Discrete diffusion algorithm implementation

Petr Smrček
Thursday 11:00
Open Informatics
smrcepet@fel.cvut.cz

***Abstract*—The paper contains definition, description of implementation and evaluation of discrete diffusion algorithm described in [4]. This algorithm distributes tasks of varying size among multiple processing units so that their final load is as even as possible and the overall communication is minimal. The tasks cannot be split, which causes the problem to be NP-hard.**

I. Assignment

A. Problem Statement

The target of this project is to implement and evaluate discrete diffusion algorithm described in [4]. The problem targeted by this algorithm is a distribution of tasks among multiple processing units. Each task might be of varying difficulty, so different tasks might consume different amount of resources of the processing units. The algorithm's main purpose is to distribute the tasks so that every processor has as even load as possible. Single task cannot be split among multiple processors, they are already defined as atomic. The algorithm should also minimize the communication load between processing units caused by task exchange. Not all processing units can exchange tasks directly, the network can be arbitrary undirected connected graph.

The algorithm input is formally defined the following way: N denotes the set of the nodes (processing units), $C \subseteq N \times N$ the connections and P the tasks. Each task $p_i \in P$ is initially assigned to some arbitrary node $n \in N$ and causes unique load $w_i \in R_0^+$ defined abstractly as fraction of its resource requirement and resources available.

The quality of the algorithm is measured in the number of iterations required to achieve a balanced state, the amount of load moved along the connections and the difference between the optimal average load and the real load on the nodes, defined as the following mean deviation:

$$v = \frac{\sum_{j=1}^{|N|} |\overline{w} - w_j|}{|N|} \tag{1}$$

$\overline{w}$ is the optimal average load that each node would have, if the tasks could be split. w_j is a sum of loads of all tasks, that are assigned to node n_j.

B. Problem Categorization

The general problem is similar to the open-shop scheduling problem [6] or to optimized version of partition problem with multiple partitions. The set of all tasks is divided among multiple partitions (processing units) as evenly as possible. The slight difference from the optimized partitioning is that the task loads are not integers, but real numbers. The analogy to the open-shop scheduling might be more accurate, if we imagine that the tasks on a single processing unit are executed one after another in arbitrary order and the goal is to minimize the overall makespan. Because the number of tasks and nodes can be arbitrary, the problem is generally NP-hard.

Due to its iterative nature, the algorithm contains another NP-hard subproblem. In each iteration, specific amount of load has to be transferred between each pair of nodes. However, the tasks are atomic, so the problem which tasks should be picked is similar to knapsack. Since this problem has to be solved in each iteration for each connection, only very rough and fast heuristic should be used.

Last, the overall solution is made even more difficult by the requirement for minimal inter-node communication.

II. Related works

This paper targets description, implementation and evaluation of the discrete diffuse algorithm based on the algorithm described in [4]. The original paper presents the algorithm in combination with heuristics for balancing the tasks among software and hardware on a single node. We decided to focus only on the original discrete diffuse algorithm, because the SW/HW evening makes the problem already very specific, while the diffuse algorithm itself is more general and can fit more applications.

The first local iterative diffuse algorithm is presented in [1]. The paper contains the idea of iterations and the rules defining how much load should be distributed to which nodes in every iteration so that the iterations converge to the average load $\overline{w}$. However, the number of iterations might be large and is generally not known before the algorithm is run.

The algorithm described in this paper is a discrete version of optimized diffusion algorithm presented in [3]. The optimized algorithm is based on computing the spectrum of the network graph. The spectrum is a set of eigenvalues of the adjacency matrix of the graph. The computation of eigenvalues for arbitrary large graphs might be very time-consuming, but the algorithm then needs only $m-1$ iterations where m is the number of distinct eigenvalues. More, the flow is l_2-optimal which means that the l_2-norm defined as

$$||x||_2 = (\sum_{i=1}^{N} x_i^2)^{\frac{1}{2}} \tag{2}$$

is minimal for the vector of flows x. The idea of computing eigenvalues and achieving l_2-optimality was first introduced in [2], however the optimized algorithm presented in [3] is much simpler without any performance tradeoff.

The discrete version of the algorithm presented in [4] deals with the atomicity of tasks. This restriction breaks the assumption of the original diffuse algorithm presented in [1], since the amount of load exchanged in each iteration cannot be arbitrary.

III. Problem solution

A. Design

At the beginning the eigenvalues of the network graph Laplacian matrix must be computed or already known. The algorithm then runs $m-1$ exchange iterations where m is the number of distinct eigenvalues. In each iteration, each node communicates and potentially exchanges load with all adjacent nodes. After the $m-1$ iterations, error correcting iterations are run until the load error on every node (against the optimal average load) is not below certain threshold. The target of the solution is to minimize the amount of iterations and load exchange.

The optimal load exchange y_c^k between nodes n_i, n_j connected by edge c in iteration k is defined as

$$y_c^k = \alpha(w_i^k - w_j^k) \tag{3}$$

where w_i^k is the load of node n_i at the beginning of iteration k and α is a parameter determining the fraction of load to send. The optimal choice of parameter α is described in [3]. It's value changes in every iteration and is determined by precomputed eigenvalues based on the following formula:

$$\alpha = \frac{1}{\lambda_k} \tag{4}$$

In order to have the flows distributed as evenly as possible trough all iterations, the $m-1$ eigenvalues (without zero eigenvalue) should be sorted based on value and applied in the center-started order:

$$\begin{aligned} \lambda_{\frac{m}{2}}, \lambda_{\frac{m}{2}-1}, \lambda_{\frac{m}{2}+1}, \lambda_{\frac{m}{2}-2}, ... \qquad & \text{for even } m \\ \lambda_{\frac{m-1}{2}}, \lambda_{\frac{m-1}{2}+1}, \lambda_{\frac{m-1}{2}-1}, \lambda_{\frac{m-1}{2}+2}, ... \qquad & \text{for odd } m \end{aligned} \tag{5}$$

The problem of the discrete version of the algorithm is in the atomicity of tasks. Even though we can compute optimal load that should be exchanged by adjacent nodes in any given time, it is not easy to actually send it. We have to choose some subset of tasks currently present on the node and it is likely that the sum of their loads won't fulfill the optimal load. Because this selection is NP-hard problem and we have to do it for every edge in every iteration, simple heuristic should be used. The method suggested in [4] is to randomly select tasks as long as the optimal value is not exceeded. The sum of loads of the selected tasks is defined as $ydisc_c^k$ and obviously $ydisc_c^k \leq y_c^k$. The error created by this approximation propagates to next iterations and is defined as:

$$e_c^k = y_c^k + e_c^{k-1} - ydisc_c^k \qquad \text{with} \qquad e_c^0 = 0 \tag{6}$$

The presence of this error has 2 consequences. First, the load sent is smaller than it would optimally be and we should try to compensate this fact by increasing the discrete flow limit in next iteration, as is suggested in [4]:

$$ydisc_c^k \leq y_c^k + e_c^{k-1} \tag{7}$$

Second, if the error is present, the next iteration node loads will be different from the optimum computed in [3]. Thus, in next iteration, y_c^k might be different from the optimal value. We believe that by not addressing this issue, the proofs about the maximal final error in [4] are incorrect, starting with their equation (14). Our observations support the fact. This issue can be compensated by always making the computation of y_c^k based on the loads the nodes would have in the optimal algorithm:

$$y_c^k = \alpha((w_i^k + \sum_{z=\{i,x\}\in C} e_z^k) - (w_j^k + \sum_{z=\{j,x\}\in C} e_z^k))) \tag{8}$$

However, because it might not be possible to transport optimal amount of load in every iteration, the error after $m-1$ iterations can still be non-zero. To achieve the reasonable final error, we run correcting iterations:

$$ydisc_c^k \leq e_c^{k-1} \tag{9}$$

Note that for the correcting iterations, the optimal flow $y_c^k = 0$ and the remaining error is still computed according to (6). The correcting iterations stop when for all nodes:

$$|\overline{w} - w_i^k| < d_i * S_{max} \tag{10}$$

where d_i is the degree of node n_i and S_{max} is the load of the biggest task. This treshold was incorrectly proven to be always achieved after just one correcting iteration in [4]. We believe that this is not true and possibly more error correcting iterations might be needed. We have chosen the same treshold as [4], because we believe it is the best generally reachable value. If $w_i^k \geq \overline{w} + d_i * S_{max}$, al least one of the edges of this node has to have bigger error than S_{max}, so the node can send at least one task. The same applies for the value lower than average (the node can receive at least one task). We don't know what the upper bound on the number of correcting iterations is, however in all our simulations it stayed very small. The final flow is always smaller than the flow of the optimal algorithm, because we compute y_c^k based on the optimal algorithm and at any point in the algorithm, and never exceed its overall value (in any iteration we send more than y_c^k only when we sent less in previous iterations).

B. Implementation

The algorithm was implemented in JavaSE with usage of Jama library [5] for eigenvalue computation. 3 versions of the algorithm were implemented:

- Original non-discrete algorithm presented in [3]
- The same non-discrete algorithm with one change - any node in any iteration cannot send more load than it currently has. The original algorithm ensures there is never negative load on any node in any iteration. However, the nodes can send more than they currently have (provided that in next iteration, they will also receive some load).
- Discrete algorithm presented in [4] with small modifications described in III-A.

The input data are provided by file, whose relative path has to be specified as the first argument. The input file has the following format. There is a single integer value n on the first line specifying the number of nodes of the network graph. Then there are n lines (one for each node) with arbitrary number of double values. Each double value represents one task (its size) initially present on respective node. If there are no initial tasks for some node, the line is empty. Finally there are another n lines, each containing n values, either 1 or 0. This is the adjacency matrix. The nodes in first n lines with tasks are in the same order as the nodes of adjacency matrix.

By default, the program will execute the discrete algorithm. You can select different algorithm by passing optional second argument: *cont* or *cont-mod*.

The skeleton of the implementation is the following. First the input is parsed and the nodes, edges and tasks created and correctly connected. Then, the eigenvalues are computed and the eigenvalue iterations take place. In every iteration, current nodes loads (the values the optimal algorithm would have) are computed and saved, so the loads won't be changed in the middle of iteration by sending some tasks. Then for each edge, the amount that should be sent is computed and tasks from source node are selected and pushed to the edge. After all edges are processed, the tasks they contain are send to destination nodes. The two steps here are in order to prevent nodes from receiving any tasks before selecting their own tasks to send (so they can send only what they really have in given iteration). Finally, error correcting iterations with the same logic take place until the treshold is reached. The results are printed to standard output.

For the task selection knapsack problem, we haven't used random selection, but simpler solution which repeatedly selects the biggest feasible task available, as long as the limit allows.

Fig. 1: Number of error correcting iterations for discrete and continuous modified algorithm.

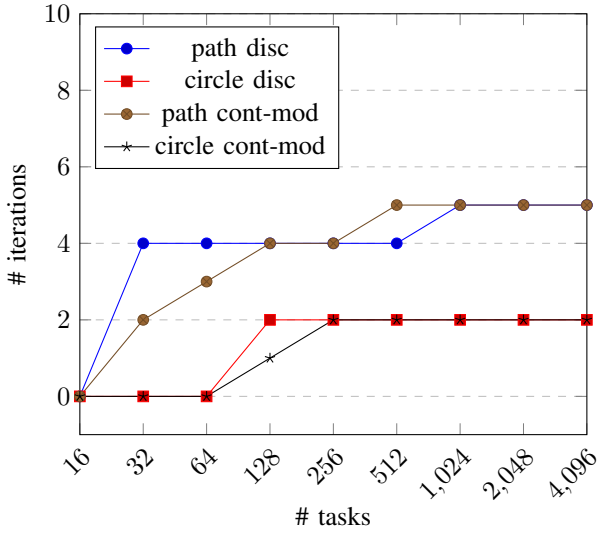
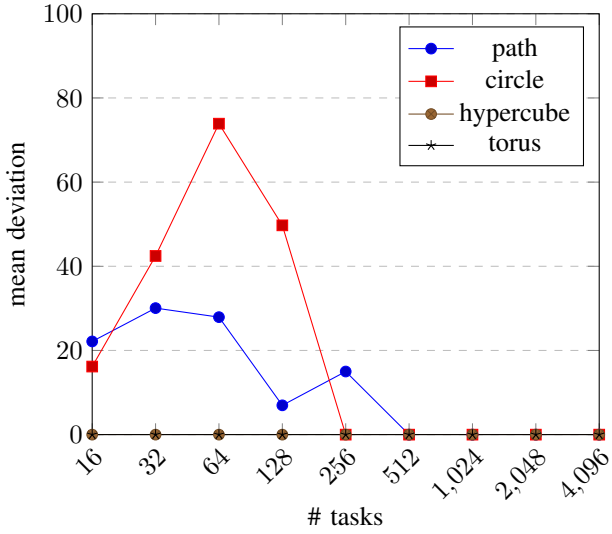


Fig. 2: Mean deviation of modified continuous algorithm for all tasks starting on single node.



IV. Experiments

A. Benchmark Settings

Experiments were run on 16 node graphs of various structures - path, circle, hypercube and torus. Varying number of tasks of size from 1 to 100 were added to either the first node, which is the worst case scenario of the distribution, or evenly distributed (each node got same number of tasks, but randomly selected). The algorithm was run in all 3 versions in order to compare the results. The tasks were generated randomly and then saved and reused so each version ran with the same tasks. Because the termination condition depends on the size of the biggest task, one task had always the size 100 in order not to bias the final precision of scenarios with varying task counts. The algorithm is not computationally heavy at all, since all heuristics are very simple. Thus we haven't measured its CPU load or memory consumption, since these values were for our scenarios practically immeasurable.

B. Results

Figure 1 shows the number of error correcting iterations. Torus and hypercube never needed any of these iterations, as well as any of the scenarios where the tasks were initially evenly distributed or obviously the original continuous version of the algorithm.

Figure 2 shows the only cases where the modified continuous algorithm wasn't absolutely precise, which were the smaller numbers of tasks initially all in one node on path or circle. These were obviously also the only cases the modified continuous algorithm used less flow than the original.

Figure 3 shows the final mean deviation (defined in equation (1)) for the discrete algorithm across all nodes. Concretely, figure 3a shows the mean deviation for the scenario where all tasks were initially put to a single node and figure 3b the scenario with initially even distribution.

Finally figure 4 shows the flow of discrete algorithm as a portion of the original continuous algorithm on the same problem instance. The flow values were compared in n_2-norm. Again, figure 3a represents all load initially on a single node and figure 3b even distribution.

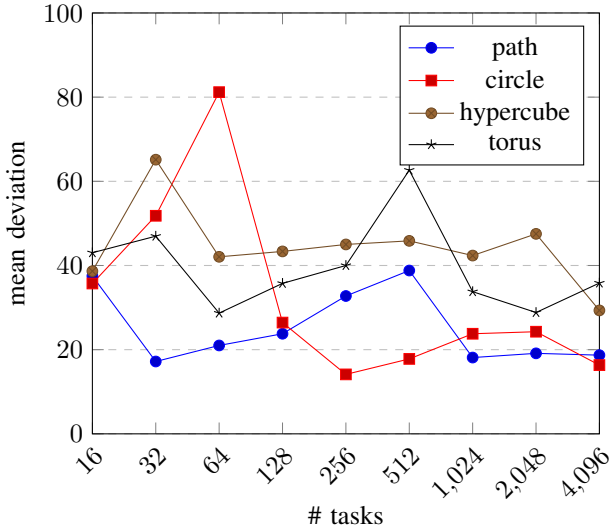
C. Discussion

The experiments confirm that more than one error correcting iteration might be needed and that the discrete algorithm flow is always smaller than for the original continuous version.

We were pleased to see that the number of required error correcting iterations was very small. Although the number tends to slowly increase with the number of tasks, the tendency is so slow that we don't expect it to increase much higher. Also, the error correcting iterations are needed only for the worst cases: all tasks starting on a single node on a graph topology, that doesn't really allow effective diffusion. It might be interesting to test whether a different task selecting heuristic would change these numbers, since it appears, that with the higher number of tasks it is more likely, that mostly big tasks are present on some nodes, so the selection might actually suffer from this. Very interesting is the fact, that the number of these iterations for modified

Fig. 3: Mean deviation of discrete algorithm.

(a) All tasks are initially on a single node.



(b) Tasks are initially distributed evenly.

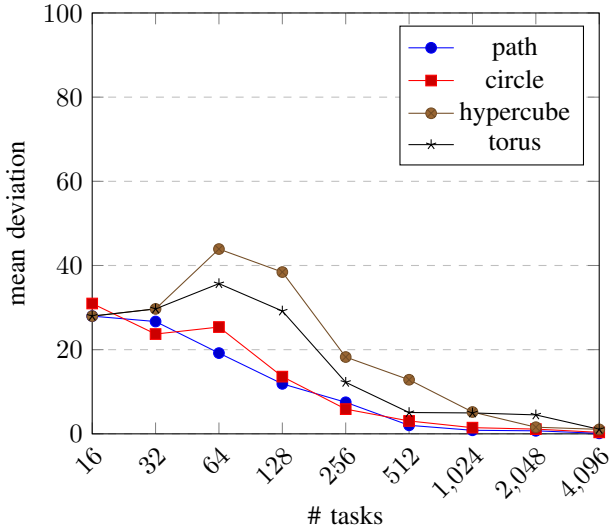
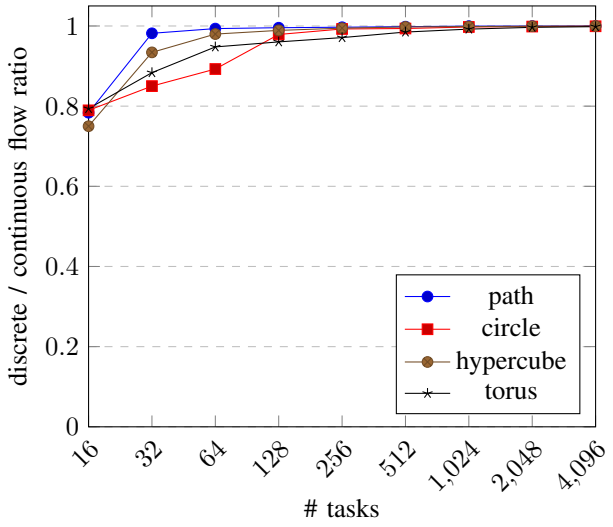
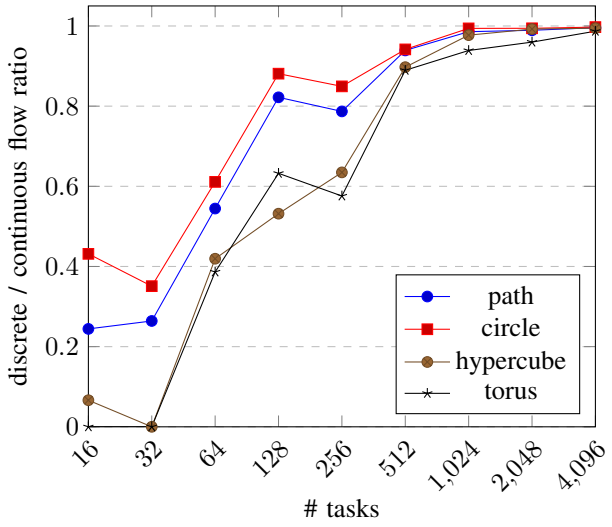


Fig. 4: Discrete algorithm flow l_2-norm / continuous algorithm flow l_2-norm ratio.

(a) All tasks are initially on a single node.



(b) Tasks are initially distributed evenly.



continuous algorithm is only slightly lower than for the discrete version. It might suggest that the need for these iterations is not mostly caused by the atomicity of the tasks, but by the inability to send more than the node currently has.

The mean deviation of the discrete algorithm is not optimal, as expected. The experiments show that if the scenario is hard (all tasks start on single node), the final mean deviation tends to be relatively high in all cases. On the other hand, if the scenario is easier (tasks evenly distributed), the mean deviation decreases with the incresing number of tasks. We believe the decrease is caused by the higher chances for the node to have almost exactly the task sizes needed to satisfy the optimal flow. In the hard scenario, the nodes might not receive the smaller tasks due to the heuristic selection, so the mean deviation decreases only slowly. Interestingly, the modified continuous algorithm is in most cases able to reach the optimal flow, even if it needed some error correcting iterations. It doesn't reach the optimal flow only in the hardest cases (only few tasks on a difficult topology). This fact might suggest, that the higher final mean deviation is mostly caused by the atomicity of tasks.

The flow of the discrete algorithm for the all tasks starting on a single node scenario is very similar to the original continuous algorithm flow. This is caused by the fact that most of the flow is used on simply delivering the load to empty nodes. As the

task count increases, this is even more apparent, since the mean deviation target treshold remains the same, while the overall load increases. On the other hand, for the evenly distributed scenario, the flow ratio increases as the mean deviation decreases. This is because all the nodes already have some load and most of the flow just lowers the mean deviation.

Finally we want to note that increasing the number of tasks is equivalent to keeping the load the same and increasing the granularity of the tasks (i.e. lowering the maximal task size). All test cases except the mean deviation would remain the same while the mean deviation would just be lowered proportionately to the maximal task size.

V. Conclusion

We have corrected, implemented and evaluated the discrete diffuse algorithm presented in [4]. The performance of the algorithm depends on the amount (or granularity) of the tasks and on the connectivity of the graph. However, the network congestion caused never exceeds the congestion of the non-discrete optimal algorithm described in [3] and the load deviation on each node is guaranteed to be under certain treshold, although few additional error correcting iterations might have to be run. Future work might explore the impact of different task selecting heuristics on the algorithm performance or finding the guaranteed upper bound of the number of error correcting iterations needed.

References

[1] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[2] Ralf Diekmann, Andreas Frommer, and Burkhard Monien. Efficient schemes for nearest neighbor load balancing. *Parallel computing*, 25(7):789–812, 1999.

[3] Robert Elsässer, Burkhard Monien, Robert Preis, and Andreas Frommer. Optimal and alternating-direction load balancing schemes. In *Euro-Par'99 Parallel Processing*, pages 280–290. Springer, 1999.

[4] Thilo Streichert, Christian Haubelt, and Jürgen Teich. Online hardware/software partitioning in networked embedded systems. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 982–985. ACM, 2005.

[5] National Institute of Standards and Technology The MathWorks. Jama.

[6] DP Williamson, LA Hall, JA Hoogeveen, CAJ Hurkens, Jan Karel Lenstra, SV Sevast'Janov, and DB Shmoys. Short shop schedules. *Operations Research*, 45(2):288–294, 1997.