CompletableFuture

To improve performance when calling multiple REST services in Java, you can use non-blocking asynchronous calls. One of the most efficient ways to achieve this is by leveraging `CompletableFuture` along with a non-blocking HTTP client such as `HttpClient` from Java 11 onward.

Here's how you can implement non-blocking calls to multiple REST services:

Step-by-Step Guide:

1. **Use Java 11's** `HttpClient`: This HTTP client supports asynchronous, non-blocking requests out of the box using `CompletableFuture`.
2. **Make Asynchronous Calls**: Use `CompletableFuture` to fire off multiple HTTP requests concurrently.
3. **Combine Results**: Wait for all the responses using `CompletableFuture.allOf()` to process them once all requests are completed.

Example Code:

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.List;
import java.util.ArrayList;

public class AsyncRestCallExample {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        // Initialize HttpClient
        HttpClient client = HttpClient.newHttpClient();

        // List of URLs to call
        List<String> urls = List.of(
                "<https://jsonplaceholder.typicode.com/posts/1>",
                "<https://jsonplaceholder.typicode.com/posts/2>",
                "<https://jsonplaceholder.typicode.com/posts/3>"
        );

        // List to hold the CompletableFuture for each asynchronous call
        List<CompletableFuture<HttpResponse<String>>> futures = new ArrayList<>();

        // Iterate over the URLs and make asynchronous requests
        for (String url : urls) {
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(url))
                    .GET()
                    .build();

            // Send the request asynchronously and store the CompletableFuture
            CompletableFuture<HttpResponse<String>> future = client.sendAsync(request,
HttpResponse.BodyHandlers.ofString());
            futures.add(future);
        }
```

```
37
38          // Wait for all requests to complete
39          CompletableFuture<Void> allFutures = CompletableFuture.allOf(futures.toArray(new
    CompletableFuture[0]));
40
41          // Once all are completed, process the results
42          allFutures.thenRun(() -> {
43              futures.forEach(future -> {
44                  try {
45                      HttpResponse<String> response = future.get();
46                      System.out.println("Response: " + response.body());
47                  } catch (InterruptedException | ExecutionException e) {
48                      e.printStackTrace();
49                  }
50              });
51          }).join(); // Blocks until all tasks are completed
52
53          System.out.println("All REST service calls completed.");
54      }
55  }
```

Explanation:

1. `HttpClient`: This is the new non-blocking HTTP client introduced in Java 11. It sends HTTP requests asynchronously.
2. `CompletableFuture`: Used to send HTTP requests asynchronously. Each request returns a `CompletableFuture` that will be completed when the response is received.
3. `CompletableFuture.allOf()`: This method waits for all the `CompletableFuture` objects in the list to complete. Only then does it continue with the response processing.
4. **Non-blocking I/O**: Since the requests are sent asynchronously, the main thread is not blocked while waiting for the responses, improving performance significantly.

Improvements Over Blocking:

- **Concurrency**: Instead of sending requests one by one, all requests are fired concurrently, reducing overall waiting time.
- **Non-blocking**: The program can perform other tasks while waiting for responses from the services, making it more efficient.

Conclusion:

Using asynchronous non-blocking calls via `CompletableFuture` and `HttpClient` greatly improves the performance of calling multiple REST services simultaneously by reducing wait times and making better use of system resources.

Let me know if you'd like to explore further optimizations or enhancements!