PMC Lecture 08

Gianmaria Romano

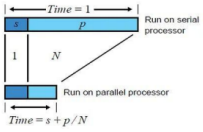
28 October 2025

Chapter 2

Message passing interfaces

2.1 Estimating the speed-up factor of a parallel application

Generally speaking, each program features a "serial fraction" that cannot be parallelized, resulting in a limit to the speed-up factor achieved by parallelizing the application.



In fact, it is possible to reformulate the running time of a parallel application using p processes in the following way:

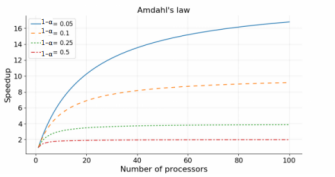
$T_{par}(p) = (1-\alpha)T_{seq} + \alpha\dfrac{T_{seq}}{p}$, where $0 \leq \alpha \leq 1$ is the parallelizable fraction.

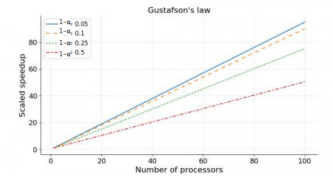
Most particularly, Amdahl's law exploits this formulation in order to provide an asymptotic estimate of the speed-up factor as follows:

$$S(p) = \frac{T_{seq}}{(1-\alpha)T_{seq} + \alpha\frac{T_{seq}}{p}} \text{ and take } \lim_{p\to\infty} S(p) = \frac{1}{1-\alpha}$$

Generally speaking, however, Amdahl's law lies can only be used to explain the speed-up factor of strong scaling applications.
In fact, in the context of weak scaling applications, it is possible to observe that the parallel fraction tends to increase with the problem's size, meaning that the speed-up factor should be reformulated as follows:

$S(n,p) = (1-\alpha) + \alpha p$, where $0 \leq \alpha \leq 1$ is the parallelizable fraction.





Example: Suppose you want to parallelize the sum of two vectors $\vec{x}$ and $\vec{y}$. In the context of message passing interfaces, the master process can use a scatter to read and distribute pieces of the vector to the other working processors, which compute a partial sum each.

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

// Try to parallelize the computation of the sum.
void parallel_vector_sum(double local_x[], double local_y[], double local_z[], int local_n) {
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++) {
        local_z[local_i] = local_z[local_i] + local_y[local_i];
    }
}

// Use MPI_Scatter() to read a vector and distribute it among the working processes.
void read_vector(double local_a[], int local_n, int n, char vec_name[], int my_rank, MPI_Comm comm) {
    double* a = NULL;
    int i;

    // Here, process 0 reads the input vector and scatters it among the other processes.
    if (my_rank == 0) {
        a = malloc(n * sizeof(double));
        printf("Enter the vector %s\n", vec_name);
        for (i = 0; i < n; i++) {
            scanf("%lf", &a[i]);
        }
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
        free(a);
    } else {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);
    }
}
```

Then, the master process can perform a gather in order to collect the partial sums and recover the final result.

```c
void print_vector(double local_b[], int local_n, int n, char title[], int my_rank, MPI_Comm comm) {
    double* b = NULL;
    int i;

    // Here, process 0 will gather and print all components of the vector.
    if (my_rank == 0) {
        b = malloc(n * sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++) {
            printf("%f ", b[i]);
        }
        printf("\n");
        free(b);
    } else {
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    }
}
```

2.2 Derived datatypes

Derived datatypes are used to simulate structs by storing the type and memory location of each item in the collection.

Therefore, if a function that needs to send data knows this information about a collection, it will be able to collect the items before they are sent.
Similarly, a function that receives data is able to collect the items from the memory soace before they are received.