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 percolates. The matrix percolates, if there exits 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 parallel version of `percolate`. 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

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

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

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

Proof There exits no smaller non-negative integer than 0. $\sigma(y) = filled$, only for $y = 0$. Therefore, every number x, for which $\sigma(x) = empty$, 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 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 $\sigma(A(i, j)) = empty$, than $\sigma(\mu(A, i, j)) = empty$ 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 $y \in \mathbb{N}_0 : \sigma(y) = filled$ must be less than $A(i, j)$, if $\sigma(A(i, j)) = empty$. ■

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). 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:

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

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 $\sigma(A(i, j)) = filled$ 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.

To tackle the poor performance, `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. The processes are arranged in a virtual, two dimensional cartesian topology (see Message Passing Interface Forum, 2015, Chapter 7).

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`.