Parallel Programming
Exercise sheet 6: Message Passing Interface (MPI)

1 A simple example: matrix-vector multiplication

The matrix-vector multiplication proves to be one of the most commonly used building blocks for numerical algorithms. Therefore, and efficient implementation of this operation can be decisive for the efficiency of the entire algorithm. We are concerned here with the parallelization of the computation needed for the multiplication matrix-vector. Employ different MPI constructs for the parallel computation of the matrix-vector product. Discuss the different implementation alternatives from the perspective of performance.

2 Deadlocks: blocking vs. non-blocking

Given is a uni-directional ring network with $n$ processing nodes, thus, one processing node $n_i$ can only send messages to its successor $n_{i+1}$ and receive messages from its predecessor $n_{i-1}$. All processing nodes now want to communicate in parallel – first sending a message to their successor before receiving a message from their predecessor. If done in the described manner, there shouldn’t arise any complications. But what happens if we change the order?

Now every processing node first wants to receive a message from its predecessor before sending a message to its successor. We need a short message printed to the console. When finished, every processing node should confirm with a short message printed to the console.

3 Collective communication

Collective communication is used when data of one processing node should be distributed among all processing nodes – a scatter – or when data from all processing nodes should be collected in one processing node – a gather. A collective communication can also be bundled with an operator for manipulating the distributed data. Within MPI exist several operators like finding the minimum/maximum value, calculating the sum/product of all values, executing binary operations, and much more.

Write an MPI-Program collective.c to compare the built-in MPI_Reduce() command with your own implementation of that function. Therefore, write a function reduce() to collect distributed data from all processing nodes in one processing node, such that the different operators minimum, maximum, sum, and product can be applied to your function—for reasons of simplicity you can restrict your function only to work with integers. Think also of intelligent strategies of collecting the distributed data! A point-to-point connection from the calling processing node to all other processing nodes will take too much time, some more sophisticated connections – e.g. a binary tree – might increase the amount of communications but decrease the overall execution. Determine the time for both implementations, the built-in MPI command MPI_Reduce() and your own implementation reduce(), and discuss your results!

4 The sieve of Eratosthenes

A versatile scholar, Eratosthenes of Cyrene (today Libya) lived approximately 275—195 BC. He was the first to estimate accurately the diameter of the earth. For several decades, he served as the director of the famous library in Alexandria. He was highly regarded in the ancient world, but unfortunately only fragments of his writing have survived. Eratosthenes died at an advanced age from voluntary starvation, induced by despair at his blindness.

Eratosthenes also conceived the “Sieve of Eratosthenes”, a method of identifying prime numbers. Therefore, first the set of all natural numbers $P = \{2, \ldots, n\}$ from 2 to an upper limit $n$ is formed. Then, after searching for the minimum $p$ of set $P$ – the first prime number – all multiples of $p$ from
$P$ are deleted. Successively repeating this step until $p = \lfloor \sqrt{n} \rfloor$ there are only prime numbers left in $P$.

Write an MPI-program `sieve.c` following the algorithm described above to find all prime numbers in a set $P = \{2, \ldots, n\}$. For a faster processing, the main work – namely to delete all multiples of a prime number – should be done in parallel. Thus, the set $P$ has to be divided into equal parts $P_i$ and distributed to all processes $i \in [0, \text{max\_processes}]$. Each process $i$ searches for its local minimum $p_i$ from its subset $P_i$ and sends it to the master process. The master process determines the global minimum $p = \min p_i, i \in [0, \text{max\_processes}]$, and tells it to all slave processes, that now can delete all multiples of $p$ from their subsets $P_i$. This step will be successively repeated until the condition $p \leq \lfloor \sqrt{n} \rfloor$ becomes false. Now every subset $P_i$ of process $i$ only consists of prime numbers. In the end, the largest prime number found should be printed to the console.

For a better estimation of the speedup of your program the time for finding the prime numbers should be measured (MPI_Wtime()). Run your program with different upper limits $n$ (5,000,000/10,000,000/25,000,000/50,000,000) and different amounts of processes (1/2/4/8). Illustrate your results in a small chart!

**Hint:** Try to use non-blocking communication and think about the use of MPI_Reduce() or MPI_Allreduce()!

### 5 A practical example - an image processing algorithm

For manipulating images, modern image processing tools provide a vast range of different algorithms for all kind of circumstances. A simple but nevertheless effective algorithm is the smoothing algorithm that calculates the mean of each pixel’s value and its eight neighbours (some algorithms consider only the values from the diagonal neighbours or the horizontal and vertical neighbours). Considering an image as a two-dimensional array of colour values – a matrix –, the smoothing algorithm can be applied to all coefficients of this matrix—the pixels.

Write an OpenMP-Program `smooth_omp.c` and an MPI one `smooth_mpi.c` that read an image, apply a smoothing algorithm as described above to all pixels, and finally store the results into a new image. Therefore, the image should be decomposed into equal parts that are processed by the single threads/processes of your program.

The files `smooth_omp.c` and `smooth_mpi.c` already contain program skeletons, thus, you don’t have to care about reading and writing an image. After successfully reading an image in the PPM format `image` points to a data structure containing all image data. The functions `ImageWidth(image)` and `ImageHeight(image)` return the resolution of the image in pixels, with `ImageGetPixel(image, i, j, c)` you can retrieve a pixel’s value, where $i \in [0, \text{ImageWidth(image)}]$ and $j \in [0, \text{ImageHeight(image)}]$ denote the pixel coordinates and $c \in \{0, 1, 2\}$ the corresponding colour channel (0 for red, 1 for green, and 2 for blue). For writing the new pixel’s value you can use the function `ImageSetPixel(image, i, j, c, val)`, where `val` denotes the new colour value and $i$, $j$, and $c$ have the same meaning as in `ImageGetPixel()`.

### 6 OpenMP and MPI - which way to go?

Nowadays architectures tend to integrate shared memory machines into clusters. Under some circumstances, when the number of processors in a node exceeds the number of nodes of the cluster, this hybrid product is called constellation. In this context, the following question stands to reason: Is there a need for a new programming paradigm, or can the existent ones be modified, adapted, mixed so that these hybrid architectures can be tackled?

Consider the previous exercise. Which solution was easier to program (OpenMP or MPI)? Why? Program a solution to the previous exercise, having in mind a mixed approach: OpenMP and MPI. Does this approach give better time results than the ones from the previous exercise?

**Have fun!**