B160509

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

Abstract

Keywords:

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:

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

`percolate` `v0.1.0` randomly initializes A and 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 ($\mathit{size} : \mathbb{N}_0^{n\times n} \times \mathbb{N} \to \mathbb{N}_0; \mathit{size}(A,x) := |\{i,j=1,\ldots,n : A(i,j) = x\}|$)(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 $|\mathit{clusters}| + 1$. 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.

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
```

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.

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.

Algorithm 2 : build_clusters(A')

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

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).

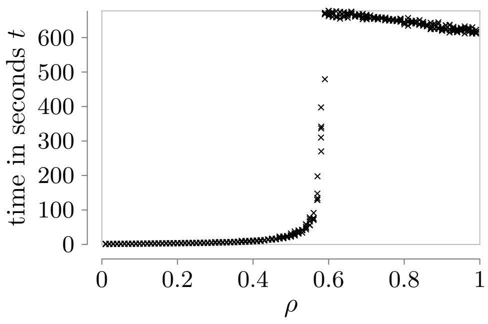
The execution time is measured in seconds. OpenMP 4.5's `omp_get_wtime` function was used for the time measuring (see OpenMP Architecture Review Board, 2015, Chapter 3.4.1).

3. Results

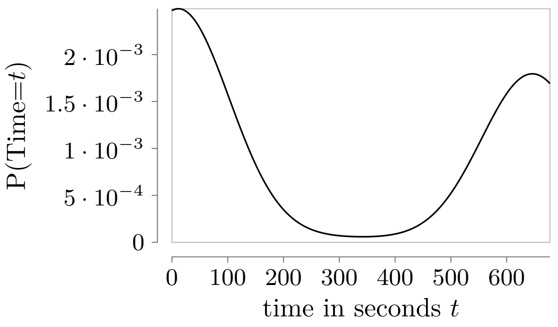
First, the overall execution time t in seconds (the sum of the execution time of the three operations, see previous chapter) is measured. t is mapped to the density ρ of A (see Figure 1). The benchmark also looks at how the three different measured operations influence t, to see how they scale and to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

The relationship between ρ and the execution time is clearly not linear—like expected—and 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 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.

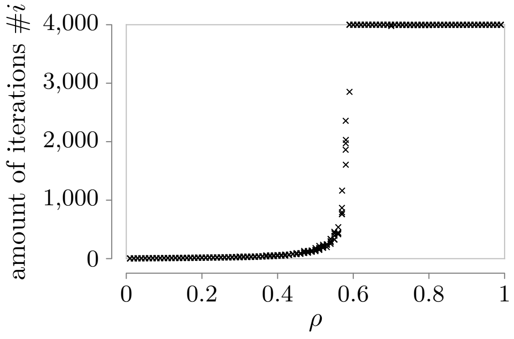
Figures 1c and 1d show the same relation, t replaced with the amount of iterations $\#i$ of the while-loop in Algorithm 2. The plots look rather similar to each other. Figure 1e shows the relation between t and $\#i$ and it is clearly linear. The Pearson correlation coefficient is approximately 0.999, very close to a perfect linear relationship, making it save to say, that



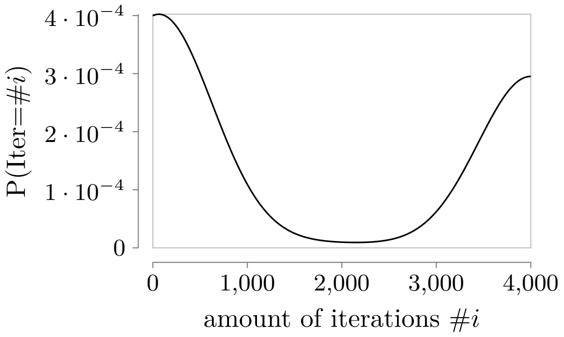
(a) Scatter plot mapping ρ to the execution time of Algorithm 1.



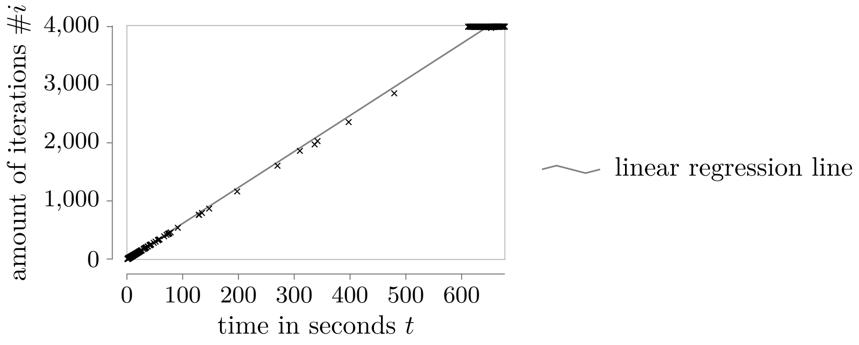
(b) Gaussian kernel density estimation of the execution time. Clearly shows that Algorithm 1 favors extreme times (the function has its maxima at the edges).



(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.

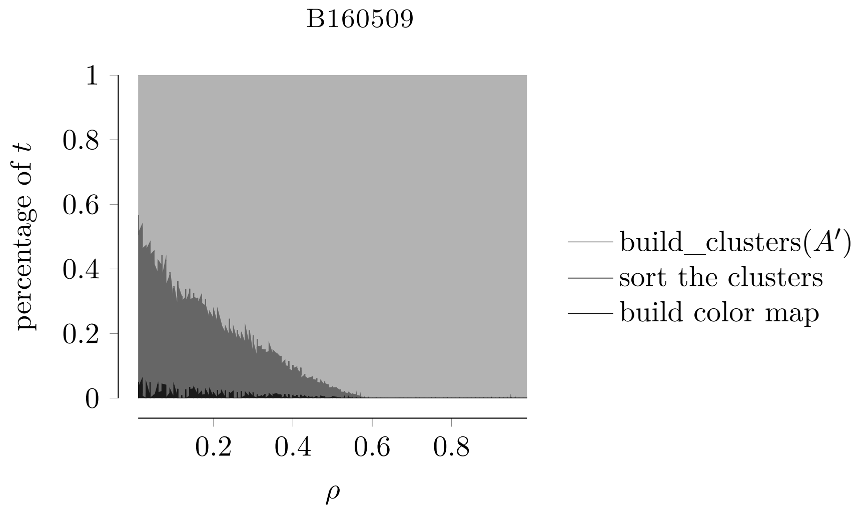


Figure 2: Percentage of t spent for every measured operation in Algorithm 1.

$\#i$ 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

5. Conclusion

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`.