Programming Skills coursework II b: benchmarking `percolate v0.1.0` with different densities

Abstract

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

The benchmark of `percolate` looks at how the program scales with different densities of filled cells. The goal of this benchmark is to find bottlenecks of the execution time, in order to optimize `percolate` in coming releases.

The benchmark was executed on the Cirrus supercomputer with exclusive access to one back end node. The results of the benchmark show extreme scaling behavior of `percolate`, but successfully exposes the bottleneck.

This paper discusses the results of the benchmark and gives an outline for optimizing `percolate` in future releases.

Keywords: Scientific programming, performance optimization

1. Introduction

This paper documents the results of a benchmark performed on `percolate v0.1.0`. `percolate` 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 neighbors and `percolate` finds clusters that begin at the first column and end at the last. If such a cluster exists, the matrix percolates.

One way to configure `percolate` is by setting the density of the empty cells. This benchmark looks at the behavior of `percolate` with different densities of empty cells and how the density influences the execution time. The goal is to find the bottlenecks concerning the execution time, in order to optimize `percolate` and make it perform better, executing faster.

The benchmark was performed on the Cirrus supercomputer with exclusive access to one back end node (see EPCC, 2019). The program was compiled with the GNU Fortran Compiler (`gfortran`) version 4.8.5, with the maximum optimization provided (optimization level `O3`) (see GNU Compiler Collection Team, 2019).

This paper begins by describing `percolate v0.1.0` and the conducted benchmark. Afterwards the results are presented and discussed. At last a conclusion is drawn and next steps for a follow-up benchmark are outlined.

2. Method

Let $n \in \mathbb{N}$ be a positive integer. Let $A : n \times n$ be a matrix, $A \in \mathbb{N}_0^{n \times n}$. $A(i,j); 1 \leq i,j \leq n; i,j \in \mathbb{N}$ is the cell of A in the ith row and the jth column. Let $S := \{filled, empty\}$ be a binary set containing the two states a cell of A can have and let $\sigma : \mathbb{N}_0 \to S$ be a function that maps a cell of A to its state:

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

Let $A' : n+2 \times n+2$ be a version of A with a halo containing filled cells:

$$i,j = 0,\ldots,n+1 : A'(i,j) := \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \vee i = n+1 \vee j = n+1 \\ A(i,j) & \text{otherwise} \end{cases}. \quad (1)$$

The density of the empty cells ρ of A is determined by the following function:

$$\rho(A) := \frac{|\{i,j = 1,\ldots,n : \sigma(A(i,j)) = empty\}|}{n^2}.$$

`percolate v0.1.0` randomly initializes A with empty and filled cells. The empty cells are taken continuously from the sequence $1, 2, \ldots, n^2$. So, after initialization the empty cells of A are $1,\ldots,|\{i,j = 1,\ldots,n : \sigma(A(i,j)) = empty\}|$, which are randomly distributed over A. Afterwards the program sets A' according to (1) (see Algorithm 1, lines 1–2). After initialization, clusters in A' are build iteratively (see Algorithm 2). This is done by setting all empty cells of A' to their biggest neighbor. Let $\mu : \mathbb{N}_0^{m \times m} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}_0; m \in \mathbb{N}$ be the function returning the biggest value of all neighboring cells and the current cell:

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

The clustering is finished, once no cell changes value anymore (see Algorithm 2, line 2).

After the clustering the clusters are sorted in descending order based on their size ($size : \mathbb{N}_0^{n \times n} \times \mathbb{N} \to \mathbb{N}_0; size(A,x) := |\{i,j = 1,\ldots,n : A(i,j) = x\}|$)(see Algorithm 1, line 5). Sorting is done using the quicksort algorithm (see Hoare, 1961).

Afterwards every empty cell of A is mapped to its index in the array containing the sorted clusters. Every filled cell is mapped to $|clusters| + 1$. The result of this operation is the color map (see Algorithm 1, line 6). The color map is needed for writing a Portable Gray Map Image (PGM).

After doing this operations, `percolate` writes a log to `stdout` and writes A and the color map to files. These io-operations are not measured during the benchmark.

For the benchmark, n was set to 2000. `percolate` offers an interface which controls the initialization behavior. One can define a density ρ_{goal} and a seed which is passed to the pseudo random number generator. During initialization, ρ_{goal} is approximated.

Algorithm 1 : `percolate`

1: randomly initialize A
2: set A' according to (1)
3: build_clusters(A')
4: $A := A'(1:n, 1:n)$
5: sort the clusters based on their size
6: build the color map

Algorithm 2 : build_clusters(A')

1: $A''(i,j) := 0; i,j = 0,\ldots,n+1$
2: **while** $\Sigma_{i,j=1}^{n}(A'(i,j) - A''(i,j)) > 0$ **do**
3: $A'' := A'$
4: **for** $i,j = 1,\ldots,n$ **do**
5: $$A'(i,j) = \begin{cases} \mu(A',i,j) & \text{if } \sigma(A'(i,j)) = empty \\ 0 & \text{otherwise} \end{cases}$$
6: **end for**
7: **end while**

The benchmark ran `percolate` with $\rho_{goal} := 0.01, 0.02, \ldots, 0.99$. Every ρ_{goal} was tried with five seeds, resulting in 495 distinct measures, with an approximately uniform distribution over ρ generated during the initialization.

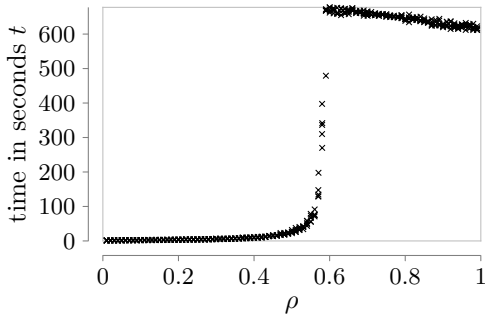
The three operations: clustering, sorting and building the color map were measured (see Algorithm 1, lines 3ff). The sum of all three measures is the execution time (io-operations and initialization excluded).

The execution time is measured in seconds. OpenMP 4.5's `omp_get_wtime` function was used for the time measuring (see OpenMP Architecture Review Board, 2015, Chapter 3.4.1).

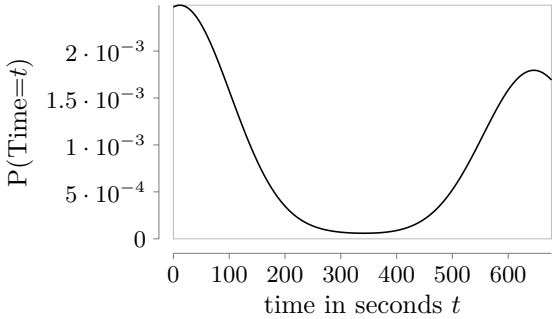
3. Results

The overall execution time t in seconds (the sum of the execution time of the three operations, see previous chapter) gets mapped onto the corresponding density ρ of A (see Figure 1). The benchmark also looks at how the three different measured operations influence t, to see how they scale and to further underline where the bottlenecks lie.

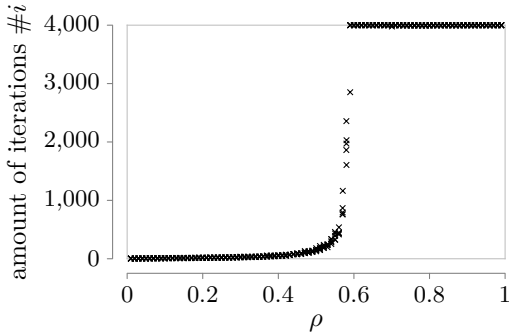
The relationship between ρ and the execution time is clearly not linear—like expected—and behaves rather extreme (see Figure 1a). The program produces a discontinuity like jump at $\rho \approx 0.58$. While the overall average t is approximately 278.49 seconds, the average t over all measures where $\rho \leq 0.58$ is just approximately 14.79 seconds, while the average t over all measures where $\rho > 0.58$ is approximately 639.33 seconds. The average of the



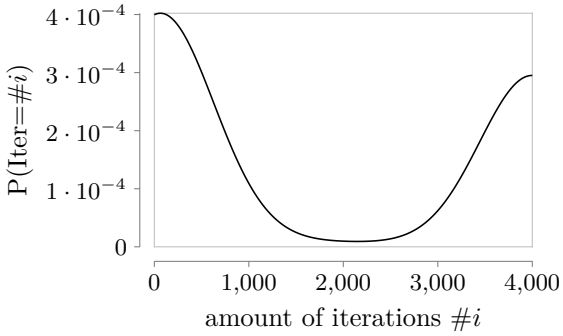
(a) Scatter plot mapping ρ to the execution time of Algorithm 1.



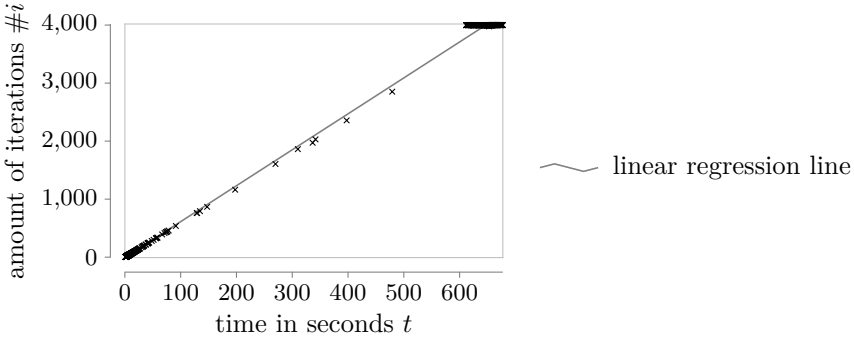
(b) Gaussian kernel density estimation of the execution time. Clearly shows that Algorithm 1 favors extreme times (the function has its maxima at the edges).



(c) Scatter plot mapping ρ to the amount of iterations of Algorithm 2.



(d) Gaussian kernel density estimation of the amount of iterations. Clearly shows that Algorithm 2 favors extreme amounts of iterations (the function has its maxima at the edges).



(e) Plot showing the correlation of the execution time of Algorithm 1 and the amounts of iterations of Algorithm 2.

Figure 1: Results of the first phase of the benchmark.

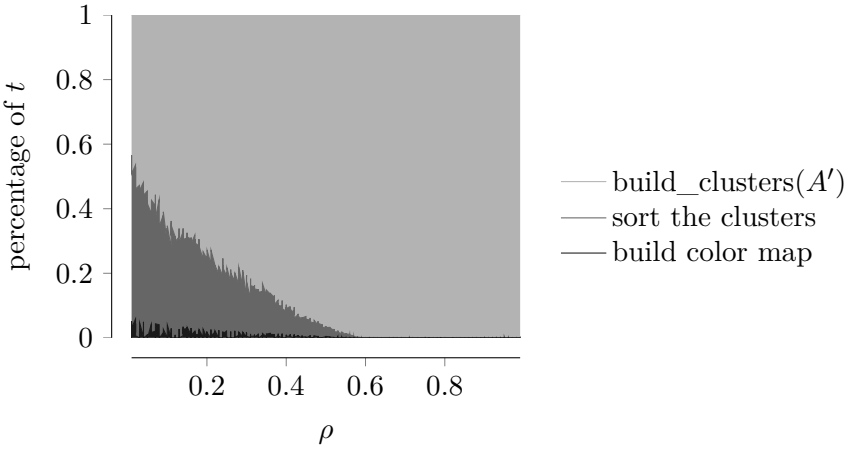


Figure 2: Percentage of t spent for every measured operation in Algorithm 1.

measurements where $\rho \leq 0.58$ is a staggering 2 percent of the average of the measurements where $\rho > 0.58$. The density of the execution time is a third factor that makes the measurement extreme (see Figure 1b). Added to this jumping behavior and the scale, the distribution of the execution time also favors extreme timings. The upper and lower quartiles of the distribution of t contain 50 percent of all measurements. This means, not only is there an extreme jump from very low execution time to very high, there are also just a few time measurements that are close to the overall average execution time of approximately 278.49 seconds.

Figures 1c and 1d show the same relation, t replaced with the amount of iterations $\#i$ of the while-loop in Algorithm 2. The plots look rather similar to each other. Figure 1e shows the relation between t and $\#i$ and it is clearly linear. The Pearson correlation coefficient is approximately 0.999, very close to a perfect linear relationship, making it save to say, that $\#i$ is the main cause for t. The relation is not perfectly linear, because t drops slightly with higher ρ ($\rho > 0.58$), while $\#i$ stays constant over the same interval (see Figures 1a–1d).

The fact that Algorithm 2 is the main reason for t—and therefore the bottleneck—is underlined by looking at the three different measured operations (clustering, sorting and building the color map) distinctly. Especially for the extreme high timings (for $\rho > 0.58$), approximately 99 percent of t was spent on the clustering (see Figure 2).

For the lower densities—which produce very low t—the clustering takes approximately linearly less with lower ρ. At its lowest point, the clustering only takes approximately 45 percent of t, while the rest of the time is spent sorting the clusters.

The building of the color map is constant over ρ. The average time spent on building the clusters is 0.05 seconds and does not deviate much with a standard deviation of 0.04 seconds. The sorting takes slightly more time than the building of the color map, but it is also constant over ρ and takes only 0.71 seconds on average with a standard deviation of

0.12 seconds. Both operations can be deemed irrelevant to t concerning its scale. Sorting only plays a role for very low ρ which also produce very low t.

4. Discussion

On the one hand, this benchmark sufficiently answers the question, where the performance bottleneck of `percolate v0.1.0` lies. Clearly Algorithm 2 is the main source of computation and therefore execution time.

On the other hand, this benchmark raises the question, why the scaling over ρ is not linear and favors such extreme t, with a abrupt jump at $\rho \approx 0.58$. This question is not answered by the benchmark and needs to be further evaluated.

The most obvious optimization strategy for `percolate`—based on this benchmark—would be to reduce the high execution time for more dense A.

This could be done by the following possible optimizations:

- Update μ to contain the diagonal neighbors as well. This would reduce the amount of iterations, respectively reducing the maximum distance of two cells in the matrix from n^2 to n.
- Parallelizing the inner loop of Algorithm 2.
- Not using an iterative approach. Instead of iterating over the matrix, one could use a priority queue, which prioritizes higher $\mathbb{N}$ and points to all the empty cells of A. This empty cell with the highest value is taken and all neighbors are recursively updated. The updated cells are removed from the queue. If the update process stops, the now highest value from the queue is taken and the update process is done again. This is continued until the queue is empty.

  Since taking the highest value is not even a necessity (it was just used as the criteria for an early return from the while-loop of Algorithm 2), one could even replace the priority queue with a different, more easy to parallelize, data structure to further optimize `percolate`.

If these proposed optimizations can reduce the execution time remains to be tested.

Another open question not answered is the divergence between the amount of iterations of the while-loop of Algorithm 2 and the execution time for $\rho > 0.58$ (see Figures 1a, 1c). While the amount of iterations stays constant on this interval, the execution time drops with bigger ρ. This is suggestive for another factor influencing the execution time. This benchmark has not tested the amounts of clusters or their size and how they influence the execution time, which could also be a factor.

5. Conclusion

`percolate v0.1.0` scales rather extreme and unexpected over different densities ρ:

1. A discontinuity like jump for $\rho \approx 0.58$.

2. The scale of the execution time. The average execution time for $\rho > 0.58$ is approximately 639.33 seconds, while the average execution time for $\rho \leq 0.58$ lies at approximately 14.79 seconds—a staggering 2 percent of the average execution time for $\rho > 0.58$.

3. The distribution of the execution time favors the extremes (both minimum and maximum). 50 percent of all measurements lie in the outer quartiles.

Even though this benchmark raises some unanswered questions (most of all the scaling behavior), it shows the bottleneck of the execution time in Algorithm 2. This allows for precise optimization of `percolate` in order to improve its performance, especially for $\rho > 0.58$. The previous chapter outlined different approaches, which can be taken in order to optimize `percolate`.

On the other hand the unanswered questions must be addressed in a follow up benchmark, possibly combined with the results of the optimized version of `percolate`, which may scale differently.

The next tasks for optimizing `percolate` will be to set up a thorough benchmark suite for regression testing, before optimizations will be tested (`percolate v0.1.1`). Hopefully, `percolate v0.1.2` will already contain an optimized version of Algorithm 2, making it perform and scale better.

References

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

GNU Compiler Collection Team. The GNU Fortran Compiler, 2019. URL `https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gfortran/`.

C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961. ISSN 0001-0782. doi: 10.1145/366622.366644. URL `http://doi.acm.org/10.1145/366622.366644`.

OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.