Programming Languages Project

Rohit Shah (IMT2021027), Keshav Chandak (IMT2021003),
Sunny Kaushik (IMT2021007)

May 13, 2024

Chapter 1

Problem Description

Harnessing Functional Programming for Parallelism

This project is aimed at exploring and implementing parallel programming concepts using the constructs and paradigms provided by functional programming languages.

- The investigation will focus on how features such as higher-order functions, lazy evaluation, persistent data structures, futures/promises, actors, parallel collections, etc., can be leveraged to design and implement parallel algorithms.
- A set of computational problems suitable for parallel processing (e.g., numerical simulations, data processing algorithms,and various other algorithms) will be selected, and algorithms will be designed using functional programming concepts that exploit parallelism.
- The designed algorithms will be implemented in functional programming language(OCaml). The implemented solutions will be compared with equivalent algorithms written in imperative programming languages with parallel constructs. Differences in code complexity, maintainability, performance, and scalability will be analyzed.

Chapter 2

Solution Outline

2.1 Deliverables

We aim to leverage features and strategies such as higher-order functions, lazy evaluation, etc., to implement algorithms in a parallel fashion in OCaml.

For this purpose, we have selected the following algorithms:

1. **Linear Algebra:**
   - Matrix Multiplication
   - QR Decomposition
   - Singular Value Decomposition
   - LU Decomposition
   - Bareiss Algorithm for Computing Determinant
2. **Sorting Algorithms:**
   - Merge Sort
   - Quick Sort
3. **Graph Traversal Algorithms:**
   - Depth-First Search (DFS)
   - Breadth-First Search (BFS)
4. **N-body Problem**

Furthermore, we aim to assess the performance of OCaml implementations in comparison to other imperative solutions. In the section 6 (Benchmarking & Performance Analysis), we elaborate on our use of the `hyperfine` tool for this purpose. Additionally, we plan to conduct unit testing of our parallel code to ensure consistency and correctness compared to normal variants, while also evaluating performance improvements.

2.2 Setup Instructions

Follow the below instructions to setup the project locally.

```
git clone "https://github.com/RohitShah1706/Parallel_Progamming_in_OCaml.git"
cd Parallel_Progamming_in_OCaml

# Update opam
opam update
opam upgrade

# Install OCaml 5.0.0 for OCaml Multicore
opam switch create 5.0.0
eval $(opam env --switch=5.0.0)

# Development tools
opam install ocaml-lsp-server odoc ocamlformat utop

# Install Domainslib
opam install domainslib
```

2.3 Running the Code

You need to have OCaml v5.0.0 installed for `Multicore OCaml` to work. For this follow above instructions to create a new switch for OCaml v5.0.0

```
# Compile the code
dune build @all

# Run the code
# replace fibonacci with any other executables mentioned in bin/dune
dune exec fibonacci

# in each executable provide parallel as argument to run parallel version
dune exec fibonacci parallel
```

2.4 Framework

We have employed the `Dune` build system in conjunction with the `Domainslib` and `OCaml Multicore` libraries to establish a robust framework for our project. Within this framework, we leverage key functionalities such as `Task.async` and `Task.parallel_for` to facilitate parallel computation.

2.4.1 Domains in OCaml Multicore

`Domains` serve as the fundamental unit of parallelism in OCaml Multicore. Each domain corresponds to an operating system thread and can execute computations concurrently.

- **Domain Spawning:** The `Domain.spawn` function is utilized to execute computations in parallel with the calling domain. Subsequently, the `Domain.join` function is employed to synchronize the completion of the spawned domain's execution.
- **Work Stealing:** The MultiCore OCaml runtime (version 5.0.0) employs a work-stealing scheduling algorithm to maximize CPU utilization and overall system throughput by distributing tasks across available CPU cores.

2.4.2 Domainslib

`Domainslib` provides support for nested-parallel programming in OCaml. It facilitates the asynchronous execution of parallel tasks through mechanisms such as async/await. Additionally, it offers parallel iteration functions and an efficient implementation of a work-stealing queue to optimize task sharing among domains.

2.4.3 Task Pool vs. Individual Domains

While creating new domains incurs overhead, reusing existing domains or employing task pools can mitigate this cost. Task pools, such as the one created using `Task.setup_pool`, allow the execution of parallel workloads within the same set of domains initialized at the beginning of the program.

```
open Domainslib
let pool = Task.setup_pool ~num_domains:3 ()
val pool : Task.pool = <abstr>
```

We instantiated a task pool comprising three domains (excluding the parent domain), resulting in a total of four domains. This pool facilitates the execution of parallel tasks efficiently.

2.4.4 Task.parallel_for

`Task.parallel_for` is particularly beneficial for parallelizing iterative tasks. For asynchronous parallel tasks, consider utilizing *async/await*. This function operates within a specified task pool, iterating over a range of indices with a user-defined function applied to each index.

Pros & Cons of `Task.parallel_for`

- **Implicit Barrier (Pro):** All tasks waiting for execution within the same pool commence execution only after completion of all chunks in the `Task.parallel_for` loop. Hence, explicit synchronization barriers are unnecessary.
- **Unpredictable Execution Order (Con):** `Task.parallel_for` may execute iterations in arbitrary order, leading to variation between successive runs of identical code. Consequently, if loop iterations have dependencies, such as each iteration relying on the preceding one, incorrect results may arise.

2.4.5 Task.async/await

The use of *Async-Await* offers increased flexibility in executing parallel tasks, particularly beneficial in recursive functions. The Task API provides the capability to designate specific tasks to run on a task pool.

- **Task.async**: This function executes the task asynchronously within the pool and returns a `promise`, a computation that is yet to be completed. Upon completion of the execution, the result will be stored in the promise.
- **Task.await**: It awaits the completion of the promise's execution. Once finished, it retrieves the result of the task. In the event of the task raising an uncaught exception, `await` propagates the same exception.

Chapter 3

Main References

- Parallelising your OCaml Code with Multicore OCaml :
  - This project leverages Multicore OCaml's concurrency and parallelism features, utilizing domains as the unit of parallelism.
  - DomainsLib library is integrated, providing high-level parallel primitives such as channels and task pools. With DomainsLib, it easily parallelizes their algorithms, benefiting from dynamic thread management and efficient parallel execution. This addresses common pitfalls like granularity and shared state contention, ensuring optimized parallel performance.
  - Multicore OCaml simplifies migration and maintenance, while the separation of libraries like DomainsLib from the compiler cycle reduces overhead.
  - By combining Multicore OCaml and DomainsLib, the project aims to achieve enhanced parallelism, scalability, and performance in their OCaml codebase.
- OCaml official Documentation on Parallelism
  - The documentation covers the high-level parallel programming using Domainslib followed by low-level primitives exposed by the compiler.
  - Understanding Concurrency and Parallelism: We're distinguishing between concurrency and parallelism, ensuring clarity in our approach. Concurrency enables overlapped execution of tasks, while parallelism allows simultaneous execution. By understanding this distinction, we tailor our parallelization strategy accordingly.
  - Optimizing Parallel Tasks: Learning from the tutorial's emphasis on avoiding excessive domain spawning, we're mindful of the overhead associated with creating and managing domains. We aim to optimize our parallel tasks to strike a balance between concurrency and parallelism, ensuring efficient resource utilization.

- Benchmarking and Performance Measurement: We're adopting a benchmarking approach similar to the tutorial, using tools like hyperfine to measure the performance of our parallel programs. By benchmarking different configurations and analyzing the results, we can assess the scalability and effectiveness of our parallelization efforts.
- Exploring Parallel Iteration Constructs: Inspired by the tutorial's exploration of parallel iteration constructs like parallel-for, we're investigating similar techniques to parallelize iterative algorithms within our project. This approach allows us to harness the power of parallelism in tasks such as numerical computation, data processing algorithms like Singular Value Decomposition and simulations like .

- Code examples

  Using the Github repository we are referencing the parallelisation of Fibonacci and matrix multiplication. Here's how Fibonacci implementation is being referenced as:

  - File Structure Consistency: Across all files (fibonacci.ml, fibonacci_domain.ml, fibonacci_multicore.ml), the core Fibonacci calculation function fib remains consistent. This function recursively computes the Fibonacci sequence up to a given index n.
  - Parallelization Strategy: In fibonacci_domain.ml, the Fibonacci computation is parallelized using a domain-based approach, where the computation is split into multiple domains, and each domain handles a subset of the Fibonacci sequence. In fibonacci_multicore.ml, the parallelization is achieved using the Domainslib library. Here, tasks are asynchronously spawned to calculate Fibonacci numbers, and the results are combined using the Task.await function.
  - Parameterization for Parallelization: Both fibonacci_domain.ml and fibonacci_multicore.ml accept the number of domains/cores as a command-line argument (num_Domains). This parameter allows the user to control the degree of parallelism applied during Fibonacci computation.
  - Resource Management: In fibonacci_multicore.ml, resource management is explicitly handled through the Task.setup_pool and Task.teardown_pool functions, ensuring efficient utilization and release of system resources during parallel computation. This differs from the domain-based approach in fibonacci_domain.ml, which manages resources implicitly through the domain system.

Chapter 4

Challenges Faced

4.1 Dune Setup with OCaml Multicore

Dune is the recommended composable build system for OCaml, renowned for its widespread adoption (nearly 40% OPAM packages utilize Dune) and its superior speed compared to traditional systems, offering incremental builds. This tool facilitates the construction of executables, libraries, and the execution of tests, while handling the intricacies of OCaml compilation seamlessly. A project description is all that is required, and Dune takes care of the rest.

However, we encountered challenges in:

1. Understanding the `dune` file `S-expression` syntax.
2. Structuring files within specialized `bin` and `lib` directories and creating custom libraries.

4.2 Maintaining the correctness of the parallel programs

During the coding process, we confronted the significant challenge of preserving code correctness, particularly when parallelizing the code, especially in scenarios where iterations exhibited interdependencies. To mitigate this issue, we identified the dependencies and segregated the parallel work accordingly. Furthermore, to prevent race conditions, we utilized mutexes to ensure the safe updating of shared variables.

Chapter 5

Working of Domains - The Memory Layout

Domains are heavy-weight entities. Each domain directly maps to an operating system thread. Each domain brings its own runtime state local to the domain. In particular, each domain has its own minor heap area and major heap pools.

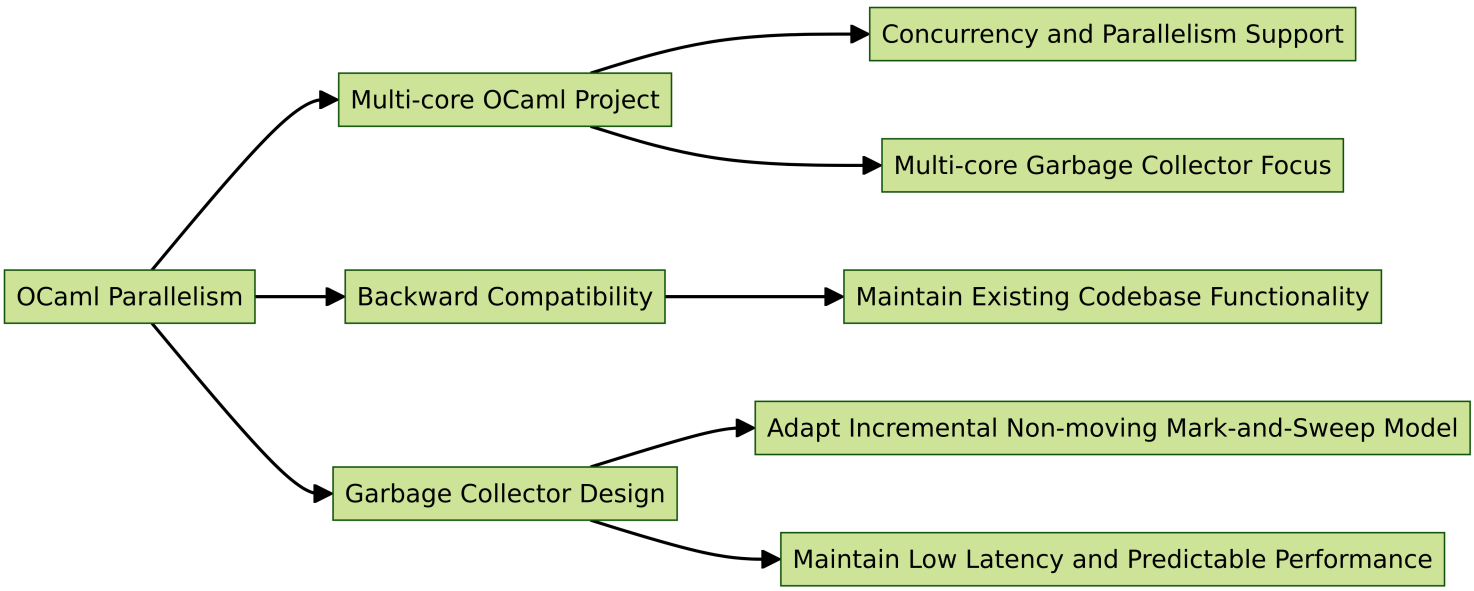


Figure 5.1: Mechanism and features of parallelism in Ocaml

5.1 Memory Management

- **Minor Heap**: Each domain has its own minor heap where short-lived objects are allocated. When a domain's minor heap is full, a "stop-the-world" garbage collection is triggered for all minor heaps across all domains. This means that all execution is paused during this garbage collection process.
  - Per-Domain Allocation: Each domain in OCaml has its own separate minor heap. This design allows multiple threads of execution (domains) to allocate memory independently, reducing contention typically associated with a single heap space in a multithreaded environment.
  - Stop-the-World GC: When the minor heap of any domain becomes full, a garbage collection process is triggered that involves all domains. This process is termed "stop-the-world" because all domains must halt their execution while the garbage collection is in progress. This GC phase cleans up short-lived objects that are no longer in use.

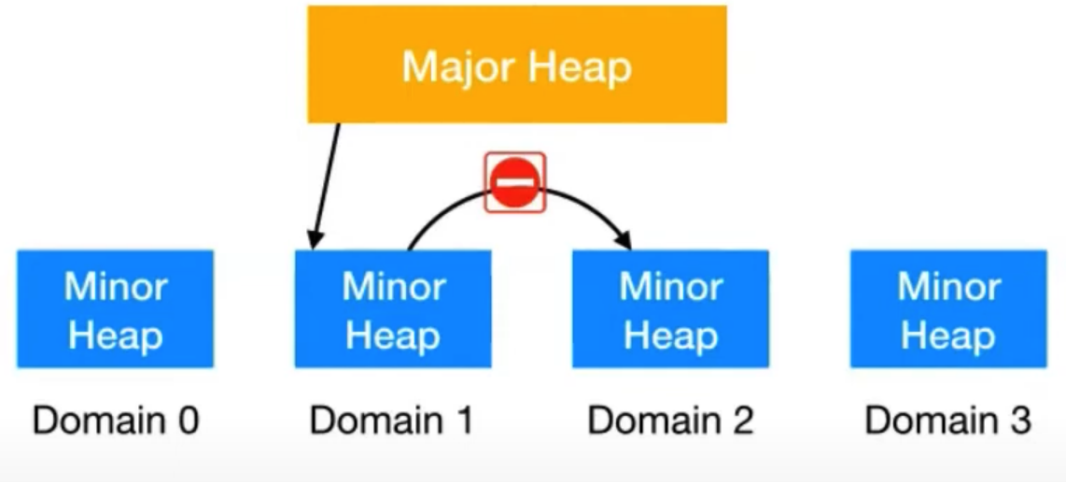


Figure 5.2: Mechanism of Garbage Collector in Ocaml

- **Major Heap**: Objects that survive the minor GC are promoted to the major heap, which is collected less frequently. The major heap collection is concurrent, meaning it doesn't require stopping all domain execution, although it does involve brief pauses that are still "stop-the-world" events but are much shorter and less frequent.
  - Object Promotion: Objects that are not collected during the minor GC (meaning they have survived the garbage collection process)

are promoted to the major heap. The major heap is shared among domains and contains longer-lived objects.

– Concurrent GC with Brief Pauses: Unlike the minor heap GC, the major heap GC is mostly concurrent, allowing domains to continue execution while the garbage collection process is underway. However, it does require brief, occasional "stop-the-world" pauses to maintain synchronization among domains during the GC process.

5.2 Importance of Memory Management in Parallelisation

- **Reduced Contention with Per-Domain Minor Heaps**: Separate minor heaps for each domain minimize memory allocation contention, enhancing independent domain execution and scalability in multi-core environments.
- **Concurrent Garbage Collection in Major Heaps**:Shorter, infrequent pauses during concurrent garbage collection ensure minimal disruption, while efficient management of long-lived objects optimizes overall memory usage.
- **Object Promotion Between Heaps**: Promoting objects based on their longevity from the minor to the major heap optimizes memory resources, aligns with object life cycle needs, and reduces reallocation overheads, enhancing application stability.

Chapter 6

Benchmarking & Performance Analysis

We adopt a benchmarking approach similar to the tutorial, employing tools such as `hyperfine` to measure the performance of our parallel programs. By benchmarking various configurations and analyzing the outcomes, we can evaluate the scalability and efficacy of our parallelization strategies.

`hyperfine` serves as a Linux tool utilized for benchmarking command-line commands. It incorporates features to evaluate the performance of individual commands in parallel.

The command to run for benchmarking (**NOTE**: replace fibonacci with any program you want to run):

```
hyperfine 'dune exec fibonacci' 'dune exec fibonacci parallel'
```

Chapter 7

Libraries/Frameworks used for Parallelisation in Other Languages: C++, Java, Python

7.1 C: Libraries for Parallel Computing

7.1.1 POSIX Threads (Pthreads)

POSIX Threads is a standard for threads implemented in UNIX-like operating systems. It allows for the creation, control, and cleanup of multiple threads in C/C++ applications.

7.1.2 OpenMP (Open Multi-Processing)

OpenMP is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It is used to parallelize loops and sections by inserting pragmas in the code for e.g., by using `#pragma omp parallel for` we can parallelise the part of the code needed.

7.2 Java: Parallelism Frameworks

7.2.1 Executor Framework

Part of the `java.util.concurrent` package, this framework provides a higher-level replacement for working directly with threads. It uses pools of worker threads to execute tasks. Tasks are submitted to the Executor, which manages

a pool of threads to execute these tasks asynchronously. The framework handles thread management, task scheduling, and thread life cycle.

7.2.2 Fork/Join Framework

Introduced in Java 7 , this framework is designed to efficiently execute parallel divide-and-conquer algorithms. It uses a work-stealing algorithm where idle threads can steal tasks from those still busy. Tasks are divided into smaller pieces, and each piece is processed recursively. The results of sub-tasks are then joined to form the final result.

7.3 Python: Libraries for Parallel Computing

7.3.1 multiprocessing

The multiprocessing module allows the programmer to leverage multiple processors on a machine. It uses processes instead of threads due to Python's Global Interpreter Lock (GIL). It is similar to threading but uses processes that are fully independent and sidestep the GIL. It provides an API similar to the threading module.

7.3.2 concurrent.futures

Introduced in Python 3.2, this high-level interface provides a way to manage pools of threads or processes. It uses `ThreadPoolExecutor` and `ProcessPoolExecutor` to manage a pool of threads or processes. The API lets you submit callable objects which are executed in parallel.

Chapter 8

Amdahl's law for parallel programs

Amdahl's Law provides a theoretical framework for estimating the speedup achievable in parallel programs based on the proportion of parallelizable code and the degree of parallelism.

8.1 Theory of Amdahl's Law

Amdahl's Law states that the speedup achievable by parallelizing a program is limited by the fraction of the program that can be parallelized. In essence, even with an infinite number of processors, the speedup is bounded by the sequential portion of the code. The formula for Amdahl's Law is given by:

$$\frac{1}{(1-p)+(\frac{p}{s})}$$

where,

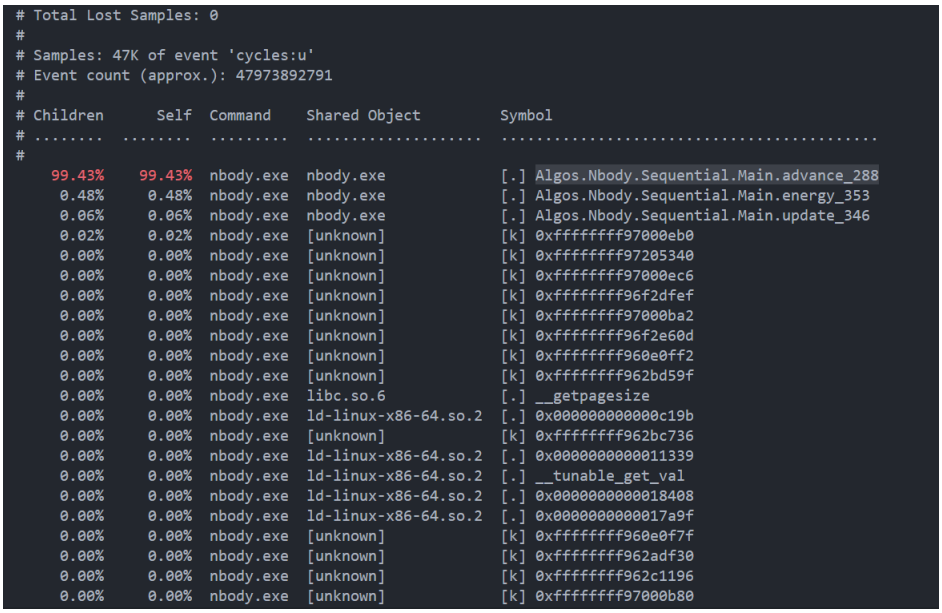
1. p represents the proportion of the code that can be parallelized,
2. s represents the degree of parallelism (number of processors).

8.2 Application to the N-Body Problem

In our performance analysis of the N-body problem, we utilized the `perf` tool to profile the execution and identify performance bottlenecks. `perf` is a Linux profiling tool that provides insights into various aspects of program execution, including CPU usage, memory access patterns, and function call traces.

Using

```
perf record --call-graph=dwarf -- _build/default/bin/nbody.exe
perf report
```



```
# Total Lost Samples: 0
#
# Samples: 47K of event 'cycles:u'
# Event count (approx.): 47973892791
#
# Children      Self  Command    Shared Object        Symbol
# ........  ........  .........  ...................  ........................................
#
    99.43%    99.43%  nbody.exe  nbody.exe            [.] Algos.Nbody.Sequential.Main.advance_288
     0.48%     0.48%  nbody.exe  nbody.exe            [.] Algos.Nbody.Sequential.Main.energy_353
     0.06%     0.06%  nbody.exe  nbody.exe            [.] Algos.Nbody.Sequential.Main.update_346
     0.02%     0.02%  nbody.exe  [unknown]            [k] 0xffffffff97000eb0
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff97205340
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff97000ec6
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff96f2dfef
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff97000ba2
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff96f2e60d
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff960e0ff2
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff962bd59f
     0.00%     0.00%  nbody.exe  libc.so.6            [.] __getpagesize
     0.00%     0.00%  nbody.exe  ld-linux-x86-64.so.2 [.] 0x000000000000c19b
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff962bc736
     0.00%     0.00%  nbody.exe  ld-linux-x86-64.so.2 [.] 0x0000000000011339
     0.00%     0.00%  nbody.exe  ld-linux-x86-64.so.2 [.] __tunable_get_val
     0.00%     0.00%  nbody.exe  ld-linux-x86-64.so.2 [.] 0x0000000000018408
     0.00%     0.00%  nbody.exe  ld-linux-x86-64.so.2 [.] 0x0000000000017a9f
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff960e0f7f
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff962adf30
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff962c1196
     0.00%     0.00%  nbody.exe  [unknown]            [k] 0xffffffff97000b80
```

Figure 8.1: Using `perf` to collect profiling data for N-Body problem

We collected profiling data and generated a report. From the generated report, we observed that the `advance` function accounted for 99.43% of the total execution time, making it a significant bottleneck.

8.3 Amdahl's Law Calculation

Given that the `advance` function is parallelizable and accounts for 99.43% of the code's execution time, we applied **Amdahl's Law** to estimate the maximum achievable speedup.

Let's start with the assumption that we have infinte degree of parallelism (infinite number of cores available), we calculated the theoretical maximum speedup as 175. (here $p = .9943$ and $s \to \infty$):

$$\lim_{s \to \text{inf}} \frac{1}{(1-p) + (\frac{p}{s})} = \frac{1}{(1-p)} = 175$$

However, considering the practical scenario with only 8 cores available, we recalculated the speedup as 7:

$$\frac{1}{(1-p) + (\frac{p}{s})} = \frac{1}{(1-.9943) + \frac{.9943}{8}} = 7$$

8.4 Real-world Considerations

In practical scenarios, the actual speedup achieved may be even lower due to factors such as limited scalability, overheads from system resource contention, and background processes competing for resources. Additionally, Amdahl's Law suggests that for values of $p < 0.89$, the achievable speedup is typically is at max 2 times.

Chapter 9

Limitations of Divide and Conquer Approach in Parallelizing Binary Tree Height Calculation

To achieve parallelizing height of binary tree, we explore the divide and conquer approach, which traditionally involves recursively splitting a problem into smaller sub-problems until they become trivial to solve.

9.1 Recursive Parallel Code Structure

```
type 'a bin_tree =
BTEmpty
| BTNonEmpty of 'a * 'a bin_tree * 'a bin_tree
```

We begin with a recursive function `height`, that computes the height of a binary tree node. If the node is empty (`BTEmpty`), the function returns 0. Otherwise, for a non-empty node (`BTNonEmpty(node, left, right)`), the height is calculated as: `1 + max(height left, height right)`.

9.2 Parallelization Strategy

To exploit parallelism, we create a new function `height_parallel` by modifying the normal recursive function (`height`) to calculate the heights of the left and right subtrees concurrently using `Task.async` and `Task.await` (2.4.5) functions, which trigger parallel computations. We allocate a fixed number of threads from our `Task.pool` (2.4.3) to handle these parallel tasks.

9.3 Deadlock Scenario

Consider a perfect binary tree where each node has two children as shown in image 9.1.

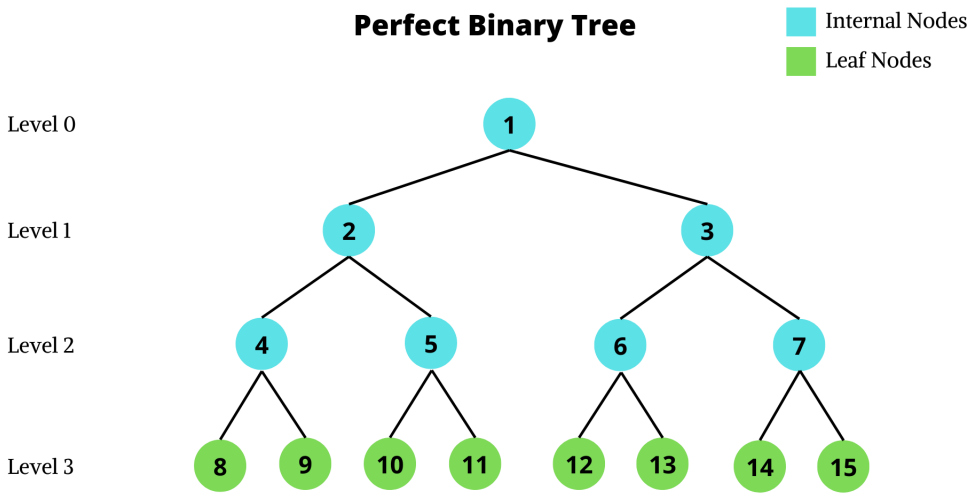


Figure 9.1: Perfect Binary Tree

We call the parallel recursive `height_parallel` function on root of the tree (`node 1`). Then, we call the `height_parallel` for the left(`node 2`) and right subtrees(`node 2`) of the root node in parallel. As we progress down the tree, the number of parallel tasks increases exponentially. Eventually, we encounter a situation where all available threads are consumed by tasks at deeper levels of the tree, leading to a deadlock.

The deadlock arises due to the nature of the binary tree structure and the inherent sequential dependencies in its traversal. While divide and conquer parallelism aims to exploit independent sub-problems, the hierarchical nature of binary trees creates a scenario where upper-level nodes must wait for lower-level nodes to complete their computations. This cascading effect exhausts the available threads, resulting in a deadlock situation where no further progress can be made.

9.4 Solution Proposed

We propose a modified solution that addresses thread management more effectively. It involves restricting the number of parallel computations based on the available threads. Instead of spawning parallel tasks indefinitely, we limit the

parallelization to the number of threads available and any calls to height after that would be done in a sequential manner.

9.4.1 Implementation Details

We introduce a new parameter `threads_left` to the parallel recursive height computation function (`height_parallel`), which represents the number of threads available for parallel tasks. Initially, `threads_left` is set to the maximum number of threads possible (8 in our case).

When `threads_left` ≥ 2, we continue to split the parallel computations for the left and right subtrees. However, once `threads_left` < 2, we revert to sequential computation to avoid deadlock situations.

In below code, this is achieved by decrementing `threads_left` before initiating each parallel computation. If `threads_left` is sufficient for parallelization, we utilize `Task.async` to trigger parallel tasks. Otherwise, we resort to the sequential height computation function.

```
let rec height_parallel (threads_left: int) (pool:
    Task.pool) = function
| BTEmpty -> 0
| BTNonEmpty(_, left, right) ->
    if threads_left >= 2 then
        let threads_left = threads_left - 1 in
        let left_height = Task.async pool (fun _ ->
            height_parallel threads_left pool left) in
        let threads_left = threads_left - 1 in
        let right_height = Task.async pool (fun _ ->
            height_parallel threads_left pool right) in
        1 + max(Task.await pool left_height,
            Task.await pool right_height)

    else
        1 + max(height left, height right)
```

9.4.2 Example Illustration

Once again consider a perfect binary tree where each node has two children as shown in image 9.1.

We initiate the height computation for the root node (`node 1`) with `threads_left` set to the maximum number of threads (8 for our system). As we recursively traverse the tree, we decrement `threads_left` and split parallel computations accordingly.

The process of height computation proceeds as follows (all calls in each step are occuring parallelly):

- Call on node 1 with `threads_left` $= 8$ initially.

- Calls on nodes 2 and 3 with `threads_left` = 7 and `threads_left` = 6, respectively (both parallel).
- Calls on nodes 4, 5, 6, and 7 with decreasing `threads_left` until reaching 2 (all parallel).
- Subsequent calls on remaining nodes with `threads_left` = 1 and `threads_left` = 0, resorting to sequential computation.

9.4.3 Result

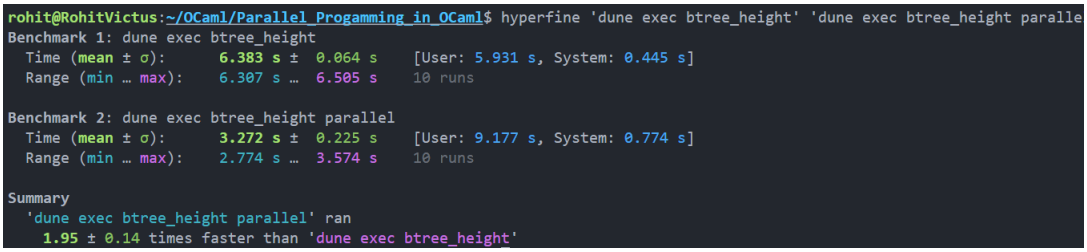


Figure 9.2: Execution time comparison between normal & parallel binary tree height

Chapter 10

Mitigating Shared State Contention Issue in Parallel Programming

The shared state contention issue arises in multicore architectures due to the inherent nature of cache coherency, where writes to memory locations lead to cache invalidations, causing cache misses. This phenomenon occurs because multiple cores in a multicore system share access to a common memory space. When one core writes to a memory location, it invalidates this cache memory location across all cores, leading to cache coherence overhead.

To address this issue, we explore two instances where exploiting cache access patterns enhances the efficiency of parallel algorithms.

10.1 Matrix Multiplication Optimization

10.1.1 Code before

```
for i = 0 to n - 1 do
  for j = 0 to n - 1 do
    for k = 0 to n - 1 do
      result.(i).(j) <- result.(i).(j) + (a.(i).(k) *
          b.(k).(j))
```

10.1.2 Code after

```
for i = 0 to n - 1 do
  for k = 0 to n - 1 do
    for j = 0 to n - 1 do
```

```
4          result.(i).(j) <- result.(i).(j) + (a.(i).(k) *
               b.(k).(j))
```

Reordering the loops in the matrix multiplication algorithm optimizes cache access patterns. By iterating over the innermost loop with the index j, which accesses contiguous memory locations in both matrices a and b, cache efficiency is improved. This mitigates the shared state contention issue, leading to enhanced parallelization performance.

10.1.3 Result

```
rohit@RohitVictus:~/OCaml/Parallel_Progamming_in_OCaml$ hyperfine 'dune exec mat_mul' 'dune exec mat_mul parallel' 'dune exe
c mat_mul write_optimized'
Benchmark 1: dune exec mat_mul
  Time (mean ± σ):      4.503 s ±  0.178 s    [User: 4.479 s, System: 0.021 s]
  Range (min … max):    4.314 s …  4.872 s    10 runs

Benchmark 2: dune exec mat_mul parallel
  Time (mean ± σ):      1.128 s ±  0.050 s    [User: 7.159 s, System: 0.025 s]
  Range (min … max):    1.064 s …  1.240 s    10 runs

Benchmark 3: dune exec mat_mul write_optimized
  Time (mean ± σ):     763.9 ms ±  22.1 ms    [User: 4468.6 ms, System: 31.6 ms]
  Range (min … max):   739.0 ms … 807.8 ms    10 runs

Summary
  'dune exec mat_mul write_optimized' ran
    1.48 ± 0.08 times faster than 'dune exec mat_mul parallel'
    5.90 ± 0.29 times faster than 'dune exec mat_mul'
```

Figure 10.1: Execution time comparison between normal & parallel matrix multiplication

10.2 NBody Problem Optimization

10.2.1 Before

In the NBody problem, we intially had a single array of planets, where each planet type had (x, y, z) coordinates & (v_x, v_y, v_z) velocity vector magnitudes. Our algorithm requires:

- Frequent reads & infrequent writes to coordinates
- Infrequent reads & frequent writes to velocity vector magnitude.

However, having a single array of planets, leads to cache contention for coordinates which can be avoidable since they are read-only for most part of the algorithm.

10.2.2 After

To solve this contention problem, we split the planets array into `planet_pos` and `planet_vec`, based on their access patterns:

- `planet_pos`: Frequently read and infrequently written.
- `planet_vec`: Frequently written and infrequently read.

By separating the planets arrays based on their access patterns, we minimize cache invalidations and improve cache locality. This optimization reduces contention between threads accessing different arrays, thereby enhancing parallel scalability and overall performance.

10.2.3 Result

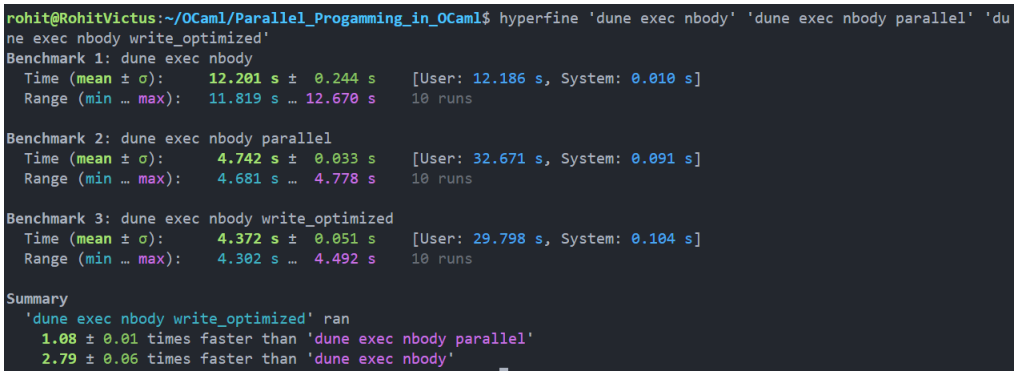


Figure 10.2: Execution time comparison between normal & parallel NBody problem

Chapter 11

Divide and Conquer Algorithms

The main feature of parallelism is that it works well with divide and conquer algorithms, since they work on different parts of the lists/arrays and thus, they can be parallelised efficiently.

11.1 Merge Sort

Mergesort is a sorting algorithm that follows the divide-and-conquer paradigm. It divides the input array into two halves recursively, compares each half separately, and then merges the sorted halves. The merging process involves comparing elements from each half and placing them in the correct order.
The average time complexity of mergesort algorithm is O(nlog n).

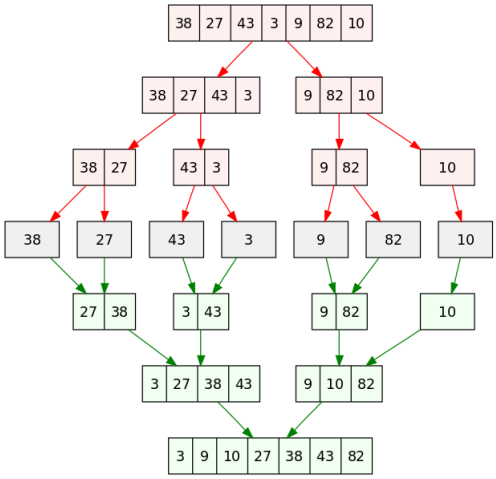


Figure 11.1: Image Source:Wikipedia, explaining mergesort algorithm

11.1.1 Functional

The command to run for benchmarking:

```
hyperfine 'dune exec mergesort_functional' \
'dune exec mergesort_functional parallel'
```

```
rohit@RohitVictus:~/OCaml/Parallel_Progamming_in_OCaml$ hyperfine 'dune exec mergesort_functional' 'dune exec mergesort_functional parallel'
Benchmark 1: dune exec mergesort_functional
  Time (mean ± σ):      1.283 s ±  0.053 s    [User: 1.221 s, System: 0.057 s]
  Range (min … max):    1.218 s …  1.399 s    10 runs

Benchmark 2: dune exec mergesort_functional parallel
  Time (mean ± σ):      1.129 s ±  0.046 s    [User: 3.388 s, System: 0.220 s]
  Range (min … max):    1.056 s …  1.226 s    10 runs

Summary
  'dune exec mergesort_functional parallel' ran
    1.14 ± 0.07 times faster than 'dune exec mergesort_functional'
```

Figure 11.2: Execution time and comparison between the functional merge sort and it's parallel code

The parallel code for computing functional merge sort is over 1.14 times faster than the original code.

11.1.2 Imperative

The command to run for benchmarking:

```
hyperfine 'dune exec mergesort_imperative' \
'dune exec mergesort_imperative parallel'
```

```
keshav-chandak@keshav-chandak-HP-Pavilion-Laptop-14-ec1xxx:~/Desktop/Parallel_Programming/Parallel_Progamming_in_OC
aml$ hyperfine 'dune exec mergesort_imperative' 'dune exec mergesort_imperative parallel'
Benchmark 1: dune exec mergesort_imperative
  Time (mean ± σ):     554.3 ms ±  21.8 ms    [User: 464.0 ms, System: 90.1 ms]
  Range (min … max):   532.0 ms … 602.4 ms    10 runs

Benchmark 2: dune exec mergesort_imperative parallel
  Time (mean ± σ):     397.9 ms ±   5.8 ms    [User: 1125.8 ms, System: 158.9 ms]
  Range (min … max):   390.6 ms … 411.7 ms    10 runs

Summary
  dune exec mergesort_imperative parallel ran
    1.39 ± 0.06 times faster than dune exec mergesort_imperative
```

Figure 11.3: Execution time and comparison between the imperative merge sort and it's parallel code

The parallel code for computing imperative merge sort is over 1.39 times faster than the original code.

11.2 Quick Sort

Quicksort is a sorting algorithm that follows the "divide and conquer" strategy. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. The base case of the recursion is when the sub-array has fewer than two elements.
Time Complexity:

1. Best Case: O(n log n) - Occurs when the pivot is consistently chosen such that the partition splits the array into two roughly equal halves.
2. Average Case: O(n log n) - Similar to the best case, but the pivot selection is random or "median-of-three".
3. Worst Case: O(n^2) - Occurs when the pivot is consistently chosen as the smallest or largest element, leading to highly unbalanced partitions.

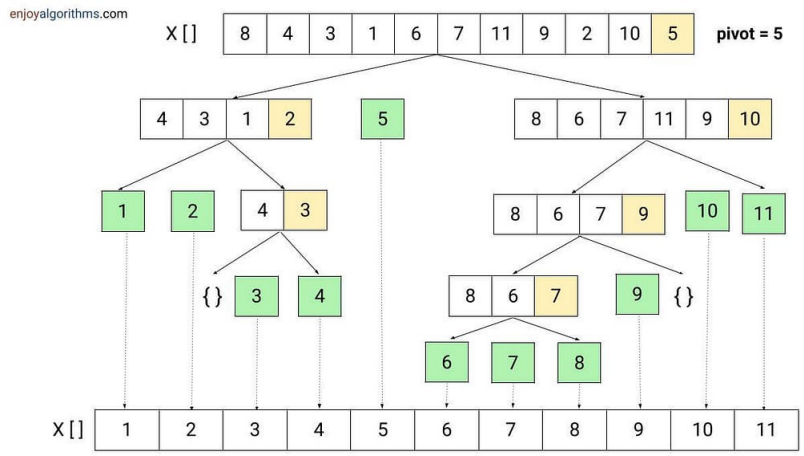


Figure 11.4: Image Source:enjoyalgorithms.com, explaining quicksort algorithm

11.2.1 Functional

```
hyperfine 'dune exec quicksort_functional' \
'dune exec quicksort_functional parallel'
```

```
keshav-chandak@keshav-chandak-HP-Pavilion-Laptop-14-ec1xxx:~/Desktop/Parallel_Programming/Parallel_Progamming_in_OCaml$ hyperfine 'dune exec quicksort_functional' 'dune exec quicksort_functional parallel'
Benchmark 1: dune exec quicksort_functional
  Time (mean ± σ):      2.100 s ±  0.374 s    [User: 1.904 s, System: 0.196 s]
  Range (min … max):    1.764 s …  2.711 s    10 runs

Benchmark 2: dune exec quicksort_functional parallel
  Time (mean ± σ):      1.677 s ±  0.034 s    [User: 5.028 s, System: 0.570 s]
  Range (min … max):    1.639 s …  1.742 s    10 runs

Summary
  dune exec quicksort_functional parallel ran
    1.25 ± 0.22 times faster than dune exec quicksort_functional
```

Figure 11.5: Execution time and comparison between the functional quick sort and it's parallel code

The parallel code for computing functional quick sort is over 1.25 times faster than the original code.

11.2.2 Imperative

The command to run for benchmarking:

```
hyperfine 'dune exec quicksort_imperative' \
'dune exec quicksort_imperative parallel'
```

```
keshav-chandak@keshav-chandak-HP-Pavilion-Laptop-14-ec1xxx:~/Desktop/Parallel_Programming/Parallel_Progamming_in_OCaml$ hyperfine 'dune exec quicksort_imperative' 'dune exec quicksort_imperative parallel'
Benchmark 1: dune exec quicksort_imperative
  Time (mean ± σ):     931.3 ms ±  22.9 ms    [User: 803.4 ms, System: 127.3 ms]
  Range (min … max):   902.1 ms … 970.2 ms    10 runs

Benchmark 2: dune exec quicksort_imperative parallel
  Time (mean ± σ):     538.7 ms ±  54.6 ms    [User: 1476.4 ms, System: 115.9 ms]
  Range (min … max):   501.1 ms … 650.3 ms    10 runs

Summary
  dune exec quicksort_imperative parallel ran
    1.73 ± 0.18 times faster than dune exec quicksort_imperative
```

Figure 11.6: Execution time and comparison between the imperative quick sort and it's parallel code

The parallel code for computing imperative quick sort is over 1.73 times faster than the original code.

11.3 Fast Fourier Transform

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT), which is essential in signal processing, data analysis, and many other fields.

11.3.1 FFT Components

The FFT algorithm decomposes a sequence of values into components of different frequencies, efficiently transforming the time-domain data into a frequency-domain representation. The two crucial aspects of the FFT algorithm implemented here are the twiddle factors and the butterfly merging step.

Twiddle Factors

Twiddle factors are complex exponential terms used in the FFT to reduce the DFT's computational complexity from $O(n^2)$ to $O(n \log n)$. The twiddle factor for the FFT is defined as follows:

$$e = e^{-2\pi i \frac{k}{n}}$$

Where:

- i is the imaginary unit.
- k is the current index within the butterfly merging process.

- n is the total number of points in the FFT.

These factors are crucial for the element-wise multiplication in the butterfly merging step, effectively rotating the complex numbers to align the transform for linear combination.

Butterfly Merging

Butterfly merging is the key operation in the FFT that combines pairs of points into frequency bins efficiently. It uses the twiddle factors to combine the results from smaller DFTs into the full DFT. The operation involves simple additions and subtractions but is applied recursively or iteratively depending on the FFT's implementation strategy.

11.3.2 Parallel FFT Implementation

The parallel implementation enhances the FFT by utilizing multiple processing units to handle different segments of the FFT simultaneously.

```
Function: Parallel FFT
Strategy: Use async and await tasks to handle even and odd indexed
          FFTs in parallel, and parallel_for to distribute butterfly
          merging across CPU cores
```

This method significantly reduces the computational time by performing multiple FFT computations concurrently, and efficiently merging them in parallel using the twiddle factors.

11.3.3 Benchmarking and Analysis

```
rohit@RohitVictus:~/OCaml/tmp$ hyperfine 'dune exec fft' 'dune exec fft parallel'
Benchmark 1: dune exec fft
  Time (mean ± σ):      1.258 s ±  0.184 s    [User: 1.195 s, System: 0.056 s]
  Range (min … max):    1.094 s …  1.606 s    10 runs

Benchmark 2: dune exec fft parallel
  Time (mean ± σ):     836.1 ms ±  33.0 ms    [User: 2844.3 ms, System: 106.3 ms]
  Range (min … max):   781.7 ms … 891.9 ms    10 runs

Summary
  'dune exec fft parallel' ran
    1.50 ± 0.23 times faster than 'dune exec fft'
rohit@RohitVictus:~/OCaml/tmp$
```

Figure 11.7: Comparison between the Normal and Parallel FFT

Chapter 12

Parallelisation of Linear Algebra Algorithms

12.1 QR Decomposition using Gram Schmidt

The Gram-Schmidt process (or procedure) is a sequence of operations that allow us to transform a set of linearly independent vectors into a set of orthonormal vectors that span the same space spanned by the original set.

The Gram-Schmidt orthogonalization process transforms a set of linearly independent vectors into a set of orthogonal vectors. Given a matrix A with columns $a_1, a_2, ..., a_n$, the process to obtain the QR decomposition involves:

1. Orthogonalization:
   - Compute the first orthogonal vector $\mathbf{q}_1$ by normalizing $\mathbf{a}_1$.
   - For each subsequent vector $\mathbf{a}_i$:
     - Subtract from $\mathbf{a}_i$ its projection onto each previously computed orthogonal vector $\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_{i-1}$.
     - Normalize the result to get $\mathbf{q}_i$.
2. Projection Calculation:
   - The projection of $\mathbf{a}_i$ onto $\mathbf{q}_j$ is computed.
3. Normalization:
   - Each orthogonal vector $\mathbf{q}_i$ is normalized using the norm of the vector obtained after subtracting all previous projections.

12.1.1 Sequential Nature of QR Decomposition

- Dependency on Previous Results: Each vector $\mathbf{q}_i$ depends on all previous vectors $\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_{i-1}$. This means you can't compute $\mathbf{q}_3$ without first

computing $\mathbf{q}_1$ and $\mathbf{q}_2$, and so forth. The orthogonalization of each vector is dependent on the results of all prior computations.

- Accumulative Computation: The calculation of each orthogonal vector requires the complete set of all previously calculated orthogonal vectors. This sequential dependency forms a bottleneck that inherently limits parallel execution because each step must wait for the results of all previous steps.
- Workload Distribution: Even with parallel computation of inner products and scalar multiplications, the workload in early steps (computing early $\mathbf{q}_i$ vectors) is significantly less than in later steps (where each $\mathbf{q}_i$ requires orthogonalization against many previous vectors). This imbalance can lead to inefficient utilization of computational resources of parallelization.

12.1.2 Parallelizable Components of QR Decomposition

- Inner Products Calculation: Computing the inner product of $\mathbf{a}_i$ with each previously computed orthogonal vector $\mathbf{q}_j$ can be parallelized. Each inner product calculation is independent and can be performed concurrently.
- Scalar Multiplication and Subtraction: Subtracting the projection of $\mathbf{a}_i$ onto each $\mathbf{q}_j$ and normalizing the result can also be parallelized. Each subtraction and normalization operation is independent and can be executed concurrently across multiple threads or processes.

12.1.3 Benchmarking and Analysis

```
rohit@RohitVictus:~/OCaml/tmp$ hyperfine 'dune exec qr_decomp_gram' 'dune exec qr_decomp_gram parallel'
Benchmark 1: dune exec qr_decomp_gram
  Time (mean ± σ):     14.831 s ±  0.772 s    [User: 14.783 s, System: 0.037 s]
  Range (min … max):   14.324 s … 16.881 s    10 runs

Benchmark 2: dune exec qr_decomp_gram parallel
  Time (mean ± σ):      5.704 s ±  0.426 s    [User: 35.242 s, System: 0.926 s]
  Range (min … max):    4.982 s …  6.489 s    10 runs

Summary
  'dune exec qr_decomp_gram parallel' ran
    2.60 ± 0.24 times faster than 'dune exec qr_decomp_gram'
rohit@RohitVictus:~/OCaml/tmp$
```

Figure 12.1: Comparison between the Normal and Parallel QR decomposition

12.2 Singular Value Decomposition

Singular Value Decomposition (SVD) is a factorization of matrix that generalizes the eigen decomposition of a square normal matrix to any $m \times n$ matrix via

an extension of the polar decomposition. SVD is particularly useful in signal processing and statistics where it serves as the foundation for principal component analysis, among other things. Given a matrix A with dimensions $m \times n$, SVD decomposes A into three matrices:

$$A = U\Sigma V^T$$

Where:

- U is an $m \times m$ orthogonal matrix whose columns are the left singular vectors of A.
- Σ is an $m \times n$ diagonal matrix with non-negative real numbers on the diagonal known as the singular values of A.
- V^T (the transpose of V) is an $n \times n$ orthogonal matrix whose columns are the right singular vectors of A.

The process to obtain the SVD involves several computational steps, crucially involving matrix transformations and orthogonalization:

1. **Matrix Transformation:**
   - Compute the matrix $A^T A$. This square $n \times n$ matrix will be used to find the matrix V of right singular vectors.
   - Similarly, computing AA^T is essential for determining the matrix U of left singular vectors if needed.
2. **Eigenvalue Decomposition:**
   - Apply an eigenvalue decomposition to $A^T A$ to find V and its eigenvalues, which are the squares of the singular values in Σ.
   - Optionally, for U, apply eigenvalue decomposition to AA^T.
3. **Construction of Σ:**
   - Extract the square roots of the eigenvalues of $A^T A$ (which are the same as those of AA^T) to form the singular values, which are placed on the diagonal of Σ.
4. **Orthogonalization of Columns:**
   - For U and V, ensure that all columns are orthogonal and normalized. This can be achieved using the Gram-Schmidt process or by ensuring the eigenvectors from the decomposition are properly orthogonalized.

12.2.1 Parallelizable Components of SVD

- **Matrix Multiplication** - Matrix multiplication is fundamental in the calculation of SVD, specifically for deriving A^TA and AA^T. Parallelization of matrix multiplication is achieved using Domainslib, which allows the distribution of row computations across multiple CPU cores.
- **Gram-Schmidt Process** - The Gram-Schmidt orthogonalization process is used for generating the orthogonal matrices U and V in the SVD. By parallelizing the Gram-Schmidt process, each vector's orthogonalization can be independently processed across different domains.

12.2.2 Benchmarking and Analysis

```
rohit@RohitVictus:~/OCaml/tmp$ hyperfine 'dune exec svd' 'dune exec svd parallel'
Benchmark 1: dune exec svd
  Time (mean ± σ):      2.027 s ±  0.043 s    [User: 2.004 s, System: 0.018 s]
  Range (min … max):    1.980 s …  2.135 s    10 runs

Benchmark 2: dune exec svd parallel
  Time (mean ± σ):     970.5 ms ±  50.8 ms    [User: 4071.4 ms, System: 295.6 ms]
  Range (min … max):   899.0 ms … 1039.1 ms    10 runs

Summary
  'dune exec svd parallel' ran
    2.09 ± 0.12 times faster than 'dune exec svd'
```

Figure 12.2: Comparison between the Normal and Parallel SVD

12.3 LU Decomposition

LU Decomposition is an algorithm where a square matrix is used to formulate the upper triangular matrix U and lower triangular matrix L, which can be further used for solving linear equations and finding determinant of the original matrix with high efficiency, as one half of the elements in both the matrices L and U are 0s .

When using LU Decomposition, to form L and U, the RREF has to be calculated sequentially, and pivot variables has to be stored denoting the largest element in the column.

The formula for LU Decomposition is:-

$$det(A) = det(LU)$$

where L=lower triangular matrix, U=upper triangular matrix and A is the square matrix

LU decomposition with pivoting isn't parallelizable due to the following reasons:

1. Data Dependencies: The elimination step has dependencies between calculations in a single row update. We can't update all elements in a row in parallel without proper synchronization.

2. Pivoting: The process of finding the pivot element (largest element in the column) for partial pivoting requires iterating through the entire column, and if we try to parallelise this way, there is no speedup and no use of parallelisation anyway.

Chapter 13

Why BFS/DFS is not possible

13.1 Race Condition in Visited array

One significant obstacle to parallelizing BFS and DFS algorithms is the frequent updating of the visited array. In a parallel execution environment, multiple threads may concurrently access and modify elements of the visited array, leading to race conditions and inconsistent results. Resolving these race conditions while maintaining correctness and efficiency is quite complex.

13.2 Complexity of Parallelization

The nature of BFS and DFS algorithms, which involve traversing a graph in a sequential manner, complicates their parallelization. While BFS explores nodes level by level, DFS delves deeply into the graph's structure, both of which are inherently sequential processes. Parallelizing these sequential operations without introducing conflicts or synchronization overheads is challenging.

13.3 Disconnected Components Requirement

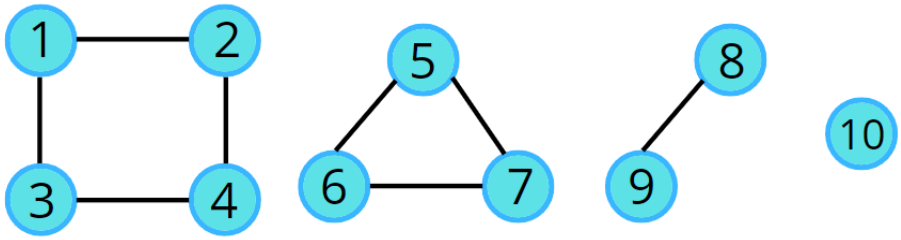


Figure 13.1: Disconnected components in a graph

In the image 13.1, we have 4 sub-graphs (G_1={1,2,3,4}, G_2={5,6,7}, G_3={8,9}, G_4={10}). Assuming each of the sub-graph is very dense we can parallelize BFS for each disconnected component (each sub-graph), as each component (sub-graph) can be explored independently in parallel. However, achieving this requires an initial BFS/DFS traversal to identify disconnected components, which presents a contradictory situation - requiring BFS to parallelize BFS.

Chapter 14

Work Distribution

- Sunny Kaushik: Implemented key linear algebra operations - QR decomposition and Singular Value Decomposition using higher order functions and functional programming. Along with optimising these matrix operations using parallelisation, I've be parallelised the Fast fourier transform using the divide and conquer approach. Compared different frameworks and libraries used in different languages like Python, C, C++ and Java. Analysed the memory layout and working of the domains in case of minor heap and major heap, and the working of stop-the-world garbage collector.
- Rohit Shah: Setup entire project using dune, created `dune` configuration files to create libraries to be imported and executables to be run using one command. Experimented different strategies: Domains, Pool, Async/Await from `OCaml Multicore` to find out best strategy for specific use-cases (recursion, iteration, parellization, etc.). Implemented parallel Matrix multiplication, Fibonacci, Binary tree height calculation, Bareiss algorithm for determinant, Graph algos (BFS & DFS), and N-body problem. Also did study on topics like **Amdahl's Law**, **Shared state cache contention problem**, **Deadlock scenarios in recursive divide & conquer algorithms**.
- Keshav Chandak: Parallelized `merge sort` and `quick sort` using functional and imperative languages paradigms seperately, and compared their execution time. I also tried to parallelise the `LU Decomposition` algorithm, and gave concrete reasons as to why it was **not parallelisable**.

Chapter 15

Future Work and Discussion Topic

15.1 Q) Can parallelization be automated ?

Automation of parallelization problem is undecidable. Research has been going on from many years regarding parallelizing programs. It may seem that we can check recursions and iterative processes for parallelization tasks and check for their correctness iteratively, but it may not guarantee correct parallelisation.

15.2 Q)What are the advantages and disadvantages of functional programming over imperative programming ?

The advantages of imperative programming is it's speed, and robustness in terms of understanding program complexity. However, the imperative programming has mutables, which might hamper parallelisation constructs .
Functional programming constructs have much utility in terms of parallelization since they do not allow mutability, and also allows us to implement function patches that specifically deal with parallel tasks. However, functional programming takes up too much time due to runtime overheads of lists, and also decrease program readability.