CS3103 Project Report: Parallel Matrix Operation
Ma Kam To Vincent
57140216
Group 31

Project Report: Parallel Matrix Operation

Introduction

This project focuses on parallelising matrix operations, specifically matrix multiplication, addition, and subtraction, based on an input expression. The aim is to improve the performance of these operations by leveraging parallel computing techniques.

Techniques for Parallelisation

1. Parallel Matrix Multiplication

Matrix multiplication is traditionally a highly parallelisable operation. Initially, an attempt was made to parallelise the matrix multiplication by dividing the result matrix into blocks and distributing the computation across different processing units. However, due to the dependencies between elements in the result matrix, achieving effective parallelisation at this level proved challenging. Dependencies arise because each element in the result matrix depends on the sum of products of corresponding elements from the input matrices.

2. Parallel Matrix Addition and Subtraction

Matrix addition and subtraction, on the other hand, were more amenable to parallelisation. Each element in the result matrix is independent of others, allowing for straightforward parallelisation. The proposed technique involved dividing the matrices into blocks and parallelising the computation of each block. Each block addition or subtraction could be performed concurrently, improving the overall efficiency of these operations.

3. Parallel Expression Evaluation

To parallelise the expression evaluation, the initial attempt involved trying to exploit parallelism using conditional statements for different operators. However, due to the nature of dependencies between sub-expressions, this approach did not yield the desired level of parallelism.

The successful parallelisation strategy involved using a stack-based approach to evaluate the expression from left to right. The expression was parsed, and matrices were operated on based on the operator precedence. The stack-based approach facilitated the management of matrices and operators, allowing for parallel evaluation of sub-expressions. When a multiplication operation was encountered, matrices involved in the multiplication were divided, and each part was processed concurrently. This method effectively utilised parallel computing resources to enhance the overall performance of matrix operations.

Stack-Based Approach

The stack-based approach involved maintaining two stacks: one for matrices and another for operators. As the expression was parsed from left to right, matrices were pushed onto the matrix stack, and operators were pushed onto the operator stack. When an operator with higher precedence was encountered, the corresponding matrices were popped from the matrix stack, the operation was performed, and the result was pushed back onto the matrix stack. This process continued until the entire expression was evaluated, and the final result remained on top of the matrix stack.

The stack-based approach facilitated a natural flow of parallelisation, allowing for the concurrent processing of sub-expressions based on operator precedence. This technique improved the efficiency of matrix operations in the overall expression evaluation.

- **Initialisation of Stacks and Variables:**
  - Two stacks are used: `matrixStack` to store matrices, and `operatorStack` to store operators.
  - `matrixTop` and `operatorTop` represent the top indices of the respective stacks.
  - `matrixIndex` keeps track of the current matrix index being processed.
  - `result` will store the final result of the expression.

```
Matrix matrixStack[MAX_EXPRESSION_LENGTH];
char operatorStack[MAX_EXPRESSION_LENGTH];
int matrixTop = -1;  // Stack top index for matrices
int operatorTop = -1; // Stack top index for operators
int matrixIndex = 0;
Matrix result;
```

- **Expression Parsing and Stack Operations:**

- The code iterates through each character in the expression.
- If the character is an alphabet (matrix identifier), it pushes the corresponding matrix onto the `matrixStack`.
- If the character is an operator `(+, -, *, /)`, it compares its precedence with the top operator on the `operatorStack`. If the top operator has higher or equal precedence, it pops the top operator, pops two matrices from `matrixStack`, performs the operation, and pushes the result back onto `matrixStack`. This process continues until the operator stack is empty or the top operator has lower precedence.
- The current operator is then pushed onto the `operatorStack`.

```
for (int i = 0; expression[i] ≠ '\0'; i++) {
    if (isalpha(expression[i])) {
        // … (Push matrix onto matrixStack)
    } else if (expression[i] == '+' || expression[i] == '-' ||
expression[i] == '*' || expression[i] == '/') {
        // … (Handle operators and perform operations)
    }
}
```

- **Perform Remaining Operations:**
  - After parsing the entire expression, any remaining operations in the stacks are performed. The process is similar to the above, ensuring that all operations are completed.

```
while (operatorTop ≥ 0) {
    char op = operatorStack[operatorTop--];
    if (matrixTop ≥ 1) {
        // … (Handle remaining operators and perform operations)
    }
}
```

- **Retrieve and Print the Result:**
  - The final result is on top of the `matrixStack` after processing the entire expression.
  - The result is assigned to the `result` variable and printed using the `printMatrix` function.

```
result = matrixStack[matrixTop];
printMatrix(&result);
```

- **Free Allocated Memory:**
  - The memory allocated for the result matrix is freed to prevent memory leaks.

```
freeMatrix(&result);
```

This stack-based approach ensures that matrix operations are performed based on operator precedence, and the final result is obtained and printed correctly. It provides a clear and organized way to handle the complexities of parsing and evaluating mathematical expressions involving matrices.

Learning Experience with Vi and Shell Commands

Vi for Code Editing

In this project, I employed the Vi text editor while working on a Linux server accessed via SSH, utilizing the gateway functionality. Vi, renowned for its efficiency in text editing, proved to be an excellent choice for coding in a remote environment. The simplicity of Vi's modal interface allowed me to seamlessly navigate and edit code directly within the terminal, enhancing productivity. SSH facilitated secure access to the Linux server gateway, and Vi's features, such as syntax highlighting and powerful search and replace functionalities, streamlined the coding process. This approach not only demonstrated proficiency in using Vi for coding but also showcased the effectiveness of leveraging SSH to work on remote servers, providing a practical and efficient development environment.

Shell Commands for File Analysis

Understanding and utilising shell commands proved valuable in studying input files and expected output formats. Key insights gained include:

- **File Inspection**: Commands like `cat`, `head`, and `tail` helped examine the content of files quickly. For instance, running `cat filename` displayed the entire content of a file.
- **Pipelining**: Combining commands using pipes (`|`) facilitated complex operations. For example, `cat filename | grep keyword` allowed searching for specific patterns within a file.
- **File Redirection**: Using `>` and `<` for output and input redirection simplified the handling of file content. For example, `command > output.txt` redirected the output of a command to a file.

- **Command-Line Arguments**: Understanding how to pass command-line arguments to programs through the terminal facilitated efficient interaction with programs.
- **File Permissions**: Commands like `chmod` helped manage file permissions, ensuring that the necessary access rights were granted for file execution.

This learning experience enhanced proficiency in the Unix command line, providing a solid foundation for file manipulation, analysis, and interaction with the development environment. The combination of Vi and shell commands streamlined the coding and testing processes, contributing to a more efficient and organised workflow.

Overcoming Challenges

Parallelising Matrix Operations

- **Challenge**: Determining efficient partition schemes for matrix operations to fully utilise computing resources, considering variations in matrix size and density.
  - **Solution**: Explored two main partitioning ideas:
    - **Row-wise/Column-wise 1D Partition**: Dividing the matrices into segments along rows or columns for parallel computation. This approach works well for large matrices with a relatively uniform distribution of elements.
    - **2D Partition**: Splitting the matrices into blocks or tiles, enables parallel computation at a more granular level. This approach is effective for matrices with irregular distributions, enabling better load balancing.
  - **Adaptation**: Depending on the characteristics of the input matrices (size, density), dynamically select the appropriate partitioning scheme. For larger, denser matrices, 2D partitioning may be more beneficial, while 1D partitioning might be suitable for smaller or less dense matrices.

Splitting the Expression for Parallel Calculation

- **Challenge**: Develop a balanced splitting scheme for the expression to ensure equal distribution of computations among threads.
- **Solution**: Implemented a stack-based approach to parse and evaluate the expression from left to right. Key considerations include:
  - **Operator Precedence**: Parsing the expression based on operator precedence, allowing for the parallel execution of higher-precedence operations before lower-precedence ones.

- **Dynamic Workload Assignment**: Dynamically balancing the workload among threads by efficiently splitting the expression. When encountering a multiplication operation, the matrices involved are divided and assigned to separate threads.
- **Thread Count Determination**: Utilised Linux interfaces like `get_nprocs()` and `get_nprocs_conf()` to determine the number of available threads. This information ensures that the workload is appropriately distributed among the available computing resources.
- **Refinement**: Regularly refined the splitting scheme based on testing and profiling. Ensured that the workload distribution is optimal, preventing thread underutilisation or overload.

Linux Thread Count Determination

- **Challenge**: Determining the number of threads dynamically on Linux for optimal workload distribution.
- **Solution**: Employed Linux interfaces to obtain information about the available processors:
  - `get_nprocs()`: Returns the number of online processors.
  - `get_nprocs_conf()`: Returns the number of configured processors.
- **Usage**: These interfaces were used to determine the optimal number of threads for parallel execution. The obtained information was crucial for workload balancing and resource utilisation.
- **Consideration**: Regularly referred to the man pages for these interfaces to ensure accurate usage and interpretation of the results.

These solutions collectively addressed the challenges, ensuring an effective parallelisation strategy for matrix operations and a balanced distribution of computations within the expression evaluation. The adaptability of partitioning schemes and dynamic workload assignment contributed to the successful parallelisation of the code.

Functions Overview

1. `getPrecedence`

- **Description**: This function plays a crucial role in determining the precedence of an operator. The operator precedence is essential for correctly parsing and evaluating expressions with multiple operators, ensuring that operations are performed in the correct order.

- **Parameters**: `char operation` - the operator for which the precedence needs to be determined.
- **Returns**: The function returns an integer value representing the precedence of the given operator. Higher values indicate higher precedence.

2. Matrix Operations

- **`multiply`**: Performs matrix multiplication.
  - **Parameters**: `Matrix mat1, Matrix mat2` - input matrices to be multiplied.
  - **Returns**: The function returns a new matrix that is the result of multiplying the input matrices.
- **`add`**: Performs matrix addition.
  - **Parameters**: `Matrix mat1, Matrix mat2` - input matrices to be added.
  - **Returns**: The function returns a new matrix that is the result of adding the input matrices.
- **`subtract`**: Performs matrix subtraction.
  - **Parameters**: `Matrix mat1, Matrix mat2` - input matrices to be subtracted.
  - **Returns**: The function returns a new matrix that is the result of subtracting the second matrix from the first.
- **`performMatrixOperation`**: Performs the specified matrix operation based on the operator.
  - **Parameters**: `char operation, Matrix mat1, Matrix mat2` - the operator and the input matrices for the operation.
  - **Returns**: The function returns a new matrix resulting from the specified matrix operation.

3. Matrix Manipulation

- **`printMatrix`**: Prints the elements of a matrix.
  - **Parameters**: `Matrix* matrix` - a pointer to the matrix to be printed.
  - **Output**: The function prints the dimensions and elements of the given matrix.
- **`freeMatrix`**: Frees the memory allocated for a matrix.
  - **Parameters**: `Matrix* matrix` - a pointer to the matrix to be freed.
  - **Action**: The function releases the dynamically allocated memory for the matrix, preventing memory leaks.

4. Input/Output Operations

- **`readMatrix`**: Reads a matrix from standard input.

    - **Parameters**: `Matrix* matrix` - a pointer to the matrix structure to store the input.
    - **Returns**: The function returns 0 for successful execution and -1 in case of failure.
    - **Action**: The function dynamically allocates memory for the matrix, reads its dimensions and elements, and handles errors in the input process.
- `performOperationsWithPrecedence`: Parses and evaluates the expression with operator precedence.
    - **Parameters**: `char* expression, Matrix* matrices, int matrixCount` - the input expression and matrices.
    - **Action**: The function utilises a stack-based approach to parse and evaluate the expression, leveraging operator precedence. It orchestrates the parallelised matrix operations based on the parsed expression.

5. Main Function

- **Description**: The main function of the program is responsible for coordinating the execution. It reads the input expression, and matrices, and orchestrates the overall parallelised matrix operations.
- **Action**: Reads the input expression and matrices, calls the `performOperationsWithPrecedence` function to evaluate the expression, and frees allocated memory for matrices to ensure proper resource management.

Conclusion

In this project, I designed and implemented a program in C that parses and evaluates mathematical expressions involving matrix operations, focusing on addressing challenges related to parallelisation. The code features modular functions for fundamental matrix operations, dynamic memory allocation, and error handling. It incorporates a stack-based approach for expression evaluation, ensuring the correct order of operations based on operator precedence. The program reads input expressions and matrices from standard input, emphasising user-friendly input handling. To address memory management, a function for freeing allocated matrix memory is included. Challenges such as parallelising matrix operations are acknowledged, with the current code serving as a foundational framework for potential future parallelisation. Additionally, the use of Vi for coding and shell commands for studying input files reflects a practical and hands-on approach to software development. Overall, the project successfully establishes a structured and extensible foundation for matrix operations, with considerations for optimisation and parallelisation in subsequent iterations.