SCHOOL OF COMPUTER SCIENCES, UNIVERSITI SAINS MALAYSIA

11800 USM, PENANG, MALAYSIA

SEMESTER 1 2023/2024

# CPT316 PROGRAMMING LANGUAGE IMPLEMENTATION AND PARADIGMS

## ASSIGNMENT 2 REPORT

GROUP 18

| NAME | MATRIC NO. |
|---|---|
| MOHAMAD NAZMI BIN HASHIM | 158616 |
| MIOR MUHAMMAD IRFAN BIN MIOR LATFEE | 158450 |
| OOI JING MIN | 158768 |
| MUNIRAH BINTI SHAMSUDIN | 157518 |
| SHONEERA SIMIN | 159160 |

LECTURER: DR NIBRAS ABDULLAH AHMED FAQERA

SUBMISSION DATE: 28 DECEMBER 2023

# Table of Contents

# ABSTRACT

This report examines the implementations of imperative and parallel programming paradigms in C, OpenMP, MPI, and CUDA, comparing them in the context of search algorithms. This report aims to identify the advantages and disadvantages of each parallel programming paradigm and gain important insights into the real-world applications of parallel programming by evaluating the performance and resource usage of C, OpenMP, MPI, and CUDA. Three search algorithms, mainly binary search, linear search, and depth-first search algorithms are reviewed in this report. The linear search algorithm is chosen for thorough implementation based on a comparison of their imperative and parallel techniques because of its ease of use and practical applicability. This report also describes how the selected algorithm is implemented using various programming models and paradigms. The implementation of the linear search in C, OpenMP, MPI, and CUDA is covered in detail, with a focus on the unique aspects and factors of each programming paradigm. The report also offers forecasts, observations, and lessons learned from the implementations regarding the expected results.

# TASK DISTRIBUTION

| Name | Task |
|---|---|
| Mior Muhammad Irfan Bin Mior Latfee | Code:<br>- Implementation of C<br>Report:<br>- Development tools for C<br>- Implementation and Pseudocode for C<br>- Discussion |
| Ooi Jing Min | Code:<br>- Implementation of OpenMP<br>Report:<br>- Development tools for OpenMP<br>- Implementation and Pseudocode for OpenMP<br>- Discussion |
| Mohamad Nazmi Bin Hashim | Code:<br>- Implementation of MPI<br>Report:<br>- Development tools for MPI<br>- Implementation and Pseudocode for MPI<br>- Discussion |
| Munirah Binti Shamsudin | Code:<br>- Implementation of CUDA<br>Report:<br>- Development tools for CUDA<br>- Implementation and Pseudocode for CUDA<br>- Discussion |
| Shoneera Simin | Report:<br>- Abstract and Introduction<br>- Review of search algorithms<br>- Justification for choosing linear search<br>- Conclusion |

# INTRODUCTION

Search algorithms are significant in computing, dedicated to finding optimal solutions or navigating vast datasets efficiently. Search algorithms explore a set of potential solutions, also known as a search space, to discover the best or most optimum solution to a problem. These algorithms are employed by search agents, intelligent systems that interact with their environment to achieve their goals. It serves a critical role in various domains, from online search engines to scientific data analysis, by swiftly and accurately retrieving specific information. By filtering and analysing numerous possibilities, search algorithms enable agents to generate plans and sequences of actions to identify the most suitable solution within a defined set of constraints.

Java and Python are versatile and widely adopted tools in imperative programming languages. Both languages offer robust support for imperative programming paradigms, emphasizing sequential execution and step-by-step instructions for solving computational problems. Java excels in building robust, scalable applications, offering strict compile-time checks and a well-defined structure that ensures reliability in larger codebases. On the other hand, Python's concise syntax and extensive library support have placed it as a favoured tool for imperative programming, allowing developers to express complex ideas with minimal code.

C is a powerful, low-level language frequently utilized in scientific computers, embedded devices, and system programming. C is renowned for being straightforward and adaptable. It offers tools for direct memory and hardware resource manipulation. Its fine-grained control and direct hardware access offer notable performance benefits. However, C mostly uses an imperative paradigm, denoting that commands run one after the other sequentially. Due to this built-in restriction, C cannot fully utilize the capabilities of contemporary multi-core and multi-processor systems.

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) that supports multi-platform shared-memory programming architecture. It addresses C's sequential limitation by adding commands that enable parallel programming. OpenMP allows developers to parallelize loops, allows multithreading, and synchronizes their execution, which empowers C programs to utilize multiple cores within a single system, boosting performance for suitable algorithms.

Message Passing Interface (MPI) is a widely used standard for message-passing libraries in parallel computing. Message Passing Interface (MPI) is used when the amount of data and the computing demands exceed the capacity of a single computer. MPI allows communication between numerous processes operating on different processors or nodes in a computing cluster. It provides functions for sending and receiving messages, synchronizing processes, and managing data distribution among nodes. This distributed approach allows for tackling large-scale computations by harnessing the combined processing power of multiple machines.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It permits developers to perform general-purpose processing tasks besides graphics rendering on NVIDIA GPUs. CUDA provides a framework for writing parallel programs that can execute on NVIDIA GPUs. Developers manage memory and execution for each platform by writing code for the CPU (host) and GPU (device). Machine learning, image processing, and scientific simulations all greatly benefit from CUDA's parallelization features.

# REVIEW OF THREE SEARCH ALGORITHMS

Search algorithms offer varied approaches to locate elements within data structures efficiently. This report will provide a review of three search algorithms, mainly Binary Search, Linear Search, and Depth-First Search (DFS).

Binary Search is a search algorithm that works on sorted arrays. Binary Search operates on the principle of the divide-and-conquer approach, repeatedly halves the search space based on the target value's comparison to the element at the midpoint. It starts by comparing the middle element of the array with the target element. The search is complete if the middle element equals the target element. Otherwise, if the middle element exceeds the target element, the search continues to the left half of the array. If the middle element is less than the target element, the search continues to the right half of the array. This process is repeated until the target element is found or the search space is exhausted. Binary Search has a time complexity of O(log n), which makes it incredibly efficient for searching large, sorted data sets. Binary Search demands a sorted array to perform the best it can (GeeksforGeeks, 2023).

Linear Search presents a simple search algorithm that works on unsorted arrays. Linear Search embraces simplicity in contrast with Binary Search. It starts at the beginning of the array and compares each element with the target element until a match is found or the end of the array is reached. It sequentially iterates through each element in the data set until the target is found. Linear Search has a time complexity of O(n) that may be inefficient for larger datasets or sorted arrays. However, its lack of reliance on sorted data and simplicity makes it an attractive candidate for parallel implementations due to its independence between elements (GeeksforGeeks, 2023).

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts at the root node and systematically explores as far as possible along each branch before backtracking and examining alternate paths. DFS can solve problems such as finding connected components in a graph, determining whether a graph is bipartite, and finding the shortest path between two nodes in a graph. DFS has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. Its parallel implementation requires careful design to avoid redundant exploration and deadlocks.

# JUSTIFICATION

Linear Search is selected as the primary algorithm for implementation among the three searching algorithms due to several factors.

Linear Search's versatility and simplicity are some of the primary factors it selected for implementation. Linear search's simplicity makes it an ideal candidate for highlighting paradigm differences without introducing unnecessary complexity. Implementing it across different languages or APIs enables a clear comparison of syntax, performance, and overhead in diverse programming environments. Linear search does not demand sorted data, making it adaptable to various scenarios. It is particularly advantageous when dealing with dynamically changing data or situations where pre-sorting is impractical. Linear search uses minimal additional memory compared to algorithms like binary search, which may require maintaining additional data structures for sorted data organization. The core logic of linear search can be easily adapted to work with various data structures like arrays, linked lists, and custom data structures.

The ease of parallelization implementation is one of the critical advantages of Linear Search. The ease of switching from an imperative method to a parallelized version is made possible by the simplicity of linear search. One way to demonstrate the transition from sequential to parallel execution is to parallelize linear search using OpenMP, MPI, or CUDA methods. This transition shift aids in demonstrating the difficulties, benefits, and variations in performance between imperative and parallel implementations. Its independent comparison of each dataset element facilitates efficient parallelization, enabling effective division of work among threads or processors.

Implementing linear search in imperative, such as in C, and parallel, such as OpenMP, MPI, or CUDA paradigms, allows for a comprehensive comparison. It offers a tangible demonstration of how programming paradigms impact algorithmic efficiency. The apparent contrast between linear search's sequential and parallel implementations helps learners understand the trade-offs and considerations in choosing the appropriate paradigm for a given task.

# TOOLS

## A. Development Tools for C

C programming is one of the most popular programming languages because it has a robust development tool that makes coding, debugging and optimization process easier. Despite being quite old programming language, C still able to maintain its popularity as it is a fundamental language in the field of computer sciences (W3schools, n.d.). In the context of C programming, the Integrated Development Environment (IDE) improved the development experience especially for the programmer as it provides them with user-friendly interfaces and various functionalities such as code completion, debugging and version control integration. These functionalities deliver the development tools that ease the work as well as making the coding tasks more efficient in every aspect. Several development tools can be used to implement C which are Visual Studio Code, Eclipse, NetBeans and Dev-C++.

Another development tool for C is GNU Compiler Collection (GCC), which will compile the overall code and create executable files as well as convert the code to assembly level while linking the code with any library dependencies (Incredibuild, 2022). GCC is designed to support various programming languages including C and C++, which use different compilers for different languages. In terms of debugging, GNU Debugger (GDB) is a powerful debugger for C that runs in command lines which allows programmers to fix the errors by inspecting the running code. In comparison with GCC, GDB is not a compiler but it works with the executable files produced by compilers like GCC. GDB is used by the programmer to set breakpoints, which are the indicators where execution is paused, allowing to inspect current program as well as stepping through code during execution to identify and resolve the errors (Chyhyryk, 2023).

Profiling tools are also one of the development tools for C and it plays a crucial role in monitoring the execution of the program to retrieve information regarding performance as well as behavior of the program (Chyhyryk, 2023). For example, gprof helps in analyzing the time taken for a function to execute within the program and Valgrind's Callgrind helps in generating a call graph that enhances the developer's understanding of the flow of execution. Hence, profiling tools can provide insights regarding the amount of time spent on each function.

## B. Development Tools for OpenMP

Open Multi-Processing (OpenMP) is a widely adopted API that directs multi-threaded and shared memory parallelism for C, C++, or FORTRAN programs. As a support for parallel programming in shared-memory environments, OpenMP provides programmers with an easy way to utilize the capabilities of multiple processors. OpenMP is managed by the OpenMP Architecture Review Board (OpenMP ARB) and specified by several hardware and software companies. (1.1. Introduction of OpenMP — Parallel Programming and Performance Optimization with OpenMP, n.d.)

OpenMP is a cross-platform, cross-compiler solution that supports many platforms like Linux, macOS, and Windows. Standard compilers include GNU Compiler Collection (GCC), Intel Fortran, and LLVM/Clang compiler. The GCC is a freely available open-source compiler compatible with various OS, including Linux, Solaris, AIX, MacOS, Windows, and NetBSD. It also works with x86_64, PowerPC, ARM, and many more architectures. (tim.lewis, n.d.) GCC is an integrated distribution of compilers for several major programming languages; it is compatible with multiple popular programming languages, including C, C++, and Fortran, all utilized in OpenMP. (Using the GNU Compiler Collection (GCC): G++ and GCC, n.d.) Meanwhile, Intel Fortran is available for C, C++, and Fortran in Windows, Linux, and macOS operating systems. On the other hand, Clang is the LLVM compiler front-end component and is permissively licensed for C or C++. (tim.lewis, n.d.)

The integrated Development Environments that support OpenMP are Eclipse, Visual Studio, Code::Blocks, NetBeans and others. In this program, we selected Visual Studio to implement OpenMP as it is user-friendly and easy to use. We don't need to get or link against OpenMP libraries individually because Visual Studio automatically includes all the necessary OpenMP libraries when we enable OpenMP support.

## C. Development Tools for MPI

Message Passing Interface (MPI) is widely known to exchange messages between parallel programs. It is not endorsed by any standards organization, but it's considered to be an industry standard. It provides language bindings for multiple programming languages such as C, C++ and FORTRAN (Yang et al, 2011). Multiple open source implementations of MPI have been adapted by various organizations. A lot of widely used MPI implementations include OpenMPI, MPICH and MS-MPI (Zhou, et al., 2019). OpenMPI is developed and maintained by a group of academics, and industry partners making it suitable for common usage.

According to RowCoding, MPICH is designed to be equipped with the latest MPI standard and basis for derivative implementations to meet special needs. Both implementations are highly portable and support a wide array of platforms such as MacOS, Linux distribution and Windows. MPICH, supports proprietary high-end computing systems such as Blue Gene and Cray based on the official MPICH webpage. The complex nature of MPICH consist of three layers to allow organisations to take advantage of certain functions of their system. For example, IBM BlueGene reduction operations (Gropp et al., 2021) In terms of process management, OpenMPI used to be superior compared to MPICH, but have been fixed since, resulting in both equally competent.

A derivative of MPICH is Microsoft MPI (MS-MPI), a specialised MPI standard for Windows platform. A few benefits include high performance in Windows operating system, binary compatibility and integration of code that uses MPICH (AnnaDaly, 2022). MS-MPI is selected to be implemented in this research. One of the debugging tools for MPI is GDB debugger (Malakar, P., 2019, December). It is part of Linux debugging toolset which provides step-by-step computer code execution and function modification. The development environment that supports MPI are IDEs like Eclipse, JetBrains and Visual Studio. In this implementation, we use Visual Studio as it is easy to use. The files for MPI can be obtained from the Microsoft official website. Once installed, the directory for the MPI header is set in Visual Studio to enable MPI.

## D. Development Tools for CUDA

CUDA, which stands for Compute Unified Device Architecture, is an innovative programming paradigm and parallel computing platform created by NVIDIA. Its primary objective is to leverage the processing capabilities of NVIDIA GPUs (Graphics Processing Units) for general-purpose computing tasks. In order to construct applications that are both efficient and parallelized, CUDA programmers commonly utilize a suite of development tools supplied by NVIDIA.

Central to this collection of tools is the CUDA Toolkit, which comprises vital elements including the CUDA compiler (referred to as 'nvcc'), libraries, headers, CUDA runtime, and a multitude of utilities. The code generates executable GPU processors by combining CUDA code compilation with conventional C code using the CUDA compiler (nvcc).

In order for the CUDA runtime and the GPU to communicate without interruption, it is the responsibility of the developer to install the appropriate NVIDIA GPU drivers on their development devices. It is crucial that these drivers exhibit compatibility with both the particular GPU model being employed and the CUDA Toolkit version. The NVIDIA CUDA Compiler ('nvcc') is an essential component in the CUDA code compilation process, providing support for the CUDA C++ and CUDA C languages. It generates host and device code by automatically invoking the device compiler and the host compiler (e.g., 'cl' or 'gcc'). It integrates seamlessly with conventional C/C++ code.

An additional essential element of the CUDA development toolset, the CUDA Runtime API provides a collection of functions that can be invoked from the host code. By facilitating operations such as device administration, GPU memory allocation, and kernel launches, these functions optimize the host system-GPU interface.

In order to assist in the debugging and profiling of GPU-accelerated applications, NVIDIA offers the Nsight suite, which comprises Microsoft Visual Studio-integrated tools such as Nsight Visual Studio Edition. These tools provide developers with an extensive array of functionalities to assess and optimize their CUDA code, thereby augmenting the overall development process.

For this C program to implement the linear search algorithm with CUDA, the following development tools are utilized: the CUDA Toolkit, NVIDIA GPU drivers, and the CUDA compiler.

# IMPLEMENTATION

## A. Implementation and Pseudocode for C

```
START
    Initialize array
    Calculate size of array
    Set the key value

    Call procedure LinearSearch(arr,size_arr,key,comparisons)
        Set comparisons to 0
        for each element in the array
            Increase comparisons by 1
            if match element equal to key
                return current element's index
            end if
        end for
    end procedure

    if element that match key value present
        Display the element's index
    else
        Display message that shows no match element present
    end if
    Display the total number comparison
END
```

The figure above shows the pseudocode of implementation of linear search algorithm in C program. At the beginning, the element in the array will be initialized and the key value will be assigned to variable 'key', where key value is the element that linear search will find within the array. Then, the program will initiate a call to LinearSearch procedure by passing 4 parameters including the array, size of an array, key value and variable that store total number of comparisons made to search for key value. Within the LinearSearch procedure, all of the operations and assignment would iterate through each element in the array to determine whether it matches a specified key value that have been initialized.

As the algorithm iterates through each element in the array, the variable for storing a number of comparisons is incremented by 1 in each iteration (every comparison made) which provides understanding regarding the efficiency of the search process. The efficiency of the linear search algorithm depends on the size of the array as well as the location of the key value within the array. Hence, the total number of comparisons shows the performance of the search algorithm. Within each iteration, the algorithm will check whether the current element is equal to the key value or not. If the element matches the key, the algorithm halts immediately and returns the index of the element, indicating the location of the matching element in the array.

After the linear search algorithm was done, the main program will check whether the element that match key value is found or not. If found, it will display message that show the index of element, otherwise, the program will display message indicating that no element matches the key value in the array. Lastly, the program will display the total number of comparisons made during the search. In the context of time complexity, worst-case scenario can be examined if the number of comparisons is equal to the size of the array, means that the element that matches the key is at the end of the array. On the other hand, best-case scenario can be observed when the number of comparisons is equal to one, means that the element located at the first element of array.

## B. Implementation and Pseudocode for OpenMP

```
function LinearSearch(arr, size, key, comparisons) :
// Initialize comparisons count
Set comparisons to 0

// Variable to store the index of the found element
 Set index to - 1

// Starting parallel traversal
#pragma omp parallel
{
  int thread_id = omp_get_thread_num()
 // Parallel loop to search for the key
 #pragma omp for
  for i = 0 to size - 1
  #pragma omp atomic
  Increment comparisons by 1
  // Checking condition whether array value is equal to key value
  if arr[i] equals key :
  // Element found, update the index and print thread information
   #pragma omp critical
   {
    Set index to i
    Print ("Thread " + thread_id + " found the element at index " + index + "\n")
   }
 End Parallel loop
}
// Element not found or found by multiple threads, return the final index
Return index
```

The figure shows the pseudocode of implementation of linear search algorithm in OpenMP. The design of OpenMP is similar as C program, initialize the comparison, set index to -1, display total comparison and etc. The difference is OpenMP is parallel programming API, it searches the element in parallel traversal. Thus, we need the '# pragma omp parallel' directive to start a parallel region in the code. It creates a team of parallel threads and the code within this parallel region is executed simultaneously by all the threads. Each thread will execute the

enclosed block of code independently. Each thread has its unique 'thread_id' variable to identify itself through the 'omp_get_thread_num' function provided by OpenMP. While the '#pragma omp for' directive is used to parallelize the following for loop. It distributes the iterations of the loop among the available threads and allowing multiple threads to execute the loop concurrently.

Furthermore, the '#pragma omp atomic' is also important for ensuring that a specific operation on a shared variable is performed atomically. In this case, it ensures that the increase in the comparison is achieved as a single, uninterruptible operation. It is important as there are multiple threads that can access and modify the shared data simultaneously. Without the atomic operation, race conditions can occur and lead to an unpredictable and incorrect outcome. The pseudocode also contains '#pragma omp critical' for a critical section to prevent the shared 'index' variable from concurrent updates by multiple threads. It appears to have a similar function with 'omp atomic', but the 'omp critical' is a generalized mutual exclusion mechanism and can wrap any kind of code, while 'omp atomic' limits the types of operations it supports. This critical section ensures that only one thread can execute the block within the curly braces at a time, thereby allowing mutual exclusion. Lastly, it will also print the thread_id which successful found the element and the actual element location.

## C. Implementation and Pseudocode for MPI

```
/* Linear Search with MPI */
MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int globalSize = sizeof(arr) / sizeof(arr[0]); // Size of the array
int size_array = globalSize / numProcs;

int *chunk = (int *)malloc(size_array * sizeof(int));

if(myRank == 0){ // Master rank
    MPI_Scatter(arr, size_array, MPI_INT, chunk,size_array, MPI_INT,
                0, MPI_COMM_WORLD);
}
else{// Child rank
    MPI_Scatter(arr, size_array, MPI_INT, chunk,size_array, MPI_INT,
                0, MPI_COMM_WORLD);
}
double startTime = MPI_Wtime();

// Calling MPILinearSearch function
localIndex = MPILinearSearch(chunk, size_array, key, &comparisons);
```

```
double endTime = MPI_Wtime();
double executionTime = endTime - startTime;

Each process displays the result;

MPI_Finalize();
```

The figure above illustrates the design of MPI collectives with linear search algorithms. First, the MPI environment is initialized (*MPI_Init*). *&argc and &argv* will look for any arguments on the command line that is specific to the MPI functions. It will configure and remove any MPI related configurations after the termination. After the MPI processes is terminated, any MPI routines will not affect the subsequent programs.

*MPI_Comm_size* and *MPI_Comm_rank* is used to retrieve the total number of processes and the rank of the processes from the specified communicator respectively. *MPI_COMM_WORLD* acts as a communicator between all the processes. MPI routines requires a specified communicator to be defined. Thus, *MPI_COMM_WORLD* is the default communicator. It groups all the processes when the program started. Each process can send and receive updates when the program is running. The array size is identified and divided by the number of processes to get an even block for each process.

*MPI_Scatter* scatters data from one process to all the others. In this case, rank 0 is regarded as the master process, thus the root is set to 0. The first three parameters are the sender while the next three is the receiver part. It will distribute the original array from set root into blocks of partial array size into other processes. The children (non-master processes) will receive a specified size of array along with the divided chunks of array. In addition, the master rank will also receive the chunks. The datatype is set to *MPI_INT* for integer value as the array list contains integers.

All the process executes the same program. They initiate MPI and complete their allocated part. The time is collected with the *MPI_Wtime()* function to prove the parallelism of MPI. The linear search function is then called, and the results are displayed along with their execution time. Lastly, the MPI environment is terminated using MPI_Finalize. The function will be the last MPI function called and no other MPI routines are called after it.

## D. Implementation and Pseudocode for CUDA

```
// Constants
const N = 1000
const threadsPerBlock = 256

// CUDA Kernel for Linear Search
function linearSearch(array, size, target, result, comparisons)
    tid = threadIdx.x + blockIdx.x * blockDim.x
    shared comparisonsInBlock[threadsPerBlock]
    comparisonsInBlock[threadIdx.x] = 0
    synchronize threads

    // Search in parallel
    for i = tid to size with step blockDim.x * gridDim.x
        comparisonsInBlock[threadIdx.x]++
        if array[i] == target
            atomicMin(result, i)
            break

    synchronize threads

    // Parallel reduction to find total comparisons in the block
    for i = blockDim.x / 2 to 0 with halving step
        if threadIdx.x < i
            comparisonsInBlock[threadIdx.x] += comparisonsInBlock[threadIdx.x + i]
        synchronize threads

    // Accumulate total comparisons at thread 0
    if threadIdx.x == 0
        atomicAdd(comparisons, comparisonsInBlock[0])

// Main function
function main()
    // Initialize array and copy to GPU
    array[N]
    values[] = {402,279,570,263,...}  // (provided array refere to LinearCUDA.cu source code)
    for i = 0 to N - 1
        array[i] = values[i]

    // Allocate GPU memory
    dev_array, dev_result, dev_comparisons = cudaMalloc()

    // Copy array from host to device
    cudaMemcpy(dev_array, array, N * sizeof(int), cudaMemcpyHostToDevice)

    // User input
    maxTargets = 5
    numTargets, targets[maxTargets] = getUserInput(maxTargets)

    // Loop over each target
    for targetIndex = 0 to numTargets - 1
        target = targets[targetIndex]

        // Set initial result and comparisons count
        cudaMemcpy(dev_result, &N, sizeof(int), cudaMemcpyHostToDevice)
        comparisons = 0
        cudaMemcpy(dev_comparisons, &comparisons, sizeof(int), cudaMemcpyHostToDevice)

        // Timing events
        start, stop = cudaEventCreate()

        // Record start time
        cudaEventRecord(start)

        // Launch linear search kernel
        gridSize = (N + threadsPerBlock - 1) / threadsPerBlock
        linearSearch<<<gridSize, threadsPerBlock>>>(dev_array, N, target, dev_result, dev_comparisons)

        // Record end time
        cudaEventRecord(stop)

        // Copy back result and comparisons count
        result = cudaMemcpy(dev_result, sizeof(int), cudaMemcpyDeviceToHost)
        comparisons = cudaMemcpy(dev_comparisons, sizeof(int), cudaMemcpyDeviceToHost)

        // Synchronize and handle errors
        cudaDeviceSynchronize()
        handleCudaErrors()

        // Print results and timing information
        printResults(target, result, comparisons, start, stop)

    // Free GPU memory
    cudaFree(dev_array)
    cudaFree(dev_result)
    cudaFree(dev_comparisons)

// Helper function to get user input
function getUserInput(maxTargets)
    printf("Linear Search Implementation in CUDA\n")
    printf("====================================\n\n")

    // Get number of elements to compare
    printf("Enter the number of elements to compare (up to %d): ", maxTargets)
    scanf("%d", &numTargets)
```

```
    // Validate user input
    if numTargets <= 0 or numTargets > maxTargets
        printf("Invalid number of elements.\n")
        exit(1)

    // Get target values
    for i = 0 to numTargets - 1
        printf("Enter the number %d: ", i + 1)
        scanf("%d", &targets[i])

    // Return user input
    return numTargets, targets

// Helper function to handle CUDA errors
function handleCudaErrors()
    cudaError = cudaGetLastError()
    if cudaError != cudaSuccess
        fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(cudaError))
        exit(1)

// Helper function to print results and timing information
function printResults(target, result, comparisons, start, stop)
    // Print result for each target
    if result < N and result >= 0
        printf("Element number %d found at index %d\n", target, result)
    else
        printf("The Target's Element %d is not found.\n", target)

    // Calculate and print execution time
    milliseconds = 0
    cudaEventElapsedTime(&milliseconds, start, stop)
    printf("Execution Time: %f ms\n", milliseconds)

    printf("Number of Comparisons made to find the key value: %d\n\n", comparisons)
```

The pseudocode above outlines a CUDA implementation of linear search which simple searching algorithm using parallel processing on a GPU.The pseudocode starts by defining constant for array size (*N*) and threads per block (*threadsPerBlock).* These constants are for establishing GPU parallel execution.

The next step is to create a CUDA kernel function called *linearSearch*. This kernel does a parallel linear search on an array. This kernel performs a linear search on an array in parallel. It takes parameters such as the array to search (*array*), its size (*size*), the target value to search for (*target*), a result variable to store the found index (*result*) and a variable to count the number of comparisons made (*comparisons*). Parallelism in CUDA is used by the function, which starts multiple threads to search the collection at the same time.

Each thread in the kernel is given an index called a *tid* that is based on where it is in the thread grid.These are built-in variables provided by CUDA; (threadIdx.x) is the index of the thread within its block (blockIdx.x) is the index of the block within the grid, (blockDim.x) is the number of threads per block. The shared memory *(__shared__)* is used to store the number of comparisons made by each thread in a block. A loop goes through the parts of the array and updates the result with an atomic operation if a match is found. Atomic operations are to ensure correct result in case of concurrent updates. Once the search is complete, a parallel reduction operation is performed to sum up the comparisons made by threads in the block.

The main function of the *linearSearch* kernel include initializes an array '*array'* with predefined values,searching for the element in parallel.Allocate GPU memory for the array *(dev_array)*, result *(dev_result)* and comparisons count *(dev_comparisons).* The array is copied

to the device (GPU) from the host (CPU) such *cudaMalloc(), cudaMemcpy(), cudaFree()* is used as CUDA runtime API functions for memory management.

*__syncthreads()* is used to synchronizes all threads in a block that is ensures all threads have reached the same point in the code before proceeding.

After the kernel function is set up, the main function is constructed. The program prompts the user to enter the number of elements to compare "*getUserInput(maxTargets)"* with a maximum limit specified by the variable *maxTargets.* It checks if the input is valid and then prompt the user for each element value. Then, store the input values in the variable *numTargets* and the array *targets*.

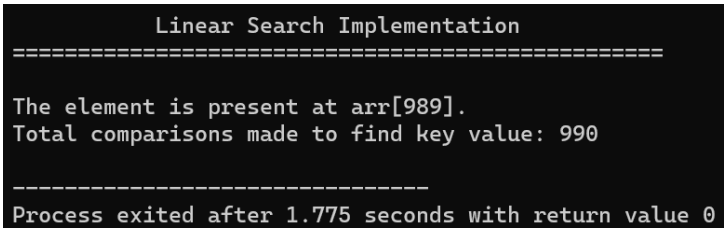
The code proceeds iteratively in order to locate every element that the user has input. The kernel is initiated in dynamic parallelism for each element.As *for "linearSearch<<<gridSize, threadsPerBlock>>>"* is to launch a kernel with specific number of blocks and threads per block.then,the execution time is measured using CUDA runtime API functions which are *cudaEventCreate(), cudaEventRecord(),cudaEventElapsedTime(), cudaEventDestroy()*.Also the CUDA error checking is to ensure that kernel execution is error-free.The *cudaDeviceSynchronize()* ensures that the CPU waits for all previously issued CUDA calls to complete before continuing.

To get user input on the number of parts to compare and the goal values, there is a helper function called "*getUserInput"*. This function also checks the numbers that the user enters to make sure they are right.

Finally, the "*printResults"* function prints the result of the linear search for each element, including the index where the element was found, the execution time and the total number of comparisons made during the search.

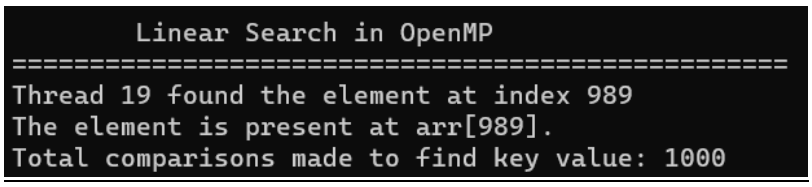
# DISCUSSION

## C




The figure above shows the execution of linear search using C. The array used in the code contain 1000 integers will be searched for the key value that have been initialized. The search process involves traversing the array sequentially by comparing each element until a key value is found. In this output, the key value which is 119 located at index 989 and it shows that the total number of comparisons made to discover key value is 990. That means integer 119 is located at the last 10 element within the array. That shows linear search algorithm has a time complexity of O(n), where n is the size of the array. This time complexity indicates that the worst-case scenario would be happen if the number of comparisons made is equal to the size of the array, which means the key value is located at the lase index inside the array. In this particular output, the algorithm can locate the key value after a relatively high number of comparisons. Hence, the efficiency of the linear search implemented using C depends on the location of the key value within the array.

## OpenMP




The figure above shows the output of linear search using OpenMP. The search process traverses the array parallel by comparing each element until the key value is found. There will be numerous threads and OpenMP will often utilize as many threads as there are available CPU cores. Even though the total comparison in OpenMP is greater than the C program, normally the search time of OpenMP for large arrays will be faster than using C that uses the sequential comparison. OpenMP use the parallelism and run all threads at the same time so the total comparison in OpenMP is equal to the array size.

## MPI

```
mpiexec -n 2 linear_search_mpi
```

```
Process 0 did not find the key value
        Total comparisons made: 500
        Execution time for Process 0: 0.000002 seconds

Process 1 found the key value at its local index: 489 and the global index: 989
        The element is present at index arr[989]
        Total comparisons made: 490
        Execution time for Process 1: 0.000002 seconds
```

```
mpiexec -n 1 linear_search_mpi
```

```
Process 0 found the key value at its local index: 989 and the global index: 989
        The element is present at index arr[989]
        Total comparisons made: 990
        Execution time for Process 0: 0.000004 seconds
```

The figure above shows the MPI execution of linear search. The MPI file is first build, producing an execution file. The command to run the execution file is mpiexec -n 2 linear_search_mpi. The number of parallel processes in defined. In this case, we initiate two processes, meaning the array is divided into two. The output shows the search efficiency along with the performance of each process. To prove the parallelism of MPI, we also execute it with only one process (no parallelism). The execution time between both instances is different. The one with more process is faster than searching linearly. As we increase the number of processes, the speed will be proportional as well. This further proves the advantage of parallel computing over sequential.

However, there are concerns regarding the size of the array. Depending on it, the number of processes to achieve the highest efficiency varies. For our array of a thousand values, dividing into too many processes might have an adverse effect on the overall computation power. More processes are suitable for larger array values. However, smaller array size can result in longer execution time. The right amount needs to be carefully selected.

## CUDA

```
Linear Search Implementation in CUDA
====================================

Enter the number of element to compare(up to 5): 2
Enter the number 1: 868
Enter the number 2: 1001


Element number 868 found at index 459
Execution Time: 22.556448 ms
Number of Comparisons made to find the the key value: 1000

The Target's Element 1001 is not found.
Execution Time: 0.036864 ms
Number of Comparisons made to find the the key value: 1000
```

The presented output from the execution of the CUDA-based linear search algorithm provides insights into the functionality and performance of the implemented code. The code successfully compiles using the NVIDIA CUDA compiler (`nvcc`), producing an executable named "LinearCUDA." Upon execution, the program prompts the user to input the number of elements to compare, allowing for a maximum of 5 elements. In this specific example, the user inputs 2 elements to search for: 868 and 1001.

The first element, 868, is found at index 459, as indicated by the output "Element number 868 found at index 459." The execution time for this search is reported as 22.556448 milliseconds, and the number of comparisons made during the search is revealed as 1000. This information provides valuable insights into the efficiency of the linear search algorithm for this specific target.

On the other hand, the second element, 1001, is not found in the array, as indicated by the output "The Target's Element 1001 is not found." The execution time for this unsuccessful search is considerably lower, reported as 0.036864 milliseconds, with the same number of comparisons (1000). This difference in execution times reflects the algorithm's adaptability to different scenarios, emphasizing the variability in search times based on the presence or absence of the target element.

The recorded execution times and comparison counts demonstrate the parallel nature of the CUDA implementation, allowing for concurrent searches on the GPU. The relatively low

execution time for unsuccessful searches further highlights the efficiency of parallel algorithms for scenarios where the target element is not present in the array.

In conclusion, the output provides valuable information on the successful and unsuccessful searches, execution times and comparison count, showcasing the effectiveness of the CUDA-based linear search algorithm in parallel processing. The reported metrics contribute to the understanding of the algorithm's performance characteristics and its potential applications in scenarios involving large datasets and parallelizable search tasks.

# CONCLUSION

In conclusion, the comparison of imperative and parallel programming paradigms is explored while reviewing three search algorithms, Binary Search, Linear Search, and Depth-First Search. While each algorithm offers unique characteristics, Linear Search is the most fitting for implementation due to its simplicity, adaptability, and ease of demonstration across programming paradigms. We aimed to demonstrate the essential distinctions between sequential and parallel execution using Linear Search implemented in imperative (C language) and parallel paradigms (OpenMP, MPI, and CUDA). As Linear Search is simple, it is easy to compare syntax, overhead, and performance in various programming environments and see how each paradigm approaches the algorithmic task.

Several challenges are expected while exploring these imperative and parallel paradigms with Linear Search implementation. Resource constraints, particularly in environments like CUDA, limitations related to memory allocation, thread management, and data transfer between the CPU and GPU, provide significant obstacles that could significantly affect the efficiency of parallel implementations. Besides, learning different programming frameworks required additional time. Extended research is necessary as each framework presents its syntax, semantics, and programming paradigm. For instance, OpenMP is used for shared-memory parallelism, MPI is used for distributed-memory systems, and CUDA is used for GPU-based computations. Identifying and resolving errors within the interplay of multiple threads or processes presented unique challenges compared to traditional sequential debugging.

Despite these challenges, valuable knowledge and experiences are gained throughout this assignment. We developed a practical understanding of the advantages and disadvantages of parallel programming paradigms and gained a deeper understanding while designing algorithms for parallel programming paradigms. We also improved our debugging and problem-solving abilities in parallel programming. Finally, we acquired additional knowledge through hands-on experience in performance analysis and optimization techniques.

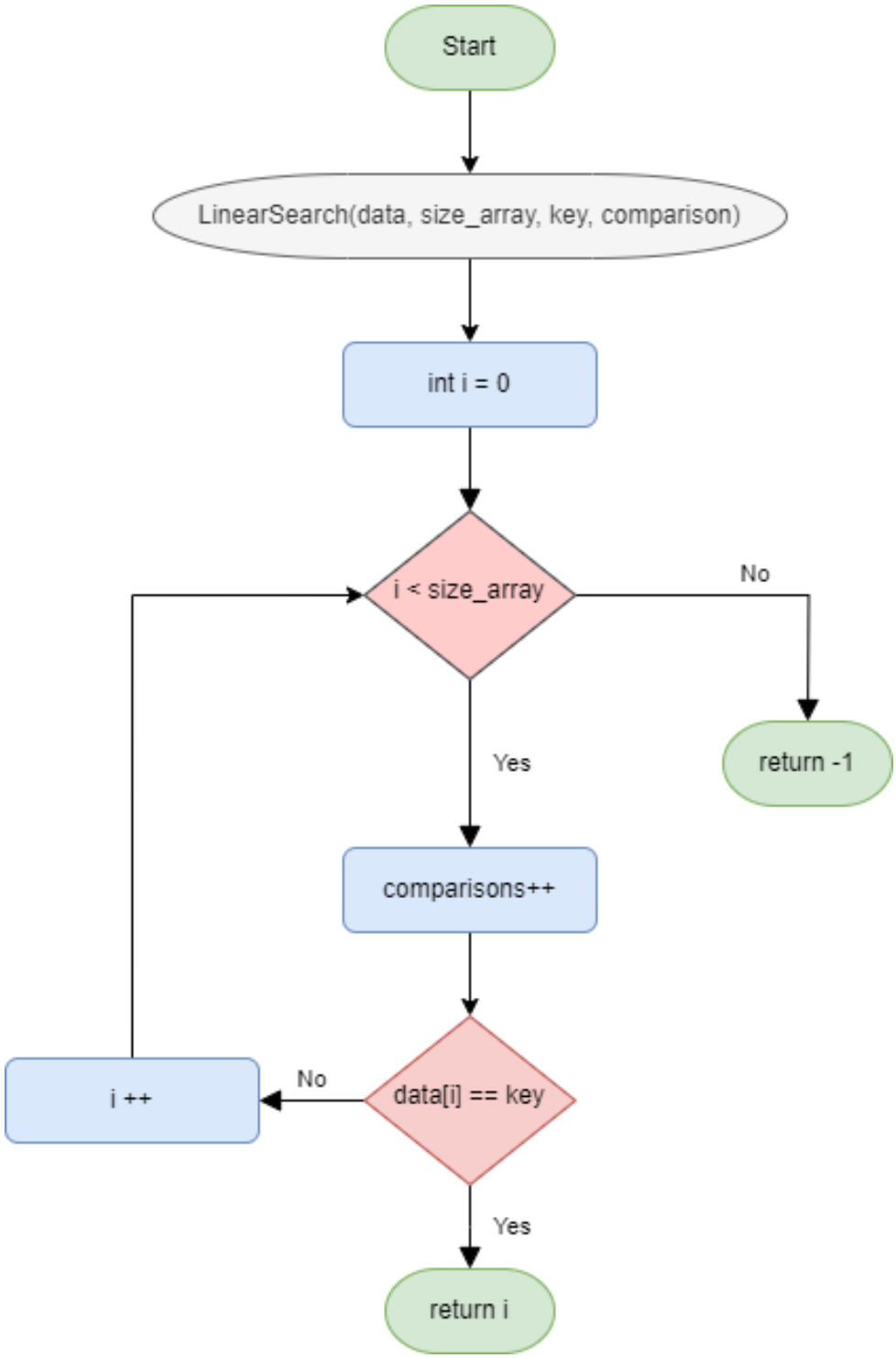
# REFERENCES


1. Gropp, W., Thakur, R., & Balaji, P. (2021). Translational Research in the MPICH project. *Journal of Computational Science*, *52*, 101203. https://doi.org/10.1016/j.jocs.2020.101203
2. Malakar, P. (2019). Experiences of teaching parallel computing to undergraduates and Post-Graduates. *2019 26th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, 40–47. https://doi.org/10.1109/hipcw.2019.00020
3. Zhou, H., Gracia, J., & Schneider, R. (2019). MPI collectives for multi-core clusters. *Workshop Proceedings of the 48th International Conference on Parallel Processing*, 1–10. https://doi.org/10.1145/3339186.3339199
4. Wächter, A., & Biegler, L. T. (2005). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, *106*(1), 25–57. https://doi.org/10.1007/s10107-004-0559-y
5. Yang, C., Huang, C., & Lin, C. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Computer Physics Communications, 182(1), 266–269. https://doi.org/10.1016/j.cpc.2010.06.035
6. 1.1. Introduction of OpenMP — Parallel Programming and Performance Optimization With OpenMP. (n.d.). Passlab.github.io. https://passlab.github.io/OpenMPProgrammingBook/openmp_c/1_IntroductionOfOpenMP.html#:~:text=OpenMP%20is%20a%20standard%20parallel\
7. AnnaDaly. (2022, September 21). Microsoft MPI - Message Passing Interface. Microsoft Learn. https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi
8. Chyhyryk, O. (2023, October 22). *Must-have C programming tools for beginner and advanced developers*. MarketSplash. https://marketsplash.com/tutorials/c/c-programming-tools/
9. GeeksforGeeks. (2023, June 2). *Linear Search Algorithm Data Structure and Algorithms Tutorials*. https://www.geeksforgeeks.org/linear-search/
10. GeeksforGeeks. (2023b, July 26). *Binary Search Data Structure and algorithm Tutorials*. https://www.geeksforgeeks.org/binary-search/
11. GeeksforGeeks. (2023, October 6). *Introduction to searching Data Structure and algorithm tutorial*. https://www.geeksforgeeks.org/introduction-to-searching-data-structure-and-algorithm-tutorial/

12. Heller, M. (2022, September 16). *What is CUDA? Parallel programming for GPUs*. InfoWorld. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html
13. Incredibuild. (2022, December 28). *What is GNU Compiler Collection (GCC)*. GCC. https://www.incredibuild.com/integrations/gcc
14. Krishna, A. (2022, January 11). *Search algorithms – linear search and binary search code implementation and complexity analysis*. freeCodeCamp.org. https://www.freecodecamp.org/news/search-algorithms-linear-and-binary-search-explained/
15. MPICH Overview | MPICH. (n.d.). https://www.mpich.org/about/overview/
16. Open MPI: Open Source High Performance Computing. (n.d.). https://www.open-mpi.org/
17. S, R. A. (2023, July 27). *What is Linear Search Algorithm | Time Complexity*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm
18. Staff, C. (2023, November 21). *Python vs. Java: Which Should I Learn?* Coursera. https://www.coursera.org/articles/python-vs-java
19. Tarik. (2022, October 20). MPICH vs OpenMPI - Row Coding. Row Coding. https://rowcoding.com/mpich-vs-openmpi/
20. tim.lewis. (n.d.). OpenMP Compilers & Tools. OpenMP. https://www.openmp.org/resources/openmp-compilers-tools/
21. Using the GNU Compiler Collection (GCC): G++ and GCC. (n.d.). Gcc.gnu.org. Retrieved December 26, 2023, from https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/G_002b_002b-and-GCC.html
22. W3schools (n.d.) *C introduction*. Introduction to C. (n.d.). https://www.w3schools.com/c/c_intro.php

# APPENDIX

Linear Search Flowchart



First page of research papers

ARTICLE IN PRESS

Journal of Computational Science xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

# Translational research in the MPICH project

William Gropp [a], Rajeev Thakur [b,*], Pavan Balaji [b]

[a] University of Illinois, Urbana, IL 61801, USA
[b] Argonne National Laboratory, Lemont, IL 60439, USA

ARTICLE INFO

*Keywords:*
MPI
MPICH
Translational computer science
Message passing libraries

ABSTRACT

The MPICH project is an example of translational research in computer science before that term was well known or even coined. The project began in 1992 as an effort to develop a portable, high-performance implementation of the emerging Message-Passing Interface (MPI) standard. It has enabled the widespread adoption of MPI as a way to write scalable parallel applications on systems of all sizes including upcoming exascale supercomputers. In this paper, we describe how the translational research process was used in MPICH, how that led to its success, the challenges encountered and lessons learned, and how the process could be applied to other similar projects.

## 1. Introduction

MPI (Message Passing Interface) is a portable, standard API for writing parallel programs using a distributed-memory programming model. MPI began as an effort in 1992 by a group known as the MPI Forum to define a standard API for message passing instead of the plethora of different APIs provided by existing vendor and research message-passing libraries at the time. Version 1.0 of the MPI specification was released in 1994, and MPI has been widely used since then for developing parallel scientific applications on all kinds of parallel computer systems. Several new versions of the specification, with additional features, have been released over the years, the latest being MPI 3.1 in 2015. MPI 4.0 is expected to be released in a year. MPI has been essential to the success of supercomputing in the past 25 years. It has provided a standard message-passing interface for scalable parallel systems and clusters. It is supported by all supercomputer and cluster vendors, and it is used by essentially all science and engineering parallel simulation codes running on supercomputers and is expected to continue to do so in the next decade.

MPICH [14] began as a research project in 1992 with the goal of developing a portable implementation of the evolving MPI standard as it was being defined. The idea was to provide early feedback on decisions being made by the MPI Forum and provide an early implementation to allow users to experiment with the specification even as it was being developed. In many cases, the MPI design pushed features for new capabilities not seen in previous systems, and MPICH became the reference implementation that proved that these features can be implemented efficiently. Targets for the implementation included all systems capable of supporting the message-passing model—from single-node, shared-memory systems to clusters of systems and the largest supercomputers in the world. MPICH has been and continues to be the de facto implementation driver leading to the success of MPI.

MPICH was always intended as both a research project and a software development project. As a research project, its goal is to explore methods for delivering the highest possible performance for communication in parallel applications on all available hardware. As a software project, its goal is to promote the adoption of the MPI Standard by providing users with a high-quality, high-performance implementation on a diversity of platforms, while aiding vendors in providing their own customized implementations.

MPICH was designed from the ground up to serve as a reference for other derived implementations. It was cleverly layered so that vendors could replace the lowermost "driver" level of MPICH, thus customizing it to their own hardware, while reusing most code. More important, vendors could choose to customize fewer or more layers in MPICH, and all did so.

While the full picture is more complex, MPICH has three major layers [6]. The top layer provides the implementation of the MPI API and provides code for features that are part of MPI but not critical to performance, such as MPI attributes (a way for software libraries and tools to attach information to MPI objects) and error reporting. The next layer, called the "device," provides architecture-specific optimizations for performance-critical portions of the MPI interface, including point-to-point communication, collective communication with an

* Corresponding author.
*E-mail address:* thakur@anl.gov (R. Thakur).

https://doi.org/10.1016/j.jocs.2020.101203
Received 29 May 2020; Received in revised form 20 July 2020; Accepted 13 August 2020
Available online 21 August 2020


Please cite this article as: William Gropp, *Journal of Computational Science*, https://doi.org/10.1016/j.jocs.2020.101203

# MPI Collectives for Multi-core Clusters: Optimized Performance of the Hybrid MPI+MPI Parallel Codes

Huan Zhou
HLRS, University of Stuttgart
Stuttgart, Germany
huan.zhou@hlrs.de

José Gracia
HLRS, University of Stuttgart
Stuttgart, Germany
jose.gracia@hlrs.de

Ralf Schneider
HLRS, University of Stuttgart
Stuttgart, Germany
ralf.schneider@hlrs.de

## ABSTRACT

The advent of multi-/many-core processors in clusters advocates hybrid parallel programming, which combines Message Passing Interface (MPI) for inter-node parallelism with a shared memory model for on-node parallelism. Compared to the traditional hybrid approach of MPI plus OpenMP, a new, but promising hybrid approach of MPI plus MPI-3 shared-memory extensions (MPI+MPI) is gaining attraction. We describe an algorithmic approach for collective operations (with allgather and broadcast as concrete examples) in the context of hybrid MPI+MPI, so as to minimize memory consumption and memory copies. With this approach, only one memory copy is maintained and shared by on-node processes. This allows the removal of unnecessary on-node copies of replicated data that are required between MPI processes when the collectives are invoked in the context of pure MPI. We compare our approach of collectives for hybrid MPI+MPI and the traditional one for pure MPI, and also have a discussion on the synchronization that is required to guarantee data integrity. The performance of our approach has been validated on a Cray XC40 system (Cray MPI) and NEC cluster (Open MPI), showing that it achieves comparable or better performance for allgather operations. We have further validated our approach with a standard computational kernel, namely distributed matrix multiplication, and a Bayesian Probabilistic Matrix Factorization code.

## CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**.

## KEYWORDS

MPI, MPI shared memory model, collective communication, hybrid programming


*ICPP 2019, August 5–8, 2019, Kyoto, Japan*

ACM ISBN 978-1-4503-7196-4/19/08…$15.00
https://doi.org/10.1145/3339186.3339199

**ACM Reference Format:**
Huan Zhou, José Gracia, and Ralf Schneider. 2019. MPI Collectives for Multi-core Clusters: Optimized Performance of the Hybrid MPI+MPI Parallel Codes. In *48th International Conference on Parallel Processing: Workshops (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339186.3339199

## 1 INTRODUCTION

MPI [19] has for decades retained its dominance in the circle of the parallel programming model. It is widely used to write portable and scalable parallel applications. Although MPI was originally designed for distributed-memory systems where single-core compute nodes were connected by a network, it is also constantly tuned and adapted for gaining acceptable performance on other types of systems, such as on the shared memory symmetric multiprocessing (SMP) and compound systems tightly coupled with SMP nodes. Nowadays, the multi-core technology is extensively incorporated into the current commodity supercomputers. Therefore, the increased number of on-node data transfers come along with the growing computational capability of a single chip. This will be certain to exacerbate memory bandwidth in the pure MPI-based applications and then limit per-core affordable problem sizes since the intra-node data transfers are performed conventionally using the shared memory as the transport layer. As a result, the intention of making full use of shared memory drives the programmers to seek for the hybrid method which mixes the MPI programming model (inter-node parallelism) with a shared memory programming approach (on-node parallelism) for allowing resources on a node to be shared.

Traditionally, the shared-memory models are OpenMP [4] or Pthread [3]. This hybrid approach associates MPI rank with system-level or user-level threads, which share memory inside the same address space. OpenMP, as the most popular higher-level threading abstraction, cooperates with MPI to benefit the parallel applications, especially at high core counts where the scalability of pure MPI code suffers a lot [5, 24]. Importantly, a naive approach of migrating the pure MPI codes to the hybrid MPI+OpenMP ones is to incrementally add OpenMP directives to the computationally intense parts of the MPI application codes. This approach can produce serial sections that are only executed by the master thread as to an MPI process. In this regard, such hybrid implementation may hardly outperform the pure MPI version, especially when the scaling of the MPI version is still

# On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming

Andreas Wächter* and Lorenz T. Biegler†

March 12, 2004; revised March 19, 2004

**Abstract**

We present a primal-dual interior-point algorithm with a filter line-search method for nonlinear programming. Local and global convergence properties of this method were analyzed in previous work. Here we provide a comprehensive description of the algorithm, including the feasibility restoration phase for the filter method, second-order corrections, and inertia correction of the KKT matrix. Heuristics are also considered that allow faster performance. This method has been implemented in the IPOPT code, which we demonstrate in a detailed numerical study based on 954 problems from the CUTEr test set. An evaluation is made of several line-search options, and a comparison is provided with two state-of-the-art interior-point codes for nonlinear programming.

**Keywords:** nonlinear programming – nonconvex constrained optimization – filter method – line search – interior-point method – barrier method

## 1 Introduction

Growing interest in efficient optimization methods has led to the development of *interior-point* or *barrier* methods for large-scale nonlinear programming. In particular these methods provide an attractive alternative to active set strategies in handling problems with large numbers of inequality constraints. Over the past 15 years, there has also been a better understanding of the convergence properties of interior-point methods [16] and efficient algorithms have been developed with desirable global and local convergence properties.

To allow convergence from poor starting points, interior-point methods, in both trust region and line-search frameworks, have been developed that use exact penalty merit functions to enforce progress toward the solution [2, 27]. On the other hand, Fletcher and Leyffer [14] recently proposed filter methods, offering an alternative to merit functions, as a tool to guarantee global convergence in algorithms for nonlinear programming. The underlying concept is that trial points are accepted if they improve the objective function *or* improve the constraint violation instead of a combination of those two measures defined by a merit function.

More recently, this filter approach has been adapted to barrier methods in a number of ways. M. Ulbrich, S. Ulbrich, and Vicente [21] consider a trust region filter method that bases the acceptance of trial steps on the norm of the optimality conditions. Also, Benson, Shanno, and Vanderbei [1] proposed several heuristics based on the idea of filter methods, for which improved efficiency is reported compared to their previous merit function approach, although no convergence analysis

*IBM T.J. Watson Research Center, Yorktown Heights, NY; E-mail: andreasw@watson.ibm.com
†Carnegie Mellon University, Pittsburgh, PA; E-mail: lb01@andrew.cmu.edu

2019 26th International Conference on High Performance Computing, Data, and Analytics Workshop (HiPCW)

# Experiences of Teaching Parallel Computing to Undergraduates and Post-graduates

Preeti Malakar
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur, India
pmalakar@cse.iitk.ac.in

In this paper, we present teaching experiences of offering a course on large-scale parallel computing using message passing interface (MPI). This particular course was offered to under-graduates and graduates for the first time in the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur in a very long time. We will present what topics were covered, how we decided the course content, the class demographics, what resources were made available for the students to run their MPI jobs and discuss the output of the course. We will also discuss what were the stumbling blocks encountered while offering a parallel computing systems course without much support from teaching assistants, and some lessons that we took forward to the next time we offered this course.

***Index Terms*—MPI, Cluster, HPC, Education, Grading**

## I. Introduction

The most recent list (June 2019) of the top 500 supercomputers marks a historical achievement. All the top 500 supercomputers now are at least 1 PFLOPS systems [1], with the most powerful system having 148.6 PFLOPS maximum compute capacity while the $500^{th}$ system has 1.02 PFLOPS. The first exascale system is expected to arrive by 2021 [2]. In this era of powerful systems, we need more engineers and scientists who can efficiently use these petaflop and exaflop machines. The compute capacity of these systems is tremendously useful for taking a step forward towards solving scientific questions such as "Where should we drill new oil wells?", "When and where will the next tsunami occur?" and "Is there a drug capable of curing cancer?". However, using a petaflop or an exascale system requires one to know the intricacies of system programming, including parallel programming, networking concepts along with thinking about efficient parallel program design. Therefore, teaching parallel programming concepts is a necessity in every undergraduate and graduate Computer Science course curriculum. We need to build a workforce of engineers and scientists who are well-trained to understand how to program large-scale applications on a supercomputer. This is useful while a country plans to acquire and/or build more powerful systems. This will help solve crucial scientific problems currently faced by the world, such as weather forecasting, discover oil, drug discovery and protein folding. Today it is imperative to build the taskforce for tomorrow when there will be exascale systems available for usage. A large number of international efforts are already going on in this direction [3].

This paper reflects on experiences of teaching parallel programming (distributed memory) course. This course was offered at a large scale for the first time in many years in the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur. The reflections are based on two offerings of the course – 1 complete semester (batch 1), and another currently ongoing semester (batch 2). This course is offered to final-year undergraduates and graduate students. In exceptional cases, pre-final year undergraduates are also allowed. The course is open to all department students with interest in high performance computing and background in basic computer science systems subjects. A strong programming background was required as well. The course participants were from Computer Science, Electrical Engineering, Physics and Material Science Engineering. In this paper, we will describe the course content, course teaching methodology and evaluation schemes. We will also give an overview of what did not work well while teaching such a course for the first time at a large-scale. We will also mention the learnings from batch 1 that we took forward to batch 2.

This particular course focused on distributed memory programming using message passing interface (MPI). We did not include shared memory programming or programming using accelerators (such as GPU) because these are offered in a different course in the same semester. Each of these concepts require depth, and hence it is best to split into shared memory programming and distributed memory programming courses. In this paper, we will only look at the distributed memory programming course. In this course, we started with basic concepts of processes, spawning processes on multiple nodes and communication setup. This was followed by simple MPI programming, such as `printf` statements to output node names (compute host names) from all the processes. While this is a seemingly simple concept for experienced researchers, we noticed that this concept of multiple processes of the same program running on different hosts was completely new to many students, and had to be emphasized in the initial lectures. MPI was covered in depth, including basic point-to-point MPI communications, collective communications, and also advanced topics on parallel MPI I/O and MPI profiling. We gave special emphasis on teaching parallel program design, communication aspects, and effects of network topology [4].

978-1-7281-4894-6/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPCW.2019.00020





x

Computer Physics Communications 182 (2011) 266–269

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

# Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters ☆

Chao-Tung Yang *, Chih-Lin Huang, Cheng-Fang Lin

*Department of Computer Science, Tunghai University, Taichung City, 40704, Taiwan*

ARTICLE INFO

*Article history:*
Received 1 March 2010
Received in revised form 18 June 2010
Accepted 25 June 2010
Available online 16 July 2010

*Keywords:*
CUDA
GPU
MPI
OpenMP
Hybrid
Parallel programming

ABSTRACT

Nowadays, NVIDIA's CUDA is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA to achieve dramatic speedups on production and research codes. In this paper, we propose a parallel programming approach using hybrid CUDA OpenMP, and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. Loop iterations assigned to one MPI process are processed in parallel by CUDA run by the processor cores in the same computational node.



## 1. Introduction

Nowadays, NVIDIA's CUDA [1] is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA [1] to achieve dramatic speedups on production and research codes.

This paper proposes a solution to not only simplify the use of hardware acceleration in conventional general purpose applications, but also to keep the application code portable. In this paper, we propose a parallel programming approach using hybrid CUDA, OpenMP and MPI [3] programming, which partition loop iterations according to the performance weighting of multicore [4] nodes in a cluster. Because iterations assigned to one MPI process are processed in parallel by OpenMP threads run by the processor cores in the same computational node, the number of loop iterations allocated to one computational node at each scheduling step depends on the number of processor cores in that node.

In this paper, we propose a general approach that uses performance functions to estimate performance weights for each node. To verify the proposed approach, a cluster with hybrid CUDA was built in our implementation. Empirical results show that in the hybrid CUDA clusters environments, the proposed approach improved performance over all previous schemes.

The rest of this paper is organized as follows. In Section 2, we introduce several typical and well-known parallel programming schemes. In Section 3, we define our model and describe our approach. Our system configuration is then specified in Section 4, and experimental results for three types of application program are presented. Concluding remarks and future work are given in Section 5.

## 2. Background review

### 2.1. CUDA programming

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing [2] architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. CUDA architecture supports a range of computational interfaces including OpenGL [9] and Direct Compute. CUDA's parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

☆ This work is supported in part by the National Science Council, Taiwan, under grants Nos. NSC 98-2220-E-029-004- and NSC 99-2220-E-029-004-.

* Corresponding author. Tel.: +886 4 23590415; fax: +886 4 23591567.
*E-mail address:* ctyang@thu.edu.tw (C.-T. Yang).

0010-4655/$ – see front matter 
doi:10.1016/j.cpc.2010.06.035