Extended Dataflow Model For Automated Parallel Execution Of Algorithms

Maik Schumann*, Jörg Bargenda, Edgar Reetz and Gerhard Linß
*Department of Quality Assurance and Industrial Image Processing
Ilmenau University of Technology, Ilmenau, 98693 / Thuringia, Germany
Email: qualimess@tu-ilmenau.de

Abstract

The standard model of dataflow can express algorithms consisting of command sequences where each and every command is executed exactly once. This paper presents extensions to the model that allows control flow to be expressed within that model as well. Using these extensions algorithms containing branches, loops and even recursion can be composed as dataflow.

Keywords: parallel; programming; dataflow; control flow; declarative; graphical;

I. Introduction

Industrial tasks that are performed by computers become more and more complex and are invoked more frequently. Therefore it becomes more essential to optimize the software systems to reduce computation time and meet the user requirements in terms of latency and throughput. One way to enhance the performance of software is to utilize previously unused computation resources and do execution in parallel. In order to do that the developers need knowledge and experience in parallel programming which is a skill rarely seen especially in professional programmers who do not work in the field of general programming but rather have a specialized engineering background and solve associated problems by creating software. The aim of this paper is to provide a suitable model for algorithms that allows for automated parallel execution without additional effort in programming. In order to do that the existing model of dataflow is extended to include command flow structures to enable branches and loops to be incorporated in the model.

II. State Of The Art

Besides manual parallelization in terms of explicit multithreading there are multiple approaches of simplified ways to performance gain by parallel execution. On a low level layer of abstraction there are extensions to the processor's set of commands that enable vector arithmetic where identical operations are performed for each element of the given vector arguments [1]. The basic principle of OpenMP is an example of high level parallelization techniques where the individual iterations of a loop can be executed simultaneously [2], [3]. The common aspect of these and other approaches is the fact that they need to be applied inside the most basic operations in order to take away the additional considerations necessary in the development of parallel software. Depending on the kind of problems the software system is supposed to solve these basic tasks are not necessarily compatible to those approaches.

III. The Basic Model

The basic model of dataflow is a directed acyclic graph where each node is a black box operation that processes incoming data to provide output data [4], [5].

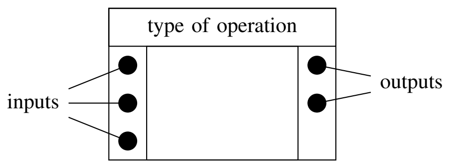


Fig. 1. Schematic representation of an operation as a black box with incoming and outgoing data ports.

As shown in fig. 1 the processing steps represented by these nodes generally take multiple (strongly typed) items of incoming data to produce multiple outputs. In terms of math they represent a mapping between a n-tuple of inputs and a m-tuple of outputs as shown in equ. 1 where each set X_i and Y_j represents the expected data type for the respective input or output.

$$f : X_1 \times X_2 \times \cdots \times X_n \to Y_1 \times Y_2 \times \cdots \times Y_m \quad (1)$$

The representation of an algorithm within this model therefore contains a set of operations that need to be performed and information about the origin of each input. In contrast to the standard representation as a fixed sequence of commands the dataflow does not specify the order of execution as a total order but as a partial order. Only the nodes which are directly or indirectly responsible for the computation of input for a certain other node need to be executed prior to that node. Therefore any two nodes A and B where neither A (indirectly) computes input for B or B computes data required for A can be considered independent of each other. If that is the case they can be executed in any order. A system that executes algorithms specified as dataflow can utilize this feature to

identify parts of an algorithm that can be executed in parallel.
Figure 2 represents an exemplary dataflow. The edges/arrows in this schematic for example state that the operation H processes output data of operations C and D. Including indirect dependencies this means that the operations A, B, C and D need to be completed, before H can be executed. The operation I on the other hand processes data produced by H so I cannot start before H is finished, but there is no such relation between H and any one of the operations E, F and G. As a result H can be computed in parallel to any of these three operations.

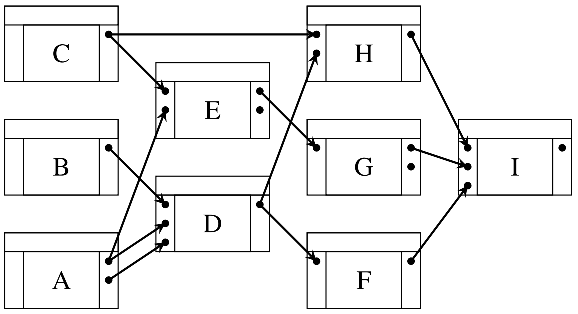


Fig. 2. Schematic dataflow containing multiple operations and their data links.

IV. The Model Extension

The basic model of flow processing is not limited to be used as a representation of algorithms. If the flow does describe an algorithm though, it has an additional restriction concerning the edges connecting the processing nodes: Each edge specifies the source for an argument of the target operation. Since the algorithm is deterministic this source information needs to be non-ambiguous. Therefore multiple edges with the same target node/operation that are associated with the same argument of the target operation are not allowed. This restriction limits algorithms to have linear flow of control which means that neither branches nor loops can be used.
In order to overcome this limitation this paper proposes an extension to the model of dataflow. In addition to the existing black boxes representing atomic operations white boxes are introduced. To the outside these white boxes have the same interface as the black boxes and therefore the processing system can treat them alike. Contrary to the atomic operations these white boxes or scopes also have a flow structure on the inside as depicted in fig. 3 where the input presented to the white box is passed on to the operations A and B and the data computed by D and E is used as output of the white box.
These local scopes enable the model to represent a whole algorithm or parts of it at different levels of abstraction by hiding or revealing their inner structure. Additionally the behavior of branching and looping can be implemented hidden within the transition between the outside and the inside of such a white box.

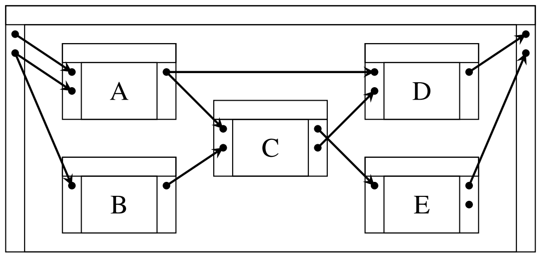


Fig. 3. Local scope or white box operation with inner dataflow structure.

For branching an outer scope can contain multiple inner scopes and choose the appropriate ’case’ during the transition. The incoming data is then forwarded to the chosen inner scope. After that scope finished its processing the output data it produced is forwarded to the outer scope and can then be passed on to the outside of the branching scope. The principle of this construct is shown in fig. 4.

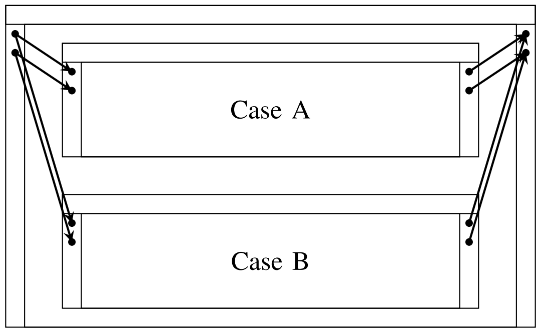
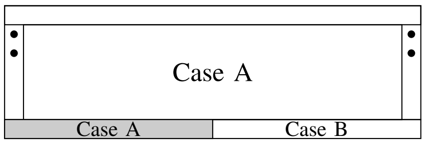


Fig. 4. Hidden implementation of a branching scope.

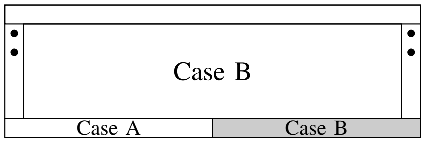
If it was up to the user to build this construct, it would contradict the restriction of non-ambiguity for data links. This problem can be solved be abstracting the branching construct and its behavior into a more general term of whitebox or scope. Each of the inner scopes is not part of the outer scope or its content, it is a view of the scope under certain circumstances. Therefore the ’operation’ the scope performs is a sequence of steps:

1) Evaluate incoming data to select the appropriate case
2) Forward the incoming data to the associated inner scope
3) Synchronously execute the selected inner scope
4) Dynamically connect the outgoing data ports of the selected inner scope to the respective ports of the outer scope
5) Forward the output data

Hence the user does not see multiple layers of scopes. Instead he or she is presented with a branching scope where only one of the cases/views is visible at any given time (see figure 5(a) and 5(b)). The stacked scope structure is implementation detail and therefore hidden from the user.
Loops can be constructed in a similar manner. An outer scope contains an inner scope which represents the body



(a) Case A selected



(b) Case B selected

Fig. 5. Representation of branching construct as a scope with multiple views.

of the loop. Before the first iteration of the loop the incoming data is forwarded to the inner scope. After each run of the body this forwarded data can be updated based on the intermediate output data effectively implementing data feedback which is essential in iterative processing. The principle structure of this construct is shown in fig. 6. Since the interface for incoming data generally does not match the output interface of the body the additional update links are neither mandatory nor does the incoming port with index i necessarily get its updated data from output port i. Arbitrary links are allowed.

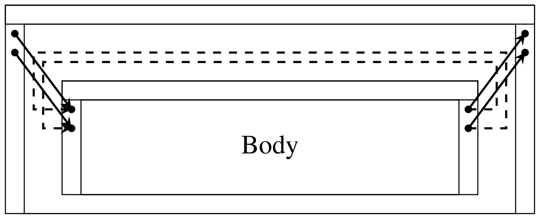


Fig. 6. Hidden implementation of a looping scope that checks the condition of continuation after each execution of the body.

The construct stated above is equivalent to a loop where the condition for continuing the iteration is checked after the body was processed. This means that the body runs at least once and the presence of (intermediate) output data is guaranteed. A loop that checks the condition before running the body needs to specify default data as output for the outer scope of the loop construct in case the body is not executed at all and therefore cannot produce any output. This can be done by combining the two constructs as shown in fig. 7.

Again this is the structure of implementation. To the user this can be presented in a manner similar to the views of branching. Figure 8 shows the graphical representation suitable for users with the multiple layers of scopes hidden.

Using the technique of white box abstraction of substructures it is even possible to enable the model to express recursion. If a scope can be used as a template for substructures it can also be used as a substructure of itself.

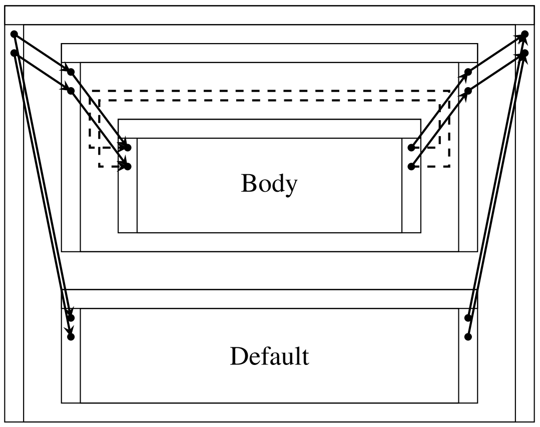
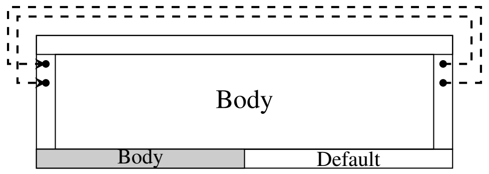
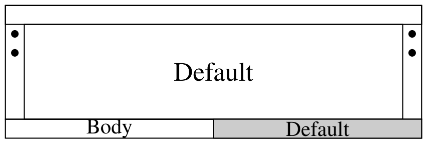


Fig. 7. Combination of branching and looping construct implementing a loop that checks the condition of continuation before execution of the body.



(a) Iterating body of the loop



(b) Default alternative

Fig. 8. Simplified representation of a loop as different views of the same scope.

As a consequence an algorithm execution engine that is based on this model is required to be able to either do the expansion of the structure dynamically at runtime or to separate the functionality and configuration of an operation from the data it actually calculates. The latter method allows for a static substructure (the content of the template scope) to be simultaneously used on different levels in the stack of recursive 'calls'. In that case the structure exists only once whereas there are multiple containers for the data calculated on each level.

V. Experimental Results

The algorithmic model described in this paper has been used for an automatic parallel algorithm execution engine which has been tested on algorithms of industrial image processing. The speedup of an algorithm executed by that engine in comparison to standard sequential execution is limited by the structure of that algorithm. If the operations processing some data mostly form long chains, these operations are not independent of each other and can only be executed in their original order. The maximum speedup of an algorithm can therefore be calculated based on its structure.

In addition to the limitation of speedup by the structure of the algorithm there also is a limitation due to overhead. Management and supervision of the operations cause additional workload where the amount of necessary processing depends mainly on the number of operations in the algorithm. Since this workload is approximately constant for a given structure, the limitation of speedup is determined by the ratio of overhead compared to operation workload. The measure of choice to determine this overhead ratio is the efficiency $E(m)$ as defined in equ. 2 where T_m is the total execution time when using m identical processors.

$$E(m) := \frac{T_1}{m \cdot T_m} \tag{2}$$

In order to determine the amount of overhead introduced by the sample implementation 1000 experiments per granularity $k \in K = \{-10, -9, \ldots, 9, 10\}$ with mean execution time per operation $\bar{t} = 2^k\,\mathrm{ms}$ were conducted on a quadcore processor based test system. The algorithm used for testing consisted of 1000 independent operations with random durations according to an exponential distribution. The median values of efficiency $E(4)$ are shown in figure 9.

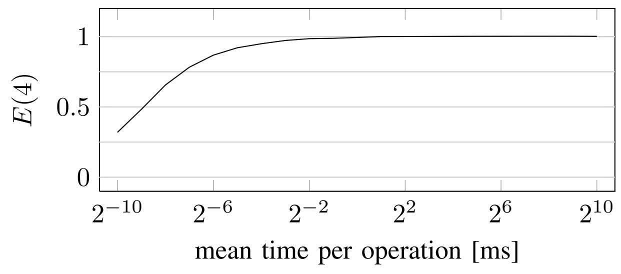


Fig. 9. Median quadcore efficiency plotted against mean time per operation.

Assuming the overhead introduces an amount α of additional workload per operation into the system, the limitation of speedup results in a maximum efficiency according to inequ. 3 for an algorithm containing n operations.

$$E(m) \leq \frac{T_1}{T_1 + \alpha \cdot n} \tag{3}$$

Using the Gaussian least squares method on the measured median values the overhead per operation on the exemplary implementation was determined to be $2.12\,\mu\mathrm{s}$.

The main goal of the algorithm modeling technique described in this paper is an automated speedup of real algorithms. In order to test the speedup performance the exemplary implementation can achieve an image processing based algorithm has been designed to solve a real world measurement problem as described in [6]. The setting of this problem is a dynamic measurement of the cutting edges on a pivot-mounted drill or endmill.

First the structure of the algorithm was analyzed to calculate a prognosis for the scaling behavior. This means that the achievable relative computation time was predicted for different numbers of available processors. This prognosis is shown in figure 10 as box-whiskers-plots.

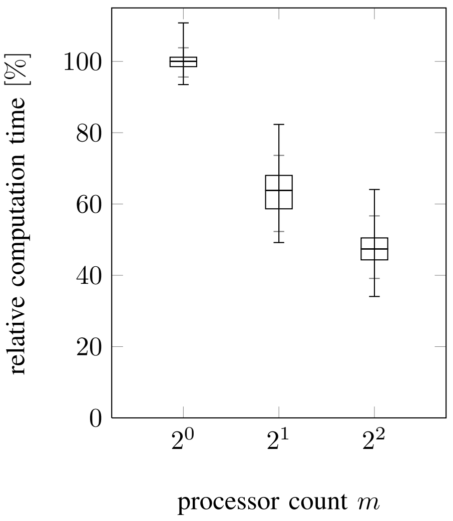


Fig. 10. Box-Whiskers-Plot of performance scaling prognosis normalized by the median computation time on a single processor.

This prognosis was created by taking the structure of the original algorithm while replacing the actual operations with placeholders that have adjustable workload. The workload of each placeholder was randomly set according to an exponential distribution using a mean execution time per operation of $\bar{t} = 2^{-6}\,\mathrm{ms}$ which is the approximated value for the given set of operations in this algorithm. As the distributions clearly show, the algorithm's structure can be sped up significantly by using multiple processor cores. On the other hand, the figure also shows, that the structure does not allow ideal parallelization which would result in a relative execution time of 0.5 for $m = 2$ processors and 0.25 for $m = 4$.

Figure 11 contains the distributions for 1000 actual test runs of the algorithm using identical input data for each of the 3 tested processor configurations.

A comparison of the measured and predicted scaling behaviors based on the median values of the respective distributions reveals that there is a 99.7% correlation. Additionally the efficiency predicted using the test algorithm's structure was calculated to be $E(1) = 0.8626$ which matches the limiting value of 0.8682 in figure 9 very well.

VI. Discussion

Using the dataflow based model for algorithm design and execution pursues two goals at the same time. First of all there is the benefit of speed that comes with parallel execution. Second of all there also is the additional benefit

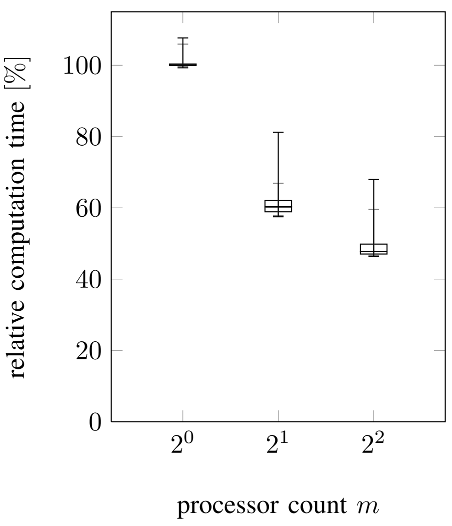


Fig. 11. Box-Whiskers-Plot of measured performance scaling normalized by the median computation time on a single processor.

of a more intuitive design environment for algorithms. Designing an algorithm as dataflow for example removes consideration of data storage and management from the list of things to do for the developer due to the fact that there are no explicit variables to store data in. The combination of simplifications like that and the fact that the algorithm can be implemented without having to write a single line of code makes such an environment more accessible to personnel without special programming training.

Using the model extensions to standard dataflow described in this paper it is possible to implement control flow features like branches and iterations into a data driven model. This allows an execution engine to automatically detect and exploit opportunities for parallelization. If the engine is capable of dynamic alterations of the structure or it handles output data and local configuration data of the operations separately it is even possible to express recursive algorithms or adapt the basic principle of OpenMP (simultaneous execution of multiple iterations of the same loop) into this system to further increase the utilization of potential for parallel execution.

The experiments conducted with the exemplary implementation have proven that it is also possible to effectively use the potential for parallelization within a given dataflow. There are two main influences to the scaling behavior of the system which essentially determines the potential benefit of using a dataflow based execution engine. The granularity of the set of commands available in the system has major influence on the performance gain since the management of parallel execution also introduces overhead that effectively cancels out any structure based speedup if the individual operations are too small. The exact limit of usefulness concerning the problem of granularity needs to be determined for any specific implementation. The implementation tested in this paper has shown that a set of commands that run for approximately $1\,\mu$s or less is not suitable to be processed with this engine. The image processing commands used in the test algorithm on the other hand can greatly benefit from this kind of parallel execution although the potential speedup can only be utilized at approximately 86%.

VII. Conclusion

This paper introduced an extension to the standard model of dataflow that allows control flow structures to be used in a data driven algorithm design environment. This can be used as a simplified way for programming while additionally speeding up the processing without the additional effort of manual parallelization. Using an experimental implementation of a graphical editor and an execution engine based on this model it has been proven that real world problems can be solved with this approach achieving a significant speedup over standard sequentially programmed code.

Acknowledgment

The support of the InnoProfile initiative Qualimess at Ilmenau University of Technology funded by the German Ministry of Education and Research BMBF is gratefully acknowledged.

References

[1] P. Herrmann, *Rechnerarchitektur*, 3rd ed. Braunschweig: Vieweg Verlag, November 2002.
[2] *OpenMP Application Program Interface*, 3rd ed., OpenMP Architecture Review Board, May 2008.
[3] L. Dagum and R. Menon, “Openmp: An industry-standard api for shared-memory programming,” *IEEE Computational Science & Engineering*, January-March 1998.
[4] J. Brehm, *Performance Analysis of Single and Multiprocessor Computing Systems*. Aachen: Shaker Verlag, 2000.
[5] K. Waldenschmidt, *Parallelrechner. Architektur-Systeme-Werkzeuge*. Wiesbaden: Teubner Verlag, 1995.
[6] M. Schumann et al., “Measuring edges on pivot-mounted objects during rotation,” in *Advanced Mathematical and Computational Tools in Metrology and Testing*, ser. Series on Advances in Mathematics for Applied Sciences, vol. 84, F. Pavese et al., Ed., vol. 9. Singapore: World Scientific, 2012, pp. 358–365.