

# Message Passing Programming coursework: optimization of `percolate` v0.1.0 using a 2d domain decomposition with MPI

## Abstract

**Keywords:** Scientific programming, benchmark, parallelization, performance optimization, MPI

## 1. Introduction

`percolate` v0.1.0 is a scientific program written in the Fortran programming language. It generates a random matrix with two kinds of cells: empty and filled. Empty cells build clusters with their neighboring empty cells. `percolate` computes all clusters in the matrix and searches for a cluster that makes the matrix percolate. The matrix percolates, if there exists a cluster that begins in the left most column of the matrix and ends in the right most column.

The computing of the clusters—the clustering—is done iteratively and is the main cause of computation in `percolate`. If the clustering is done in a serial way, it does not scale well and clustering bigger matrices can consume a lot of time and power.

This paper presents a parallelized version of `percolate`. The parallel version decomposes the matrix into smaller chunks and distributes them among worker instances. Every worker performs the clustering of its chunk and communicates with the neighboring worker instances through halo swapping, which ensures that the whole matrix is clustered. Chunks are generated by splitting the matrix on both axes. This makes it a 2d domain decomposition of the matrix. The parallel version is based on MPI and the worker instances are MPI processes (see Message Passing Interface Forum, 2015).

First, this paper describes the clustering algorithm used by `percolate` and the way `percolate` is parallelized using the MPI library. The parallel version’s correctness is tested by a regression test suite, which is briefly outlined. Afterwards a benchmark is presented, which analyzes the scaling behavior of the parallel version over multiple amounts of MPI processes and with different sized matrices. The results of the benchmark are discussed and a conclusion is drawn.

## 2. Method

This chapter presents a mathematical definition of the clustering algorithm used by `percolate`. The poor running time of the serial version of the clustering algorithm is shown. Afterwards the parallelized version of the clustering is described and a brief outline of the

regression test suite for testing the correctness of the parallel version is given. At last, this chapter presents the benchmark, that is discussed in the following chapters.

## 2.1 Mathematical definition of the clustering algorithm used by `percolate`

Let  $A \in \mathbb{N}_0^{n \times n}$  be the matrix that is clustered by `percolate`. Let  $A(i, j); 1 \leq i, j \leq n$  be the element at the  $i$ th row and  $j$ th column of  $A$ . An element from  $A$  has a state. It is either empty or filled. `percolate` randomly initializes  $A$  with empty and filled elements. The density of the filled elements  $\rho_{goal}$  can be defined as a parameter, which is approximated during initialization.  $n$ , as well, is provided as a parameter to `percolate`.

Let  $\sigma : \mathbb{N}_0 \rightarrow \{\text{empty}, \text{filled}\}$  be a function mapping a non-negative integer to its state:

$$\sigma(x) := \begin{cases} \text{filled} & \text{if } x = 0 \\ \text{empty} & \text{otherwise} \end{cases}.$$

**Proposition 1** *Let  $x, y \in \mathbb{N}_0$ . For every  $x : \sigma(x) = \text{empty}$ , follows:  $x > y$ , if  $\sigma(y) = \text{filled}$ .*

**Proof** There exists no smaller non-negative integer than 0.  $\sigma(y) = \text{filled}$ , only for  $y = 0$ . Therefore, every number  $x$ , for which  $\sigma(x) = \text{empty}$ , must be bigger than  $y$ . ■

Let  $\mu$  be the function that returns the maximum value of an element and its neighbors:

$$\mu(A, i, j) := \max(A(i, j), A(i-1, j), A(i+1, j), A(i, j-1), A(i, j+1)).$$

For now, if  $i$  or  $j$  equals 0 or  $n+1$  (in other words, if  $i$  or  $j$  violate the boundaries of  $A$ ), then  $A(i, j)$  will return 0.

**Proposition 2** *If  $\sigma(A(i, j)) = \text{empty}$ , then  $\sigma(\mu(A, i, j)) = \text{empty}$  as well.*

**Proof**  $\mu(A, i, j)$  returns the maximum of the element  $A(i, j)$  and its neighbors, wherefore  $\mu(A, i, j) \geq A(i, j)$ . From Proposition 1 follows, that every number  $y \in \mathbb{N}_0 : \sigma(y) = \text{filled}$  must be less than  $A(i, j)$ , if  $\sigma(A(i, j)) = \text{empty}$ . ■

With Proposition 2, we can now safely derive a recursive definition of the clustering operation (because  $\mu$  will never change the state of an empty cell to filled). First, let  $c_{step} : \mathbb{N}_0^{n \times n} \rightarrow \mathbb{N}_0^{n \times n}$  be a single clustering step, that maps every empty element of  $A$  to its biggest neighbor and leaves filled elements untouched:

$$c_{step}(A) := i, j = 1, \dots, n : \begin{cases} \mu(A, i, j) & \text{if } \sigma(A(i, j)) = \text{empty} \\ 0 & \text{otherwise.} \end{cases}$$

The clustering operation  $c$  of **percolate** can now be defined as a recursive function, that executes  $c_{step}$ , as long as it continues to change empty elements to their biggest neighbor:

$$c(A) := \begin{cases} A & \text{if } c_{step}(A) = A \\ c(c_{step}(A)) & \text{otherwise.} \end{cases}$$

Imagine the case where for every element  $\sigma(A(i, j)) = \textit{filled}$  follows, that  $\mu(A, i, j) = A(i, j)$ . This would make  $c(A)$  call  $c_{step}$  just a single time, resulting in the best case running time of  $c$ :  $\Omega(c) = n^2$ , which—for bigger  $n$ —is still quite slow, even though it is the best case.

After the clustering, it is easy to check whether the matrix percolates. The matrix percolates, if  $\exists i, \exists j : A(i, 1) = A(j, n)$ . In other words, if any element in the first column has the same value as any element in the last column of  $A$ , the matrix percolates.

## 2.2 Parallelized version of percolate using MPI

To tackle the poor performance of the clustering algorithm, **percolate** was parallelized using the MPI Standard, version 3.1 (see Message Passing Interface Forum, 2015).

Let  $p$  be the amount of MPI processes the parallel version of **percolate** is executed with. Every process has a rank, which is drawn from a sequence  $0, 1, \dots, p-1$ . The process with rank 0 is called the root process.

The processes are arranged in a virtual, two dimensional Cartesian topology. The  $p$  processes are as evenly distributed as possible over both axes of  $A$  with the **MPI\_Dims\_create** routine. **MPI\_Dims\_create** returns an array *dims* with two elements, one for each axis of  $A$ . *dims<sub>row</sub>* contains the amount of processes the rows of  $A$  are split by, *dims<sub>column</sub>* the amount of processes the columns of  $A$  are split by. After distributing the processes over the rows and columns of  $A$ , **MPI\_Cart\_create** is used to generate the topology from **MPI\_Comm\_World** (see Message Passing Interface Forum, 2015, Chapter 7).

The previous chapter states, that  $A(i, j) = 0$ , if  $i$  or  $j$  equals 0 or  $n + 1$  ( $i$  or  $j$  out of bounds). The actual clustering of **percolate** behaves differently.  $A(i, j) = 0$ , only if  $j$  is out of bounds. If  $i$  is out of bounds, then a periodic boundary condition is used:  $A(0, j) = A(n, j)$ ,  $A(n + 1, j) = A(1, j)$ . This condition is easily implemented with **MPI\_Cart\_create**, which actually has an argument **periods**. **periods** is an array with two elements (for each axis) containing boolean values, whether an axis is periodic or not (see Message Passing Interface Forum, 2015, Chapter 7).

The root process initializes  $A$ . Now,  $A$  needs to be distributed among the processes. Every process has a coordinate based on its rank. The coordinates of a process come from the **MPI\_Cart\_coords** routine and is a representation of the place of the process in the two dimensional Cartesian topology. Let *coords<sub>p<sub>i</sub></sub>* be the coordinates of the process with rank  $p_i$  and let *coords<sub>p<sub>i</sub>,row</sub>* be its coordinate among the first axis and *coords<sub>p<sub>i</sub>,column</sub>* be its coordinate among the second axis. *coords* behaves like the ranks, starting from 0 to *dims<sub>row</sub>* − 1 and 0 to *dims<sub>column</sub>* − 1, respectively. The chunk of  $A$ , which is assigned to

the process with rank  $p_i$  can now be defined by a tuple  $\alpha_{p_i} := (i, j, l, m)$ .  $i$  and  $j$  are pointers to the first element of the chunk, while  $l$  is the amount of rows and  $m$  the amount of columns the chunk possesses. The parts of  $\alpha_{p_i}$  are computed as follows: first, let's define split values among both axes that are elements of the natural numbers  $s_{row} := \lfloor \frac{n}{dim_{s_{row}}} \rfloor$  and  $s_{column} := \lfloor \frac{n}{dim_{s_{column}}} \rfloor$ . With use of these split values and  $coords$  and  $dims$ , we can define  $i, j, l$  and  $m$ :

$$\begin{aligned}
i &:= coords_{p_i, row} \cdot s_{row} + 1 \\
j &:= coords_{p_i, column} \cdot s_{column} + 1 \\
l &:= \begin{cases} s_{row} + n \bmod dim_{s_{row}} & \text{if } coords_{p_i, row} = dim_{s_{row}} - 1 \\ s_{row} & \text{otherwise} \end{cases} \\
m &:= \begin{cases} s_{column} + n \bmod dim_{s_{column}} & \text{if } coords_{p_i, column} = dim_{s_{column}} - 1 \\ s_{column} & \text{otherwise.} \end{cases}
\end{aligned}$$

### 2.3 Regression test suite for testing the correctness of the parallel version of `percolate`

### 2.4 The benchmark discussed in the following chapters

## 3. Results

## 4. Discussion

## 5. Conclusion

## References

Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. Technical report, 2015. URL <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.