Дополнительная теоретическая справка

1. Основные директивы OpenMP

Какие директивы OpenMP вы использовали в своей работе и для каких целей?

- #pragma omp parallel - создание параллельной области, где код выполняется несколькими потоками одновременно
- #pragma omp for - распределение итераций цикла между потоками
- #pragma omp critical - создание критической секции для защиты общих данных
- #pragma omp parallel for - комбинированная директива для создания параллельной области с распределенным циклом
- #pragma omp sections - разделение кода на независимые секции для параллельного выполнения

2. Параллельные циклы

В чем разница между #pragma omp parallel и #pragma omp parallel for?

- #pragma omp parallel

создает параллельную область, где весь код внутри выполняется всеми потоками одновременно. Каждый поток выполняет одни и те же инструкции.

- #pragma omp parallel for

- это комбинация двух директив: сначала создается параллельная область, а затем итерации следующего за директивой цикла распределяются между потоками. Это удобно для параллельной обработки массивов и других последовательных данных.

3. Синхронизация потоков

Как работает директива #pragma omp critical **и в каких случаях она необходима?**

- #pragma omp critical

создает критическую секцию - участок кода, который может выполняться только одним потоком в данный момент времени.

- Она необходима, когда несколько потоков обращаются к общим данным, и хотя бы один из потоков изменяет эти данные.
- Пример использования: защита вывода в консоль, обновление общих счетчиков, модификация общих структур данных.

Какие еще способы синхронизации потоков в OpenMP вы знаете?

- atomic - для атомарных операций над скалярными переменными
- barrier - точка синхронизации, где потоки ожидают друг друга

- sections - для параллельного выполнения независимых блоков кода
- single - выполнение блока кода только одним потоком
- master - выполнение блока кода только главным потоком

4. Производительность параллельных программ

Почему в некоторых случаях последовательная версия программы работает быстрее параллельной?

- Накладные расходы на создание и управление потоками могут превышать выгоду от распараллеливания
- Синхронизация между потоками (критические секции, барьеры) создает дополнительные задержки
- При малом объеме данных время на распределение работы между потоками может быть больше времени самой работы
- Конкуренция за общие ресурсы (память, кэш) может снижать производительность
- Неоптимальное количество потоков может приводить к простоям

Как вы определяли оптимальное количество потоков для ваших задач?

- Теоретически оптимальное количество потоков обычно равно количеству физических ядер процессора
- На практике проводились замеры производительности с разным количеством потоков
- Учитывалась поддержка Hyper-Threading (виртуальные ядра)
- Для CPU-intensive задач оптимальное количество потоков обычно равно количеству физических ядер
- Для I/O-bound задач может быть выгодно использовать больше потоков

Что такое "накладные расходы" при параллельном программировании?

- Время на создание и уничтожение потоков
- Время на синхронизацию между потоками
- Конкуренция за общие ресурсы (память, кэш, ввод-вывод)
- Неравномерное распределение нагрузки между потоками
- Ложное разделение кэш-линий (false sharing)

5. Алгоритм быстрой сортировки

Как вы реализовали параллельную версию быстрой сортировки?

- Рекурсивное разделение массива на подмассивы

- Использование директивы sections для параллельной сортировки подмассивов
- Применение порога параллелизма для предотвращения избыточного распараллеливания
- Синхронизация между потоками при разделении массива

Что такое порог параллелизма и как он влияет на производительность?

- Порог параллелизма - минимальный размер подмассива, при котором имеет смысл создавать новые потоки
- Если размер подмассива меньше порога, используется последовательная сортировка
- Слишком маленький порог приводит к избыточному распараллеливанию и росту накладных расходов
- Слишком большой порог уменьшает степень параллелизма и может оставить часть ядер простаивать
- Оптимальное значение порога зависит от характеристик процессора и размера данных

Почему при малых размерах массива неэффективно использовать параллельную сортировку?

- Накладные расходы на создание потоков и синхронизацию становятся соизмеримыми с временем сортировки
- Уменьшается локальность данных, что ухудшает использование кэш-памяти
- Увеличивается количество переключений контекста между потоками
- Снижается эффективность использования вычислительных ресурсов

6. Метрики и измерения

Как вы измеряли время выполнения программ?

- В OpenMP: omp_get_wtime() для высокоточного измерения времени
- В стандартной библиотеке C: clock() или time.h
- В C++: std::chrono для высокоточных измерений
- Измерялось только время вычислений, без учета инициализации и освобождения ресурсов
- Проводилось несколько прогонов для усреднения результатов

Какие метрики эффективности параллельных программ вы знаете?

- Ускорение (speedup): $S = T_1 / T_n$, где T_1 - время выполнения на одном потоке, T_n - на n потоках
- Эффективность: E = S / n, показывает, насколько эффективно используются вычислительные ресурсы
- Масштабируемость - способность алгоритма эффективно использовать растущее количество ресурсов
- Баланс загрузки - равномерность распределения работы между потоками

Что такое ускорение (speedup) и эффективность (efficiency) параллельной программы?

- Ускорение показывает, во сколько раз быстрее выполняется программа на n потоках по сравнению с одним потоком
- Идеальное ускорение - линейное (в n раз при n потоках)
- Эффективность показывает, насколько эффективно используются вычислительные ресурсы
- Эффективность = speedup / n, где n - количество потоков
- Эффективность 1 (или 100%) означает идеальное использование ресурсов

Почему не всегда достигается линейное ускорение при увеличении числа потоков?

- Накладные расходы на синхронизацию и управление потоками
- Конкуренция за общие ресурсы (память, кэш, ввод-вывод)
- Неравномерное распределение нагрузки между потоками
- Ограничения пропускной способности памяти
- Зависимости по данным, требующие синхронизации
- Ложное разделение кэш-линий (false sharing)

7. Теоретические основы

В чем разница между процессами и потоками?

- Процессы - это изолированные единицы выполнения с собственным адресным пространством
- Потоки - это легковесные потоки выполнения в рамках одного процесса, разделяющие память
- Процессы не разделяют память, потоки в одном процессе разделяют память
- Создание и переключение потоков требует меньше ресурсов, чем процессов
- Аварийное завершение потока может повлиять на весь процесс

Что такое гонка данных (data race) и как ее избежать?

- Гонка данных возникает, когда несколько потоков обращаются к общим данным, и хотя бы один из потоков изменяет эти данные
- Способы избежания:
    - Использование критических секций (#pragma omp critical)
    - Атомарные операции (#pragma omp atomic)
    - Локальные переменные для каждого потока (private, firstprivate)
    - Уменьшение времени нахождения в критических секциях
    - Алгоритмы, минимизирующие обмен данными между потоками

Какие модели параллельного программирования вы знаете?

- С общей памятью (OpenMP, Pthreads) - потоки разделяют общее адресное пространство
- С передачей сообщений (MPI) - процессы обмениваются сообщениями
- Гибридные модели (OpenMP + MPI) - комбинация подходов
- Потоковая модель (CUDA, OpenCL) - для GPU-вычислений
- Асинхронная модель (акторы, корутины)

В чем преимущества и недостатки модели с общей памятью?

Преимущества:

- Простота программирования
- Высокая скорость обмена данными между потоками
- Гибкость в управлении параллелизмом
- Возможность использования общих структур данных

Недостатки:

- Ограниченная масштабируемость (обычно в пределах одного узла)
- Проблемы с синхронизацией доступа к общим данным
- Сложность отладки гонок данных
- Ограниченная переносимость между разными архитектурами

8. Особенности реализации

Почему вы выбрали именно такой способ распараллеливания для ваших задач?

- Учитывалась природа задачи (независимость итераций, объем вычислений)
- Ориентация на архитектуру с общей памятью
- Простота реализации и отладки
- Возможность использования стандартных средств OpenMP

- Баланс между сложностью реализации и приростом производительности

Как вы обрабатываете исключительные ситуации в параллельных программах?

- Использование критических секций для защиты общих данных
- Глобальные обработчики ошибок
- Проверка возвращаемых значений функций
- Избегание исключений внутри параллельных регионов
- Корректное освобождение ресурсов при ошибках

Какие ограничения накладывает архитектура процессора на параллельные вычисления?

- Количество физических ядер определяет максимальный уровень параллелизма
- Размеры и иерархия кэш-памяти влияют на производительность
- Пропускная способность памяти может стать узким местом
- Поддержка Hyper-Threading позволяет эффективнее использовать вычислительные ресурсы
- Неравномерность доступа к памяти (NUMA) в многосокетных системах

Как можно было бы распараллелить ваши программы на кластере с распределенной памятью?

- Использование MPI для обмена данными между узлами
- Гибридный подход: OpenMP + MPI
- Распределение данных между узлами
- Учет задержек при передаче данных
- Оптимизация размера передаваемых сообщений
- Учет топологии сети при планировании обменов