WIKIPEDIA

Parallel all-pairs shortest path algorithm

A central problem in algorithmic graph theory is the shortest path problem. Hereby, the problem of finding the shortest path between every pair of nodes is known as **all-pair-shortest-paths (APSP)** problem. As sequential algorithms for this problem often yield long runtimes, parallelization has shown to be beneficial in this field. In this article two efficient algorithms solving this problem are introduced.

Contents

Problem definition

Let $G = (V, E, w)$ be a directed Graph with the set of nodes V and the set of edges $E \subseteq V \times V$. Each edge $e \in E$ has a weight $w(e)$ assigned. The goal of the all-pair-shortest-paths problem is to find the shortest path between **all** pairs of nodes of the graph. For this path to be unique it is required that the graph does not contain cycles with a negative weight.

In the remainder of the article it is assumed that the graph is represented using an adjacency matrix. We expect the output of the algorithm to be a distancematrix D. In D, every entry $d - i, j$ is the weight of the shortest path in G from node i to node j.

The Floyd algorithm presented later can handle negative edge weights, whereas the Dijkstra algorithm requires all edges to have a positive weight.

Dijkstra algorithm

The Dijkstra algorithm originally was proposed as a solver for the single-source-shortest-paths problem. However, the algorithm can easily be used for solving the All-Pair-Shortest-Paths problem by executing the Single-Source variant with each node in the role of the root node.

In pseudocode such an implementation could look as follows:

```
1    func DijkstraSSSP(G,v) {
2        ... //standard SSSP-implementation here
3        return $d_v$;
4    }
5
6    func DijkstraAPSP(G) {
7        D := |V|x|V|-Matrix
8        for i from 1 to |V| {
9           //D[v] denotes the v-th row of D
10          D[v] := DijkstraSSP(G,i)
11       }
12   }
```

In this example we assume that `DisjktraSSSP` takes the graph G and the root node v as input. The result of the execution in turn is the distancelist d_v. In d_v, the i-th element stores the distance from the root node v to the node i. Therefore the list d_v corresponds exactly to the v-th row of the APSP distancematrix D. For this reason, `DijkstraAPSP` iterates over all nodes of the graph G and executes `DisjktraSSSP` with each as root node while storing the results in D.

The runtime of `DijkstraSSSP` is $O(|V|^2)$ as we expect the graph to be represented using an adjacency matrix. Therefore `DijkstraAPSP` has a total sequential runtime of $O(|V|^3)$.

Parallelization for up to |*V*| processors

A trivial parallelization can be obtained by parallelizing the loop of `DijkstraAPSP` in line*8*. However, when using the sequential `DijkstraSSSP` this limits the number of processors to be used by the number of iterations executed in the loop. Therefore, for this trivial parallelization $|V|$ is an upper bound for the number of processors.

For example, let the number of processors p be equal to the number of nodes $|V|$. This results in each processor executing `DijkstraSSSP` exactly once in parallel. However, when there are only for example $p = \frac{|V|}{2}$ processors available, each processor has to execute `DijkstraSSSP` twice.

In total this yields a runtime of $O(|V|^2 \cdot \frac{|V|}{p})$, when $|V|$ is a multiple of p. Consequently, the efficiency of this parallelization is perfect: Employing p processors reduces the runtime by the factor p.

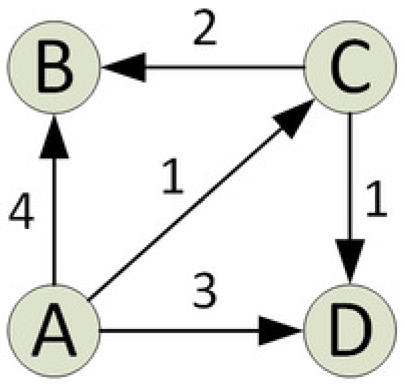
Another benefit of this parallelization is that no communication between the processors is required. However, it is required that every processor has enough local memory to store the entire adjacency matrix of the graph.

Parallelization for more than |*V*| processors

If more than $|V|$ processors shall be used for the parallelization, it is required that multiple processors take part of the `DijkstraSSSP` computation. For this reason, the parallelization is split across into two levels.

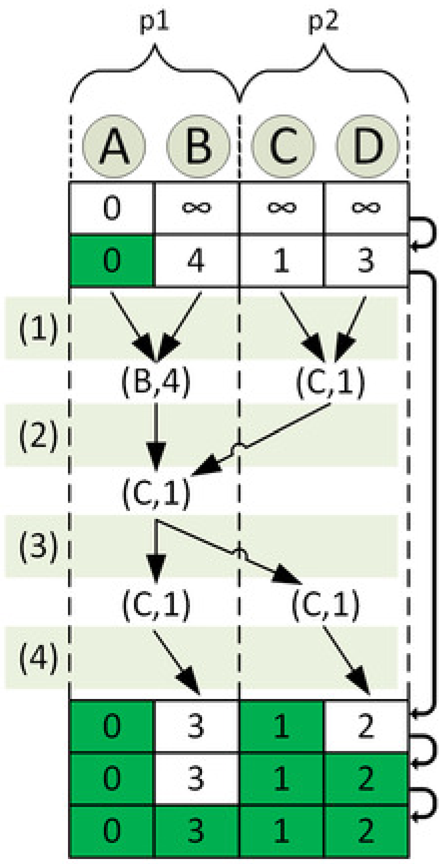
For the first level the processors are split into $|V|$ partitions. Each partition is responsible for the computation of a single row of the distancematrix D. This means each partition has to evaluate one `DijkstraSSSP` execution with a fixed root node. With this definition each partition has a size of $k = \frac{p}{|V|}$ processors. The partitions can perform their computations in parallel as the results of each are independent of each other. Therefore, the parallelization presented in the previous section corresponds to a partition size of 1 with $p = |V|$ processors.

The main difficulty is the parallelization of multiple processors executing `DijkstraSSSP` for a single root node. The idea for this parallelization is to distribute the management of the distancelist d_v in DijkstraSSSP within the partition. Each processor in the partition therefore is exclusively responsible for $\frac{|V|}{k}$ elements of d_v. For example, consider $|V| = 4$ and $p = 8$: this yields a partition size of $k = 2$. In this case, the first processor of each partition is responsible for $d_{v,1}$, $d_{v,2}$ and the second processor is responsible for $d_{v,3}$ and $d_{v,4}$. Hereby, the total distance lists is $d_v = [d_{v,1}, d_{v,2}, d_{v,3}, d_{v,4}]$.



The `DijkstraSSSP` algorithm mainly consists of the repetition of two steps: First, the nearest node x in the distancelist d_v has to be found. For this node the shortest path already has been found. Afterwards the distance of all neighbors of x has to be adjusted in d_v.

These steps have to be altered as follows because for the parallelization d_v has been distributed across the partition:



1. Find the node x with the shortest distance in d_v.
   - Each processor owns a part of d_v: Each processor scans for the local minimum $\tilde{x}$ in his part, for example using linear search.
   - Compute the global minimum x in d_v by performing a reduce-operation across all $\tilde{x}$.
   - Broadcast the global minimum x to all nodes in the partition.
2. Adjust the distance of all neighbors of x in d_v
   - Every processors now knows the global nearest node x and its distance. Based on this information, adjust the neighbors of x in d_v which are managed by the corresponding processor.

The total runtime of such an iteration of `DijkstraSSSP` performed by a partition of size k can be derived based on the performed subtasks:

- The linear search for $\tilde{x}$: $O(\frac{|V|}{k})$
- Broadcast- and Reduce-operations: These can be implemented efficiently for example using binonmialtrees. This yields a communication overhead of $O(\log k)$.

For $|V|$-iterations this results in a total runtime of $O(|V|(\frac{|V|}{k} + \log k))$. After substituting the definition of k this yields the total runtime for `DijkstraAPSP`: $O(\frac{|V|^3}{p} + \log p)$.

The main benefit of this parallelization is that it is not required anymore that every processor stores the entire adjacency matrix. Instead, it is sufficient when each processor within a partition only stores the columns of the adjacency matrix of the nodes for which he is responsible. Given a partition size of k, each processor only has to store $\frac{|V|}{k}$ columns of the adjacency matrix. A downside, however, is that this parallelization comes with a communication overhead due to the reduce- and broadcast-operations.

Example

The graph used in this example is the one presented in the image with four nodes.

The goal is to compute the distancematrix with $p = 8$ processors. For this reason, the processors are divided into four partitions with two processors each. For the illustration we focus on the partition which is responsible for the computation of the shortest paths from node **A** to all other nodes. Let the processors of this partition be named **p1** and **p2**.

The computation of the distancelist across the different iterations is visualized in the second image.

The top row in the image corresponds to d_A after the initialization, the bottom one to d_A after the termination of the algorithm. The nodes are distributed in a way that **p1** is responsible for the nodes **A** and **B**, while **p2** is responsible for **C** and**D**. The distancelist d_A is distributed according to this. For the second iteration the subtasks executed are shown explicitly in the image:

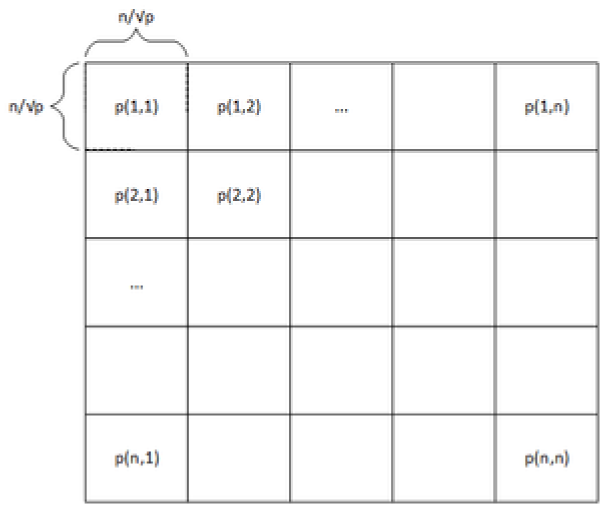
1. Computation of the local minimum node in d_A
2. Computation of the globalminimum node in d_A through a reduce operation
3. Broadcast of the global minimum node in d_A
4. Marking of the global nearest node as "finished" and adjusting the distance of its neighbors

Floyd algorithm

The Floyd algorithm solves the All-Pair-Shortest-Paths problem for directed graphs. With the adjacency matrix of a graph as input, it calculates shorter paths iterative. After $|V|$ iterations the distance-matrix contains all the shortest paths. The following describes a sequentiel version of the algorithm in pseudo code:

```
func Floyd_All_Pairs_SP(A) {
    $D^{(0)}$ = A;
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                $d_{i,j}^{(k)} := \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$
 }
```

Where A is the adjacency matrix, n = $|V|$ the number of nodes and D the distance matrix. For a more detailed description of the sequential algorithm look up Floyd–Warshall algorithm.



partition of a matrix with 2-D block mapping

Parallelization

The basic idea to parallelize the algorithm is to partition the matrix and split the computation between the processes. Each process is assigned to a specific part of the matrix. A common way to achieve this is **2-D Block Mapping**. Here the matrix is partitioned into squares of the same size and each square gets assigned to a process. For a $n \times n$-matrix and p processes each process calculates a $n/\sqrt{p} \times n/\sqrt{p}$ sized part of the distance matrix. For $p = n^2$ processes each would get assigned to exactly one element of the matrix. Because of that the parallelization only scales to a maximum of n^2 processes. In the following we refer with $p_{i,j}$ to the process that is assigned to the square in the i-th row and the j-th column.

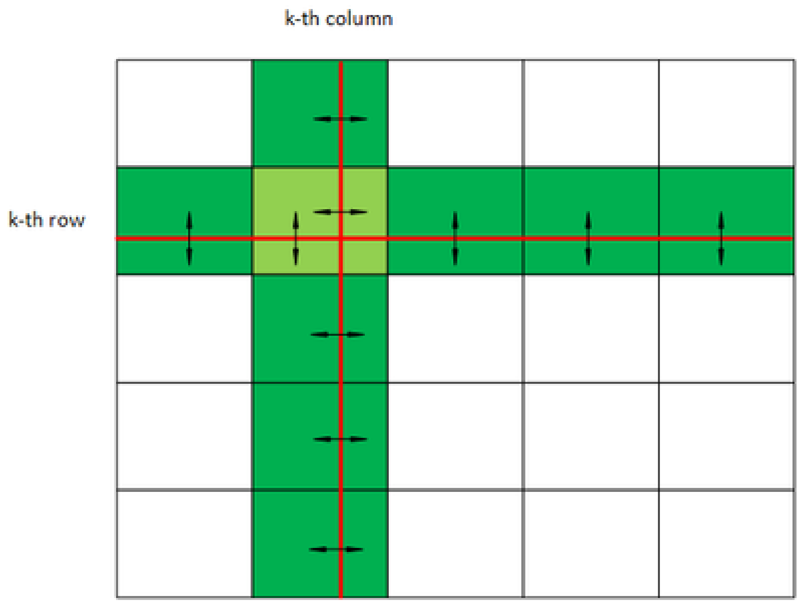
As the calculation of the parts of the distance matrix is dependent on results from other parts the processes have to communicate between each other and exchange data. In the following we refer with $d_{i,j}^{(k)}$ to the element of the i-th row and j-th column of the distance matrix after the k-th iteration. To calculate $d_{i,j}^{(k)}$ we need the elements $d_{i,j}^{(k-1)}$, $d_{i,k}^{(k-1)}$ and $d_{k,j}^{(k-1)}$ as specified in line 6 of the algorithm. $d_{i,j}^{(k-1)}$ is available to each processe as it was calculated by itself in the previous iteration.

Additionally each process needs a part of the k-th row and the k-th column of the D^{k-1} matrix. The $d_{i,k}^{(k-1)}$ element holds a process in the same row and the $d_{k,j}^{(k-1)}$ element holds a process in the same column as the process that wants to compute $d_{i,j}^{(k)}$. Each process that calculated a part of the k-th row in the D^{k-1} matrix has to send this part to all processes in its column. Each process that calculated a part of the k-th column in the D^{k-1} matrix has to send this part to all processes in its row. All this processes have to do a one-to-all-broadcast operation along the row or the column. The data dependencies are illustrated in the image below.

For the 2-D block mapping we have to modify the algorithm as follows:

```
1    func Floyd_All_Pairs_Parallel($D^{(0)}$) {
2      for $k$ := 1 to $n$ do{
3          Each process $p_{i,j}$ that has a segment of the k-th row of $D^{(k-1)}$,
           broadcasts it to the $p_{*,j}$ processes;
4          Each process $p_{i,j}$ that has a segment of the k-th column of $D^{(k-1)}$,
           broadcasts it to the $p_{i,*}$ processes;
5          Each process waits to receive the needed segments;
6          Each process computes its part of the $D^{(k)}$ matrix;
7          }
8     }
```

In line 5 of the algorithm we have a synchronisation step to ensure that all processes have the data necessary to compute the next iteration. To improve the runtime of the algorithm we can remove the synchronisation step without affecting the correctness of the algorithm. To achieve that each process starts the computation as soon as it has the data necessary to compute its part of the matrix. This version of the algorithm is called **pipelined 2-D block mapping**.



data dependencies in Floyd algorithm

Runtime

The runtime of the sequential algorithm is determined by the triple nested for loop. The computation in line 6 can be done in constant time ($O(1)$). Therefor the runtime of the sequential algorithm is $O(n^3)$.

2-D block mapping

The runtime of the parallelized algorithm consists of two parts. The time for the computation and the part for communication and data transfer between the processes.

As there is no additional computation in the algorithm and the computation is split equally among the p processes, we have a runtime of $O(n^3/p)$ for the computational part.

In each iteration of the algorithm there is a one-to-all broadcast operation performed along the row and column of the processes. There are $n/\sqrt{p}$ elements broadcast. Afterwards there is a synchronisation step performed. How much time these operations take is highly dependent on the architecture of the parallel system used. Therefore, the time needed for communication and data transfer in the algorithm is $T_{\text{comm}} = n(T_{\text{synch}} + T_{\text{broadcast}})$.

For the whole algorithm we have the following runtime:

$$T = O\left(\frac{n^3}{p}\right) + n(T_{\text{synch}} + T_{\text{broadcast}})$$

Pipelined 2-D block mapping

For the runtime of the data transfer between the processes in the pipelined version of the algorithm we assume that a process can transfer k elements to a neighbouring process in $O(k)$ time. In every step there are $n/\sqrt{p}$ elements of a row or a column send to a neighbouring process. Such a step takes $O(n/\sqrt{p})$ time. After $\sqrt{p}$ steps the relevant data of the first row and column arrive at process $p_{\sqrt{p},\sqrt{p}}$ (in $O(n)$ time).

The values of successive rows and columns follow after time $O(n^2/p)$ in a pipelined mode. Process $p_{\sqrt{p},\sqrt{p}}$ finishes its last computation after O(n^3/p) + O(n) time. Therefore, the additional time needed for communication in the pipelined version is $O(n)$.

The overall runtime for the pipelined version of the algorithm is:

$$T = O\left(\frac{n^3}{p}\right) + O(n)$$

References

Bibliography

- Grama, A.: *Introduction to parallel computing.* Pearson Education, 2003.
- Kumar, V.: *Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem (http://www.academia.edu/download/46545236/Scalability_of_parallel_algorithms_for_t20160616-15656-rc5uyt.pdf).* Journal of Parallel and Distributed Programming 13, 1991.
- Foster, I.: *Designing and Building Parallel Programs* (Online).
- Bindell, Fall: *Parallel All-Pairs Shortest Paths (http://www.cs.cornell.edu/~bindel/class/cs5220-f11/code/path.pdf)* Applications of Parallel Computers, 2011.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Parallel_all-pairs_shortest_path_algorithm&oldid=901599021"

This page was last edited on 13 June 2019, at 00:27 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.