ примитивен},$$

что гарантирует период $2^r - 1$. Примеры примитивных триномов:

$$G(x) = x^{511} + x^{15} + 1, \quad G(x) = x^{1023} + x^7 + 1$$

с периодами $2^{511}-1$ и $2^{1023}-1$ соответственно. Проверка примитивности сводится к выполнению условий

$$x^{2^r-1} \equiv 1 \pmod{G(x)}, \quad x^{(2^r-1)/h_i} \not\equiv 1 \pmod{G(x)}$$

для всех простых делителей h_i числа $2^r - 1$.

7.5. Тестирование генераторов

Качество последовательностей оценивают стандартным пакетом Diehard: 319 серий тестов (парковочный тест, ранги матриц, «обезьяний» тест и др.). ПСЧ считаются плохими, если более 6 p-value равны 0 или 1. Таблица результатов для различных генераторов демонстрирует, что хорошо подобранные M-последовательности и ЛКГ проходят все группы тестов.

Тема 8. Отказоустойчивые алгоритмы для многопроцессорных вычислительных систем

8.1. Методы обнаружения ошибок

Сбой в многопроцессорной системе может проявляться как остановка процесса, повреждение памяти или потеря сообщений. Для обнаружения ошибок применяются:

- контрольные суммы и избыточные коды;
- контроль таймаутов (если процесс не отвечает);
- репликация задач и сравнение результатов;
- протоколы подтверждений и отказов (например, ACK/NAK).

Алгоритмы отказоустойчивости должны минимизировать влияние сбоя на оставшиеся процессы и позволять восстановить согласованное состояние системы.

8.2. Восстановление состояния

Восстановление после сбоя осуществляется путём отката к сохранённому снимку состояния (checkpoint). Возможны три подхода:

1. **Периодическое сохранение состояния.** Все процессы сохраняют свои данные через равные интервалы времени.
2. **Контролируемые контрольные точки (coordinated checkpointing).** Согласованное сохранение всех процессов в момент времени t.
3. **Инкрементальные снимки.** Хранятся только изменения с последнего контрольного сохранения, что уменьшает затраты памяти.

8.3. Протокол Chandy–Lamport

Протокол консистентной фиксации Chandy–Lamport работает в асинхронной среде и позволяет:

- начать фиксацию состояния с любого процесса;
- передавать *метки снимков* (marker) по каналам;
- гарантировать, что все сообщения между процессами будут учтены корректно;
- получить глобальное согласованное состояние без остановки процессов.

Алгоритм сохраняет «чистоту» снимка: если m отправлено до метки, то оно должно быть получено до неё, или сохранено в журнал сообщений.

8.4. Другие протоколы консистентной фиксации

Протоколы логического времени: на базе Lamport-часов и векторных часов — позволяют частичную упорядоченность событий и привязку сообщений к временно́й шкале.

8.5. Стратегии резервирования

Отказоустойчивость может быть достигнута путём:

- *активного резервирования* — параллельный запуск копий задач (primary-backup);
- *пассивного резервирования* — запуск резервной копии только при сбое основной;
- *протоколов миграции* — перемещение состояния задачи на исправный процессор;
- *избыточного кодирования данных* — использование кодов Хэмминга, RAID-подобных схем.

Резервирование требует дополнительной памяти и процессоров, но позволяет повысить надёжность масштабируемых вычислений.