**Robil Sabek**

**Mandelbrot set normal code:**

This code uses NumPy and Matplotlib to generate and display an image of the Mandelbrot set. The width and height variables control the size of the image, and the xmin, xmax, ymin, and ymax variables define the rectangular region of the complex plane to be plotted. The max\_iter variable controls the maximum number of iterations to perform for each point in the complex plane. The resulting image is displayed using Matplotlib's imshow function.

**Mandelbrot set static code:**

Visualizing the Mandelbrot set involves plotting the complex plane and coloring points based on whether or not they belong to the set.

To visualize the Mandelbrot set using dynamic task assignment, you can use a parallel computing approach. The idea is to divide the visualization task into smaller sub-tasks that can be independently executed by different threads or processes. Each thread or process is assigned a sub-task dynamically based on the available resources and the workload of each sub-task. This approach allows for better utilization of the available resources and faster visualization times.

Here is a general outline of the steps involved in visualizing the Mandelbrot set using dynamic task assignment:

1. Divide the complex plane into smaller sub-regions.

2. Assign each sub-region to a thread or process dynamically based on the available resources and workload of each sub-region.

3. For each sub-region, iterate through the Mandelbrot equation for each point in the subregion to determine whether or not it belongs to the set.

4. Color each point in the sub-region based on whether or not it belongs to the set.

5. Merge the results from each sub-region to produce the final visualization.

The speed-up factor, efficiency, computation to communication ratio, and scalability of using static task assignment for generating the Mandelbrot set depends on the number of tasks, the size of each task, and the number of available processors.

Assuming that the workload is evenly distributed among the processors, the speed-up factor of static task assignment is given by:

Speed-up factor = (total number of tasks)/(number of processors)

For example, if there are 1000 tasks and 4 processors, the speed-up factor is 250.

The efficiency of static task assignment is defined as the ratio of the speed-up factor to the number of processors, and is given by:

Efficiency = (total number of tasks)/(number of processors)^2

For the same example with 1000 tasks and 4 processors, the efficiency is 62.5%.

The computation to communication ratio is the ratio of the time spent on computation to the time spent on communication. In the case of static task assignment, the communication overhead is minimal, as each processor only needs to communicate the results of its tasks to the master process. Therefore, the computation to communication ratio is very high.

The scalability of static task assignment is limited by the size of the tasks and the number of available processors. If the size of the tasks is too small, the communication overhead may become significant, and the efficiency may decrease. On the other hand, if the size of the tasks is too large, some processors may finish their tasks much earlier than others, leading to load imbalance and decreased efficiency. Therefore, the optimal size of the tasks and the number of processors depend on the specific application and hardware platform.

**Mandelbrot set parallel code:**

In this code, the mandelbrot\_set function divides the complex plane into a grid of tasks using nested loops and assigns each task to a process using the multiprocessing. Pool class. The mandelbrot function computes the Mandelbrot set for each task. The resulting Mandelbrot set is stored in a two-dimensional NumPy array of integers. The write\_to\_file function writes the Mandelbrot set to a file. The elapsed time is measured using the current\_process().\_bootstrap\_inner() method, which returns the time in seconds since the start of the process. The number of processes is obtained using the cpu\_count function, which returns the number of available CPUs. Note that the \_bootstrap\_inner() method is used to measure the time because it is not affected by the overhead of starting and joining processes.

The speed-up factor, efficiency, computation to communication ratio, and scalability of using parallel task assignment for generating the Mandelbrot set depend on the number of tasks, the size of each task, the number of available processors, and the communication overhead.

Assuming that the workload is evenly distributed among the processors, the speed-up factor of parallel task assignment is given by:

Speed-up factor = (total number of tasks)/(number of processors)

For example, if there are 1000 tasks and 4 processors, the speed-up factor is 250.

The efficiency of parallel task assignment is defined as the ratio of the speed-up factor to the number of processors, and is given by:

Efficiency = (total number of tasks)/(number of processors \* time + communication overhead)

Here, the communication overhead includes the time spent on exchanging messages between the processors. The efficiency depends on the size of the tasks, the communication overhead, and the number of processors. In general, the efficiency decreases as the communication overhead increases and as the number of processors increases. If the size of the tasks is too small, the communication overhead may become significant, and the efficiency may decrease.

The computation to communication ratio is the ratio of the time spent on computation to the time spent on communication. In the case of parallel task assignment, the communication overhead can be significant, as each processor needs to communicate with other processors to exchange data and synchronize the computation. Therefore, the computation to communication ratio may be lower than in the case of static task assignment.

The scalability of parallel task assignment is limited by the communication overhead, the load imbalance, and the memory bandwidth. If the communication overhead becomes too high, the efficiency may decrease, and the scalability may be limited. If the load imbalance is significant, some processors may finish their tasks much earlier than others, leading to wasted resources and decreased efficiency. Finally, if the memory bandwidth becomes a bottleneck, the performance may be limited by the rate at which data can be transferred between the processors and the memory. Therefore, the optimal number of processors and the size of the tasks depend on the specific application and hardware platform.

**Conclusion:**

Parallel task assignment divides the workload into smaller tasks that can be processed in parallel by multiple processors. This approach can potentially achieve higher speed-up factors and better scalability than static task assignment. However, it also incurs higher communication overhead and requires more sophisticated programming techniques.

Static task assignment divides the workload into a fixed number of tasks that are assigned to each processor before the computation starts. This approach is simpler and incurs lower communication overhead than parallel task assignment, but it may suffer from load imbalance and lower scalability.

In general, the optimal choice between parallel task assignment and static task assignment depends on the specific hardware and application requirements. Here are some general considerations:

Speed-up factor: Both parallel task assignment and static task assignment can achieve high speed-up factors, but parallel task assignment may potentially achieve higher speed-up factors due to better load balancing and resource utilization.

Efficiency: Efficiency is a measure of how well the algorithm scales with the number of processors. Parallel task assignment may potentially achieve higher efficiency than static task assignment, but this depends on the communication overhead and the workload distribution.

Computation to communication ratio: The computation to communication ratio measures the ratio of time spent on computation versus communication. Static task assignment may have a higher computation to communication ratio because it incurs less communication overhead than parallel task assignment.

Scalability: Scalability measures how well the algorithm performs as the problem size or number of processors increase. Both parallel task assignment and static task assignment can scale to a certain extent, but their scalability depends on the workload distribution, communication overhead, and hardware constraints.

In general, parallel task assignment is more suited for problems with large workloads and a high degree of parallelism, while static task assignment may be more suited for problems with smaller workloads or a fixed number of processors. However, the optimal choice depends on the specific requirements and constraints of the application.