PMC Lecture 06

Gianmaria Romano

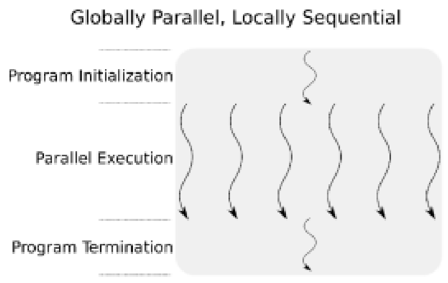
21 October 2025

Chapter 2

Message passing interfaces

2.1 Parallelizing message passing interfaces

Generally speaking, it is possible to exploit common patterns in an application in order to parallelize the programs it wants to run.
In particular, an application is said to be ”globally parallel, locally sequential” if it can perform sequential tasks concurrently.

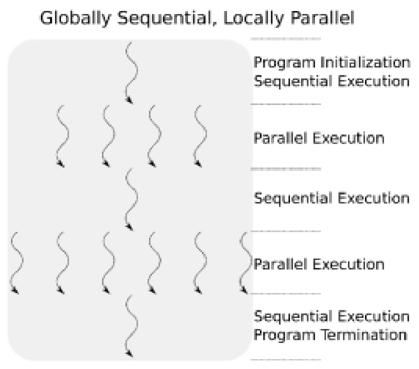


This design pattern is typically implemented according to one of the following patterns:

- **Single program, multiple data:** this pattern keeps all the application logic in a single program that can be executed by multiple processes whose specific functionalities are specified based on their rank.
  While this is a relatively simple approach, it fails when the memory requirements are too high for each node or if different architectures need to be used.
- **Multiple program, multiple data:** this pattern tries to overcome the limitations of the single program approach by allowing processes to run on different programs.

- **Master-worker:** this pattern involves a master process, which distributes the tasks among the other processes and collects their results, and one, or more, worker processes that perform computations.
- **Map-reduce:** this approach acts as a variation of the master-worker pattern where the workers can either perform a map, which consists in working on the data to achieve a partial result, or a reduce, which consists in collecting the partial results to get the final one.

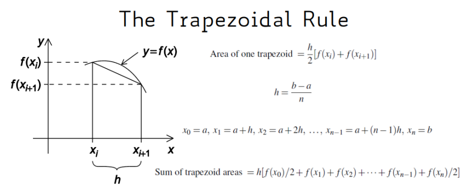
Alternatively, an application is said to be "globally sequential, locally parallel" if it runs sequentially but, whenever needed, it can refer to other processes/threads to perform parallel computation.



This design pattern is typically implemented according to one of the following patterns:

- **Fork/join:** this pattern features a single parent thread that starts the execution and, when needed, relies the task to child threads that can be created dynamically or, to achieve better efficiency, be taken from a static thread pool.
- **Parallel loops:** this pattern consists in distributing iterative loops among threads that will work in parallel.
  While parallelizing a loop is a very simple strategy, this approach is less flexible as it typically requires the loop to have particular properties.

Example: Suppose you want to parallelize the trapezoidal rule for approximating the area of a function $f(x)$ over an interval of length h:



$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

Assuming n processes are used, the general idea would be to have each process parallelly compute the area of a trapezoid and, at the end, a master process should add the partial sums to obtain the final result.

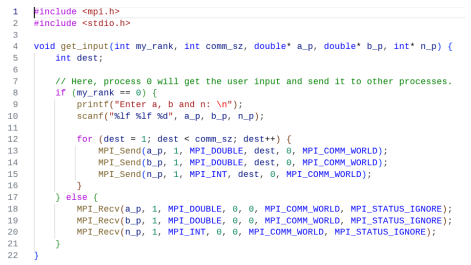
```c
#include <mpi.h>
#include <stdio.h>

// Declare a function to compute the area of each trapezoid.
double Trap(double left_endpt, double right_endpt, int trap_count, double base_len);

int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    h = (b - a) / n; // All processes use the same height.
    local_n = n / comm_sz; // Define the number of trapezoids.

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    // Here, process 0 will gather and sum the partial integrals computed by the other processes.
    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    } else {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }

    if (my_rank == 0) {
        printf("With n = %d trapezoids, the estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    }

    MPI_Finalize();
    return 0;
}

double Trap(double left_endpt, double right_endpt, int trap_count, double base_len) {
    double estimate, x;
    int i;

    // Assume f is a well-known function defined a priori.
    estimate = (f(left_endpt) + f(right_endpt)) / 2.0;
    for (i = 1; i <= trap_count - 1; i++) {
        x = left_endpt + i * base_len;
        estimate += f(x);
    }
    estimate = estimate * base_len;

    return estimate;
}
```

Overall, however, this approach tends to be inefficient as most of the workload will be left to the master process.

2.2 Reading user inputs in message passing interfaces

Generally speaking, user inputs for message passing interfaces are handled by the ”master” process, which, after reading the input, is able to send it to the

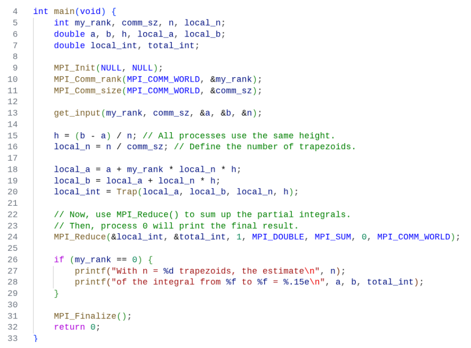
other processes.
Example: It is possible to make the parallel implementation of the trapezoidal rule more flexible by allowing users to specify the number of trapezoids they want to use through a dedicated function that reads the user inputs and broadcasts them to the working processes:



```c
#include <mpi.h>
#include <stdio.h>

void get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p) {
    int dest;

    // Here, process 0 will get the user input and send it to other processes.
    if (my_rank == 0) {
        printf("Enter a, b and n: \n");
        scanf("%lf %lf %d", a_p, b_p, n_p);

        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

2.3 Collective communication

One of the main issues of parallel communication in message passing interfaces lies in the fact that, generally speaking, most of the workload is left to the master process.
For this reason, it can come in handy to make communication more efficient by implementing collective operations that allow more processes to participate to the same routine simultaneously in order to perform (or even combine) operations like reductions or broadcasting.
Keep in mind, however, that, to avoid hanging or crashing situations, each process in the communicator should always refer to the same collective function.
Example: The following code tries to improve the parallel implementation of the trapezoidal rule through collective communication:



```c
int main(void) {
    int my_rank, comm_sz, n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    get_input(my_rank, comm_sz, &a, &b, &n);

    h = (b - a) / n; // All processes use the same height.
    local_n = n / comm_sz; // Define the number of trapezoids.

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    // Now, use MPI_Reduce() to sum up the partial integrals.
    // Then, process 0 will print the final result.
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("With n = %d trapezoids, the estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
    }

    MPI_Finalize();
    return 0;
}
```

In fact, it is possible to observe that, if sending and receiving operations take place at the same time, this algorithm is two times faster compared to the original implementation.