Mandelbrot Set

Valentino Ivanovski
UP FAMNIT
89211011@student.upr.si

This project report aims to investigate the rendering efficiency of the Mandelbrot set using various execution modes, including sequential mode (using a single processor thread), parallel mode (using multiple processor threads), and MPI (message passing between multiple single-threaded processes in a multicore configuration). Moreover, we will compare the rendering speed of using JavaFX `WritableImage` *with the speed of using buffers with* `BurfferedImage`*.*

1 Introduction

The Mandelbrot set is a mathematical set that consists of complex numbers represented on a complex plane. It is defined by a simple iterative algorithm called the **Escape Time Algorithm**:

$$z_{n+1} = z_n^2 + c \tag{1}$$

where z_n represents the complex number at iteration n with $z_0 = c$ and c represents a point on the complex plane.

The Escape Time Algorithm determines whether a given complex number c belongs to the Mandelbrot set. By repeatedly squaring the number z_n and adding c, the algorithm checks if the resulting sequence of numbers remains bounded. If the sequence remains bounded, c is considered part of the Mandelbrot set. Conversely, if the sequence diverges to infinity, c is not part of the set.

The colors assigned to the pixels on the complex plane are determined based on the magnitude of the numbers in the diverging sequence. Larger magnitudes typically correspond to more iterations and are often assigned different colors, allowing us to visually represent the intricate structures and boundaries of the Mandelbrot set.

Algorithm 1 Escape Time Algorithm

```
1:  function EscapeTime(c, totalIter)
2:      z ← 0
3:      i ← 0
4:      while i < totalIter and |z| < 2 do
5:          z ← z^2 + c
6:          i ← i + 1
7:      end while
8:      if i = totalIter then
9:          return −1                          ▷ Point is in the set
10:     else
11:         return i                           ▷ Point is outside the set
12:     end if
13: end function
```

Since every pixel is drawn independently, this algorithm can easily be parallelized, and adapted to use multiple threads, processes and machines. Thus, the rest of the paper will describe the methods used for parallelization, and the measured performance results for different modes of execution.

2 Implementation

In order to visualize the Mandelbrot set using the above-mentioned algorithm, we use JavaFX. The Mandelbrot set will be shown on a `WritableImage`, with each pixel assigned a unique color based on its behavior in the complex plane, as determined by the Escape Time Algorithm.

To parallelize the computation of the escape time algorithm, a common approach is to distribute the work among multiple workers. This involves dividing the image into sections and assigning each section to a worker. Each worker then independently calculates the escape time for the pixels within its assigned section, allowing for concurrent processing. However, when implementing parallelization, it is crucial to be aware of potential race conditions and synchronization issues. Since multiple workers are simultaneously accessing and modifying shared data (such as the calculated escape times or pixel values), conflicts can arise, leading to incorrect results or data corruption. To address these issues, we will implement appropriate synchronization mechanisms, such as `CountdownLatch` which ensures that all worker threads have finished calculating the escape time for their assigned section before proceeding to merge the results. This topic is furthermore discussed in section 5.

For the message passing interface, we will utilize the MPJ Express library, which allows us to use Java in combination with MPI. Here, we are using a `BufferedImage` from `awt` to display the Mandelbrot set, and converting it into a byte array such that message passing is possible.

Further explanation of the sequential, parallel and distributive implementations and how they work is written in the following section.

3 Mode description

Sequential mode: In this mode, the rendering of the Mandelbrot set is performed using a single thread. While this approach is simpler to implement, it comes with certain drawbacks. Compared to the parallel and distributive modes, the single-threaded mode is significantly slower in terms of performance.

The main advantage of the single-threaded mode is that it places the least amount of strain on the computer processor. By utilizing only one thread, the computational workload is handled sequentially, allowing the processor to focus its resources on one task at a time. This can be beneficial in scenarios where the computational resources are limited, or when minimizing processor usage is a priority.

However, the trade-off for reduced strain on the processor is decreased performance. Since the calculations are performed sequentially, the single-threaded mode takes longer to render the Mandelbrot set compared to parallel or distributive modes, where multiple threads or distributed processing are employed.

Parallel mode: In the parallel mode, our project takes advantage of the capabilities of worker threads, which are coordinated by a central main method. To accomplish this, we have implemented a dedicated thread class that provides specific instructions for each thread, dividing the workload and enabling them to collaborate on their assigned tasks. By combining the outcomes of these parallel computations, we achieve a synergistic effect that outperforms the sequential mode.

One of the significant advantages of parallel programming is its speed. By simultaneously employing multiple instances of the same computational process, we tap into the collective processing power available. This results in significantly faster execution times compared to the sequential mode, where tasks are executed one after another. As a result, parallel programming allows us to fully exploit the potential of modern hardware architectures and maximize overall efficiency.

Nevertheless, it is important to consider that a major drawback of parallel programming is the potential for race conditions and synchronization issues. When multiple threads or processes are executing concurrently and accessing shared resources or modifying shared data, there is a risk of unintended interactions and conflicts.

Distributive mode: In the distributive mode, our project utilizes a message passing interface (MPI) to establish communication and coordination between different processes. This approach allows us to distribute the workload across multiple nodes or machines, enabling parallel processing on a larger scale. The MPI provides a standardized protocol for exchanging messages and synchronizing the execution of tasks among these distributed processes.

One major advantage of the distributive mode with an MPI is its ability to handle large-scale computations. By distributing the workload across multiple nodes or machines, we can take advantage of the combined processing power and memory resources, resulting in faster and more efficient execution. This makes the distributive mode suitable for tackling complex and computationally intensive tasks that require significant computational resources.

However, there is a notable drawback associated with the distributive mode and the use of an MPI: communication overhead. As the number of processes and the complexity of the communication between them increases, so does the amount of time and resources required for message passing. This overhead can potentially impact the overall performance and efficiency of the distributed computation. Careful consideration must be given to optimizing the communication patterns and minimizing unnecessary data exchanges to mitigate this drawback.

In this case, we will be implementing the message passing interface in multi-core mode, which is famously used to test MPI projects on single machines.

With that being said, we can start with the presentation of the parallel implementation. We will not be discussing the sequential mode, since it does not require any special parallelism techniques - hence its name.

4 Division of workload

In the context of distributing work among multiple workers, as previously mentioned in section 2, we now delve into the practical implementation of this process. Implementing this distribution of work is straightforward, involving the use of a simple for loop. With the help of this loop, we can effectively create sections and distribute the workload among the available threads.

```
for (int i = 0; i < n; i++) {
            int begin = i * section; //compute the start point for each section
            int end = begin + section;

            //creating a service for each section of the image
            MandelbrotService service = new MandelbrotService(begin, end);
            service.setOnSucceeded(event -> {
                latch.countDown(); //decrease the latch count when the service
                    finishes
                service.cancel(); //cancel the service
            });
            service.start(); //start the service
            services.add(service);// add the service to the services list
        }
```

Additionally, in the thread class, we would like to change the for loop variables to match the sections:

```
    for (int y = begin; y < end; y++) {
                        for (int x = 0; x < scaledLocalWidth; x++) {
                            \\*code that renders the Mandelbrot set*
                        }
                    }
```

Note the `setOnSucceeded` method. It is invoked to specify the action to be taken when the service completes its task. In this case, the `latch.countDown()` operation decreases the count of the `CountDownLatch`, indicating the completion of a section's computation. Additionally, the `service.cancel()` method cancels the service to free up system resources. This method is related to the next section, where we discuss synchronization problems.

5 Synchronization problems

When we are implementing parallelism in our program, it is crucial to address potential synchronization issues that may arise when multiple threads attempt to modify shared data concurrently. Specifically, we aim to prevent scenarios where one thread modifies the shared data while another thread is actively using it. Such unwanted synchronization can lead to data inconsistency and ruin the correctness of our program.

To tackle this challenge, we have thought of a solution by incorporating a `CountdownLatch` into our implementation. The `CountdownLatch` is also referred to as *concurrency control*, allowing us to enforce proper thread safety and coordination within our parallel program.

By implementing the `CountdownLatch`, we establish a mechanism that ensures "exclusive access" to the shared data. When one thread is already actively utilizing the data, other threads are prevented from modifying it until the initial thread has completed its task. This CountdownLatch safeguards against simultaneous modifications and guarantees that each thread operates on the shared data individually.

```
1  public void MandelbrotSet(int n) {
2      // ...
3      List<MandelbrotService> services = new ArrayList<>();
4      CountDownLatch latch = new CountDownLatch(n);
5
6      for (int i = 0; i < n; i++) {
7          // ...
8      }
9
10     // ...
11
12     // Thread to wait for all the services to complete their tasks
13     Thread waitForCompletion = new Thread(() -> {
14         try {
15             latch.await(); // Wait until the countdown latch counts down to
                   zero
16         } catch (InterruptedException e) {
17             e.printStackTrace(); // Print the stack trace if an
                   InterruptedException occurs
18         }
19
20         // This code will run after all the services complete their tasks
21         Platform.runLater(() -> {
22             // *code that merges the work of the individual threads*
23         });
24     });
25
26     waitForCompletion.start(); // Start the waitForCompletion thread
27 }
```

The `CountdownLatch` operates by initializing a count equal to the number of threads involved in the parallel program. As each thread completes its work of the shared data, it signals the `CountdownLatch` by decrementing the count. This allows the next waiting thread to proceed and begin its execution, ensuring a sequential and synchronized flow of operations.

The implementation of the `CountdownLatch` eliminates the unwanted synchronization issues encountered in our parallel program. It establishes controlled environment where only one thread at a time can modify the shared data, preventing conflicts and ensuring data integrity.

One more important thing to note is that JavaFX does not allow modifying the GUI components from a background thread. Specifically, modifications to GUI components should be done on the main application thread. As a result, in this program, the background threads are responsible for computing whether the points of the complex plane belong to the Mandelbrot set, while the display of these points is handled on the main application thread using the `Platform.runLater()` method.

6 Message passing interface

In order to use the message passing interface in Java, we employ the MPJ Express library. However, this library is not compatible with the latest versions of JavaFX. As a result, we rely on the `BufferedImage` from `awt`. In this mode, we follow the same division method discussed earlier, focusing on message passing and data conversion.

MPI works by sending and receiving data between computers or nodes. In our case, we are using our computer threads as nodes, and we are sending and receiving data between them. There exists a main node that is collecting all of the data from the rest of the nodes, and concatinates it into a final image. In the code below, the main node is referring to `processNumber == 0`.

In order to send and receive data, we use the methods `Send()` and `Recv()`. The `Send()` method expects the data to be in the form of a byte array. This means that the `BufferedImage`, that we want to send to another node, in order to be compatible with the `Send()` method, it needs to be in the form of a byte array.

The byte array representation allows for a compact and efficient transmission of data. It converts the image data into a sequence of bytes, which can then be sent over the MPI communication channel. Because of this, we can realize a significant speedup of the program, compared to the other modes that use the JavaFX `WritableImage` to display the Mandelbrot set.

```
1     public static void main(String[] args) throws MPIException, IOException {
2     MPI.Init(args);
3     int processNumber = MPI.COMM_WORLD.Rank();
4     int totalProcesses = MPI.COMM_WORLD.Size();
5
6     int section = heightImage / totalProcesses;
7     int begin = processNumber * section;
8     int end = begin + section;
9
10    // Generate a section of the Mandelbrot set
11    BufferedImage mandelbrotImageSection = MandelbrotSet(begin, end, section);
12
13    if (processNumber == 0) {
14        BufferedImage mandelbrotImage = new BufferedImage(widthImage,
              heightImage, BufferedImage.TYPE_INT_ARGB);
15        Graphics2D imageGraphics = mandelbrotImage.createGraphics();
16
17        imageGraphics.drawImage(mandelbrotImageSection, 0, 0, null);
18
19        for (int i = 1; i < totalProcesses; i++) {
20            byte[] receivedData = new byte[widthImage * section * Integer.BYTES
                  ];
21            MPI.COMM_WORLD.Recv(receivedData, 0, receivedData.length, MPI.BYTE,
                   i, 0);
22            BufferedImage receivedImage = toBufferedImage(receivedData,
                  widthImage, section);
23            imageGraphics.drawImage(receivedImage, 0, i * section, null);
24        }
25        imageGraphics.dispose();
26
27        // Save final image
28        ImageIO.write(mandelbrotImage, "png", new File("mandelbrot.png"));
29    } else {
30        byte[] sendData = toByteArray(mandelbrotImageSection);
31        MPI.COMM_WORLD.Send(sendData, 0, sendData.length, MPI.BYTE, 0, 0);
32    }
33
34    MPI.Finalize();
35 }
```

7 Comparison

In terms of the comparison and results, it is expected that the sequential mode would have the slowest execution time since it operates on a single thread and it uses the `WritableImage`. However, when comparing the MPI implementation in multi-core mode to the parallel mode, the difference should not be significant. This is because the nodes in the MPI implementation correspond to the same computer threads used in the parallel mode. The distinction lies in the fact that, in the MPI implementation, these nodes send and receive data as if they were separate machines, introducing communication overhead that slows down execution.

In our specific case, utilizing BufferedImage and working with byte arrays instead of WritableImage, the execution time in the distributive mode is actually faster than in the parallel mode. It is important to note that this speed advantage in the distributive mode is attributed to the use of buffers. Generally speaking, if we were to employ writableimage with MPI, the **distributive mode would be slightly slower** than the parallel mode due to the communication overhead.

A comprehensive test will be conducted to evaluate the performance of three different modes. The test involves saving images of the Mandelbrot set on the disk at various resolutions, as the runtime is influenced by the width and height of the resulting image. The runtime of each image generation will be measured and presented in a bar graph such that it can be easily analysed.

The test will begin with an image resolution of 1000x1000 pixels, and subsequently, both the width and height will be increased by 1000 pixels for each subsequent configuration. The configurations will range from 1000x1000 to 4000x4000 pixels. To simplify the presentation, we will refer to the configurations as 1000, 2000, 3000, and 4000.

The specific region of the Mandelbrot set shown below will serve as the test case for all modes:

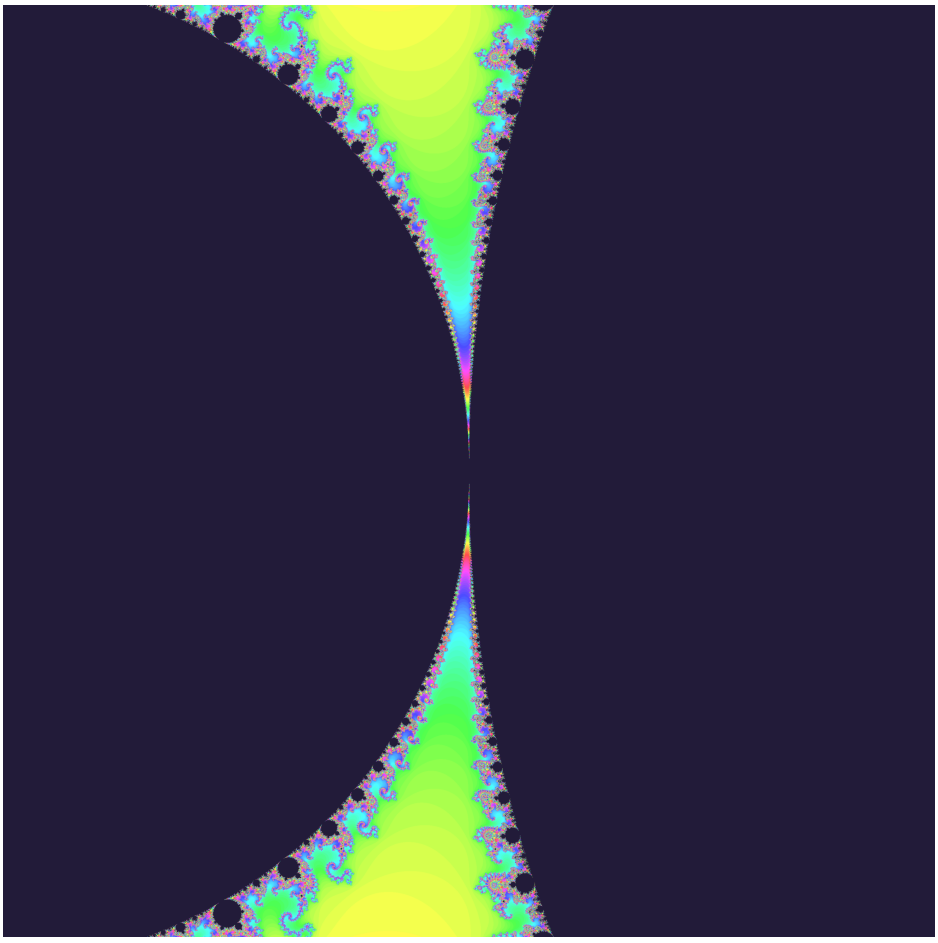
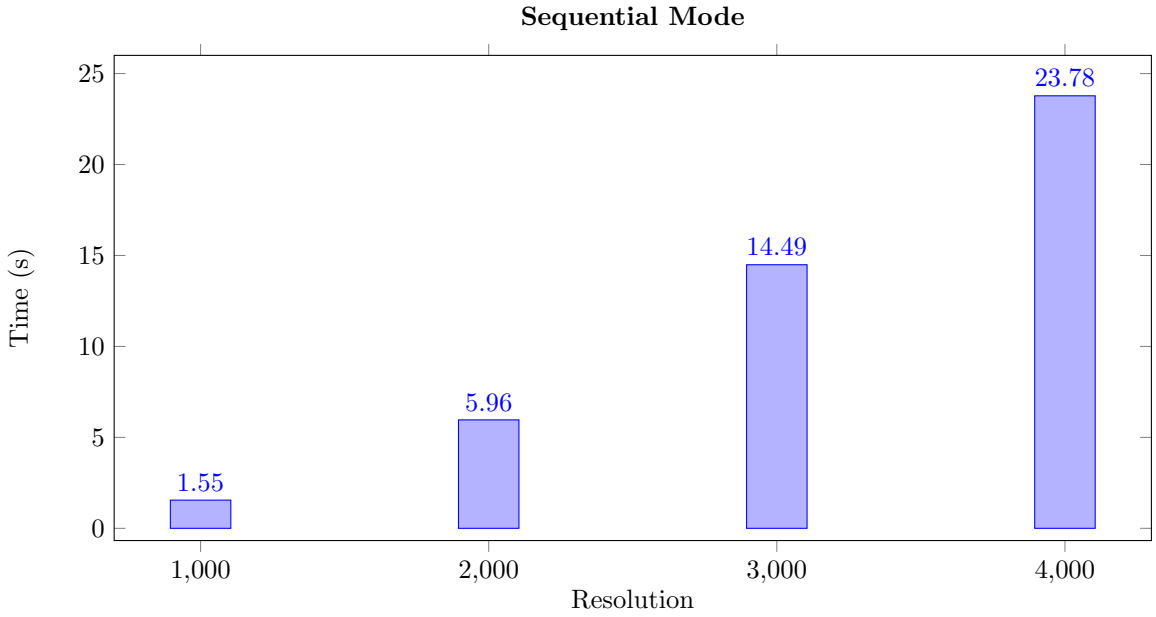
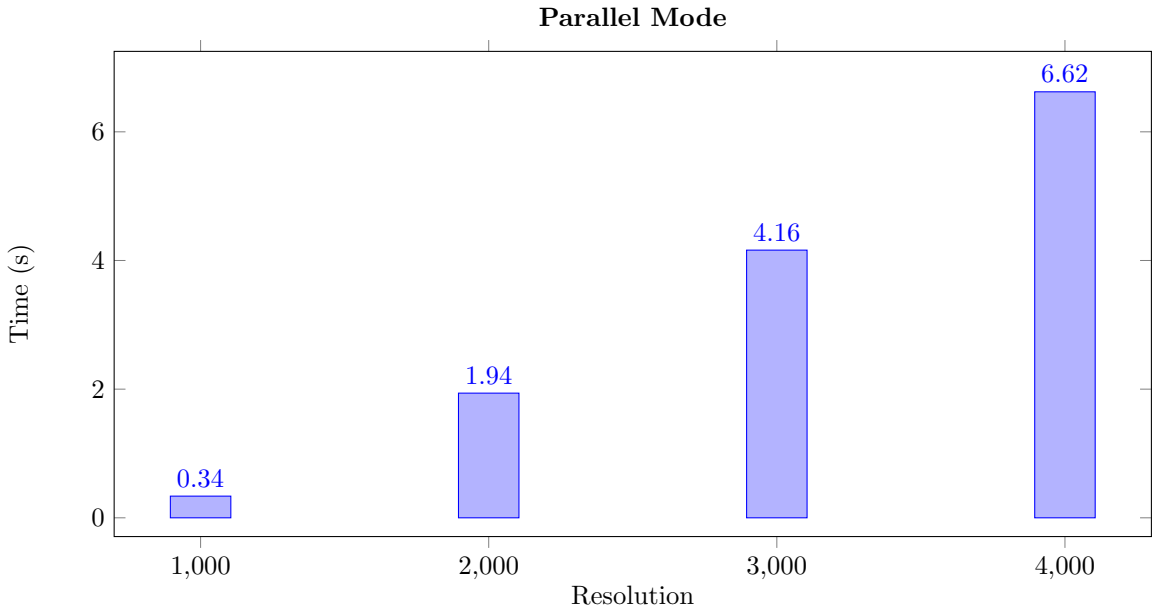
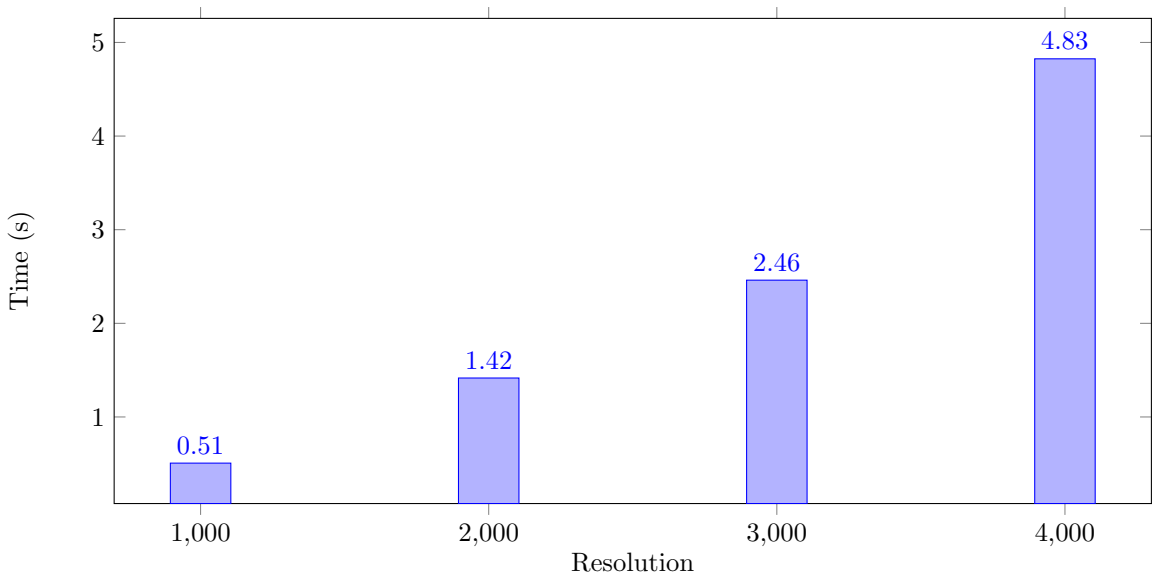


Figure 1: Part of the set where c=-3/4


Sequential Mode
Time (s)
25
20
15
10
5
0
1.55
5.96
14.49
23.78
1,000
2,000
3,000
4,000
Resolution


Parallel Mode
Time (s)
6
4
2
0
0.34
1.94
4.16
6.62
1,000
2,000
3,000
4,000
Resolution


Distributive Mode
Time (s)
5
4
3
2
1
0.51
1.42
2.46
4.83
1,000
2,000
3,000
4,000
Resolution

8 Results and Conclusion

The experiment was conducted on a laptop using an Apple M1, 8-core chip. As mentioned before, the program was ran on three different modes: Sequential, Parallel, and Distributive. In the Distributive mode, only one process per node was used, and the image was displayed with a color profile generated by the hue of different colors, using 500 iterations.

The results clearly demonstrate the significant speed advantage of both the Parallel and Distributive modes over the Sequential mode. Note that, when dealing with smaller resolution images, the impact of communication overhead becomes clear when comparing the execution times of the 1000x1000 resolution for each mode.

For the smaller image, the Parallel mode shows faster performance initially, but its speed gradually diminishes as the resolution increases. On the other hand, the Distributive mode initially lags behind in terms of speed for the smaller image, but then progressively speeds up due to the efficiency of the buffers being used.

These remarks highlight the influence of communication overhead in distributive systems. By this we conclude that such systems are more suitable for handling computations where their benefits outweigh the communication overhead limitations encountered in smaller-scale computations.