B160509

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

Abstract

This paper describes an optimized version of `percolate v0.1.0`. `percolate` is a scientific program, which generates a random matrix with two kinds of cells: empty and filled. Empty cells build clusters with their neighboring empty cells and `percolate` finds clusters that begin at the first column and end at the last. If such a cluster exists, the matrix percolates.

`percolate` performs a very costly clustering operation, wherefore the need for a faster solution arises.

This paper describes the clustering operation and demonstrates its poor performance, by analyzing its best case time complexity, before going into detail about how MPI was used for parallelizing `percolate` and how its correctness is tested with a regression test suite. Also, a benchmark is presented, which successfully shows the performance benefits of the parallel version and its scalability. The results of the benchmark are presented and discussed in this paper, before at last a conclusion is drawn.

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 exits a cluster that begins in the left most column of the matrix and ends in the right most one.

The clustering is done iteratively and is the main cause of computation in `percolate`. If it 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 time complexity 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 ith row and jth column of A. An element from A has an immutable state. It is either empty or filled. `percolate` randomly initializes A with empty and filled elements. The density of the filled elements ρ_{goal} can be defined as a parameter, which is approximated during initialization. n, as well, is provided as a parameter to `percolate`.

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

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

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

Proof There exits no smaller non-negative integer than 0. $state(y) = filled$, only for $y = 0$. Therefore, every number x, for which $state(x) = empty$, must be bigger than y. ∎

Let μ 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 any index equals 0 or $n+1$ (if it violates the boundaries of A), then $A(i,j)$ will return 0.

Proposition 2 *If $state(A(i,j)) = empty$, than $state(\mu(A,i,j)) = empty$ 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 $y \in \mathbb{N}_0 : state(y) = filled$ must be less than $A(i,j)$, if $state(A(i,j)) = empty$. ■

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

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

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 $state(A(i,j)) = filled$ 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). Algorithm 1 shows the parallel version of `percolate`. This chapter outlines the implementation of the parallel version and its intricacies.

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 $0, 1, \ldots, p-1$. 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 the topology with the `MPI_Dims_create` routine. `MPI_Dims_create` returns an array $dims$ with two elements, one for each axis of the topology. $dims_{row} = dims(1)$ contains the amount of processes the rows of A are split by, $dims_{column} = dims(2)$ the amount of processes the columns of A are split by. After generating the dimensions of the topology, `MPI_Cart_create` is used to generate the actual communicator that manages the virtual 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 of the topology is periodic or not (see Message Passing Interface Forum, 2015, Chapter 7).

The root process initializes A, which needs to be distributed to the MPI processes (see Algorithm 1, lines 2ff). Every process has coordinates in the topology, based on its rank. The coordinates of a process come from the `MPI_Cart_coords` routine (see Message Passing Interface Forum, 2015, Chapter 7).

Let $coords_{p_i}$ be the coordinates of the process with rank p_i and let $coords_{p_i,row}$ be the coordinates' value for the first axis and $coords_{p_i,column}$ be its value for the second axis of the topology. $coords$ behaves like the ranks, ranging from 0 to $dims_{row} - 1$ and 0 to $dims_{column} - 1$, respectively (see Message Passing Interface Forum, 2015, Chapter 7).

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 the indices of A, which point to the first element of the chunk, while l is the amount of rows and m the amount of columns the chunk possesses. Let $s_{row} := \lfloor \frac{n}{dims_{row}} \rfloor$ and $s_{column} := \lfloor \frac{n}{dims_{column}} \rfloor$ be the general size of the splits of A for both axes. With use of these split values and $coords$ and $dims$, we can define i, j, l and m:

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

For both axes of A, every process has a chunk of the same size, except the processes, which coordinates' value for the equivalent axis in the topology is the highest possible value for it ($dims_{row} - 1$ or $dims_{column} - 1$). The last process has a chunk of the same size as the other processes, plus the rest of the elements of A among that axis (see Figure 1a).

l and m are needed in order to generate a strided vector type with `MPI_Type_vector`, which is used for sending the chunk from the root process to p_i and vice versa. In this case, the type for sending p_i's chunk would be generated with setting `count` to m, `blocklength` to l and `stride` to n (see Message Passing Interface Forum, 2015, Chapter 4).

The chunks are scattered from the root process to every non-root process with `MPI_Ssend`. The root process simply copies its chunk from A. Gathering, after the clustering is finished works the same way as the scattering, just reverse (the non-root processes sending their chunks back to the root process with `MPI_Ssend` and the root process copying its chunk back to A) (see Algorithm 1, lines 5,7 and Message Passing Interface Forum, 2015, Chapter 3).

Algorithm 1 : parallel version of `percolate`

1: initialize MPI and the Cartesian topology
2: **if** rank $= 0$ **then**
3:    randomly initialize A
4: **end if**
5: scatter A to every process's chunk
6: execute c_{par} (Algorithm 2)
7: gather the chunks back to A
8: **if** rank $= 0$ **then**
9:    find out if A percolates
10:    save A as a Portable Gray Map file
11: **end if**
12: finalize MPI

Every process has four neighbors: a left, right, upper and lower neighbor (see Figure 1b). The neighbors are needed for swapping halos, so the clustering is actually done over the whole matrix, and not just over the chunks. The neighbors are determined with `MPI_Cart_shift`, shifting both axes of the topology up one element (see Message Passing Interface Forum, 2015, Chapter 7).

The second axis of the topology is not periodic, so processes for which $coords_{p_i,column} = 0$ have `MPI_PROC_NULL` as their left neighbor. The same goes for processes for which $coords_{p_i,column} = dims_{column} - 1$, only their right neighbor is set to `MPI_PROC_NULL` (see Figure 1b and Message Passing Interface Forum, 2015, Chapter 3).

p_i's chunk is the actual chunk it is provided from A by the root process, plus a halo (chunk: $L+2 \times M+2$, its indices ranging from $0, \dots, L+1$ and $0, \dots, M+1$). The halo is used as a container for the data received from p_i's neighbors during the halo swapping, or as a buffer of empty elements for μ, if the process's left or right neighbor is `MPI_PROC_NULL` (see Message Passing Interface Forum, 2015, Chapter 3).

The halo swapping happens before each $cluster_{step}$ (see Algorithm 2, line 3). Halo swapping means, the outer most row or column (in any direction of left, right upper or lower) is send to the corresponding neighbor, which sends its outer most row or column in return (the opposite of the direction it receives from). In the example shown in Figure 1, p_4 would send its first row $(5, 6)$ to p_2, its upper neighbor and would receive $(0, 3)$ from p_2 (its lower and only row) in return, which is then stored in $\text{chunk}_{p_4}(0, 1:2)$.

The halo swapping is realized with `MPI_Sendrecv`. Because the first axis of the topology is periodic, `MPI_Sendrecv` can not be called with the same neighbor (e.g. sending to upper and receiving from upper). Instead, `MPI_Sendrecv` must be called with upper and lower for sending and receiving (sending to upper and receiving from lower and vice versa). Otherwise `percolate` would deadlock (see Message Passing Interface Forum, 2015, Chapter 3).

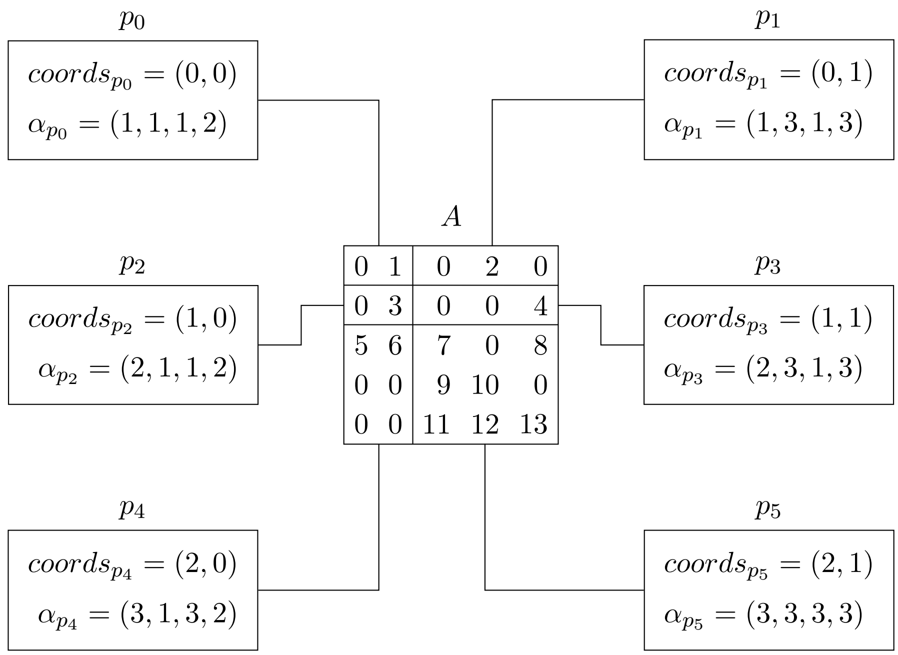
For stopping the clustering, every process computes the sum over each element in its chunk (without the halos). The sum is then reduced and broadcast to every process with `MPI_Allreduce` (see Message Passing Interface Forum, 2015, Chapter 5). If the reduced sum is equal to the sum generated by the previous step, the clustering is finished (see Algorithm 2).

Algorithm 2 : c_{par}

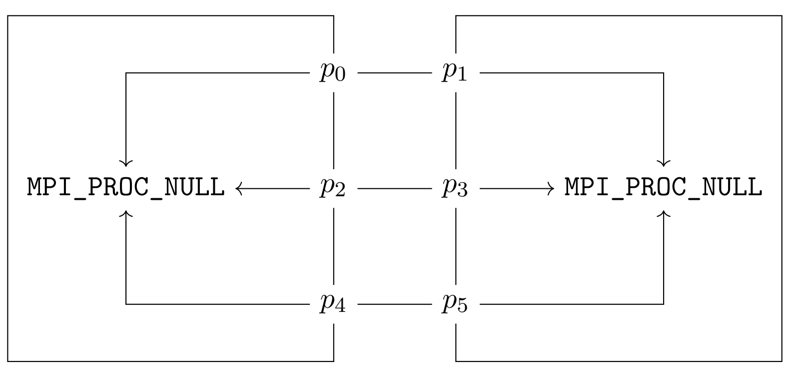
$sum_{A'} := 0$
while *true* **do**
  swap halos with neighbors
  chunk := c_{step}(chunk)
  $sum := \sum_{i=1}^{l} \sum_{j=1}^{m} \text{chunk}(i, j)$
  reduce *sum* over all processes into sum_A
  if $sum_A = sum_{A'}$ **then**
    exit loop
  end if
  $sum_{A'} := sum_A$
end while

In order to insure correctness of the result, the first axis of the topology can not contain more elements than n ($dims_{row} \leq n$), because this axis uses a periodic boundary condition. If $dim_{row} > n$, than $s_{row} = 0$. That means, only the processes with $coords_{row} = dims_{row} - 1$ would contain elements and all other processes would contain empty chunks, because $l = s_{row} = 0$. If not for the periodic boundary condition, this would only endanger the performance of `percolate`, not its correctness. But during halo swapping, a process with $coords_{row} = 0$ would receive its lower halo from a process with $coords_{row} = dims_{row} - 1$, which means they are lost to the processes not containing empty chunks (which should actually have themselves as lower and upper neighbor), making the periodic boundary condition non-periodic.

Therefore, the Cartesian communicator truncates $dim_{row} = n$, if $dim_{row} > n$, which means there are n processes among the first axis of the topology, each containing a chunk which is the subset of a single row of A. Because empty processes only produce an overhead of communication and therefore endanger performance, the same truncation happens for dim_{column}.



(a) Graph displaying how a randomly initialized 5×5 matrix is distributed among 6 processes.



(b) Directed graph showing the neighbors of each process.

Figure 1: Example of how $A : 5 \times 5$ is distributed among 6 processes. Also shows the neighbors of each process with wich the halo swapping is done.

2.3 The regression test suite

The correctness of the parallel version of `percolate` is tested with an expandable regression test suite, included in the project. All tests ran at this point were successful. The test suite tests the output of the parallel version of `percolate` against output of the serial version with the same parameters. The output, in this cases, is the generated Portable Gray Map file (see Algorithm 1, line 10). If the files are identical, the test was successful.

The parallel version of `percolate` was tested against the serial version with the following parameters: $n := 2^0, 2^1, \dots, 2^9$. Each n was combined with a seed for the random number generator: $seed := 1560, 1561, \dots, 1564$. The parallel version was executed with 1 to 4 processes on a single Linux machine running Open MPI, version 4.0.2 and 32 and 64 processes on two nodes of the Cirrus supercomputer running HPE MPT version 2.16, with 32 processes per node (see The Open MPI Project, 2019; EPCC, 2019).

2.4 The benchmark

The benchmark tests the performance and scalability of the parallel version of `percolate`. It was executed on eight back end nodes of the Cirrus supercomputer with exclusive access (see EPCC, 2019).

Measured were the clustering plus the scattering and gathering of A. Initialization and io were not part of the measurements. `MPI_Wtime` was used for measuring the execution time (see Message Passing Interface Forum, 2015, Chapter 8).

The benchmark executed `percolate` with $2^0, 2^1, \dots 2^8$ processes, 32 processes per node. Each was tested with $n := 2000, 3000, 4000, 5000$ and ten different seeds (one to ten), resulting in 360 distinct time measurements.

`percolate` was compiled with the Intel Fortran Compiler (`ifort`) version 18.0.5, with the maximum serial optimization provided by the compiler (optimization level `O3`) (see Intel, 2018).

3. Results

Figure 2 shows the speedup of the parallel version of `percolate` with more processes. Every time measurement was grouped by the amounts of processes p the measurement was taken with. The average over the grouped timings were used to compute the speedup. `percolate`'s parallel version's speedup is very close to the optimal speedup. The biggest difference between the optimal and the actual speedup happens for 256 processes. The actual speedup is still approximately 89 percent of the optimum.

Table 1 shows the average time in seconds and the standard deviation over every benchmarked seed per amount of processes p and size of the matrix n. For all p and n, the average execution time and the standard deviation decreases for bigger p and smaller n.

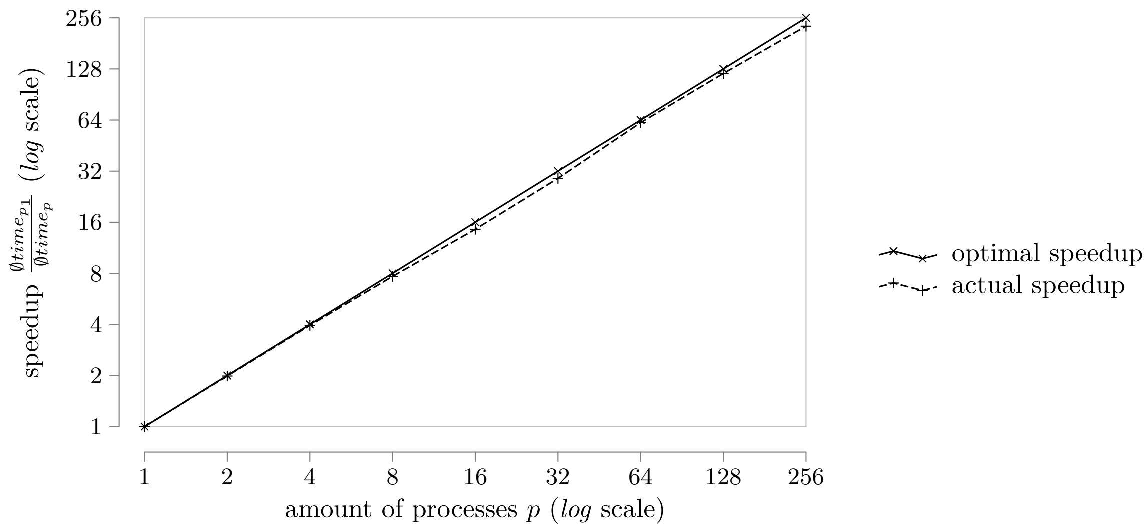


Figure 2: Speedup when using more processes. The speedup was calculated over the average of every measurement, grouped by the amount of processes.

	2000		3000		4000		5000	
p	time	σ	time	σ	time	σ	time	σ
1	182.000	52.643	495.825	136.927	969.675	248.402	2101.600	575.685
2	91.575	26.429	249.325	68.908	488.400	125.101	1058.475	289.985
4	45.825	13.239	125.200	34.560	245.650	62.895	532.425	145.910
8	23.150	6.689	64.100	17.673	126.900	32.593	274.250	75.165
16	11.675	3.385	33.125	9.140	67.300	17.230	145.700	39.869
32	6.050	1.731	15.875	4.345	32.800	8.409	74.525	20.422
64	3.125	0.907	8.000	2.233	15.325	3.905	34.250	9.347
128	1.675	0.409	4.300	1.224	7.950	2.047	17.225	4.687
256	0.875	0.295	2.325	0.646	4.200	1.019	9.000	2.441

Table 1: The average execution time and standard deviation σ per amount of processes p and matrix size n. Each combination of p and n was executed with ten different seeds (see Chapter 2.4).

4. Discussion

The benchmark successfully shows the performance benefits and scalability of the parallel version of `percolate`. The fact that the actual speedup is very close to the optimal speedup is an indication for the implementation's efficiency and the fact, that the overhead of communication between processes is negligible in comparison to the performance gains. Overall can the benchmark be described as rather unremarkable, since it does not reveal any issues with the implementation.

Also the fact that the average time and standard deviation displayed in Table 1 continuously decrease with smaller n and bigger p show the robustness of the parallel version of `percolate`. The fact, that the speedup slightly drops with higher amounts of processes could be the result of added distance between them. For example, the tests with 256 processes were distributed among eight nodes of Cirrus, while the tests with 128 processes were only distributed among half of that. This adds distance between processes in form of higher network latency, which can decrease the performance of the program.

A ceiling for the scalability of the parallel version, where it hits a plateau and performance does not increase anymore, or only with an exponential amount of more processes, was not determined by the benchmark.

5. Conclusion

The parallel version of `percolate` truly increases the performance of the program, which is shown by the conducted and presented benchmark. The fact that the actual speedup of the parallel version is close to the optimal speedup (see Figure 2), shows its strong scaling capabilities. It makes the overhead of using message passing in form of the MPI library negligible in comparison to the performance gains.

Therefore, based on the results presented in this paper, the parallel version of `percolate` is deemed a successful optimization and enhancement of the program.

References

EPCC. Cirrus, 2019. URL `https://www.cirrus.ac.uk`.

Intel. Intel Fortran Compiler 18.0 for Linux, 2018. URL `https://software.intel.com/en-us/articles/intel-fortran-compiler-180-for-linux-release-notes-for-intel-parallel-studio-xe-2018`.

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`.

The Open MPI Project. Open MPI: Open Source High Performance Computing, 2019. URL `https://www.open-mpi.org`.