CiSj
computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing
International Scientific Journal of Computing
ISSN 1727-6209

VISPAR: A VISUAL TOOL FOR DESIGNING PARALLEL PROGRAMS

Sergei Gorlatch [1], **Henry Kehbel** [2]

1) Technische Universitдt Berlin, Fachbereich Informatik
Sekr.FR 5-6, Franklinstr. 28/29, D-10587 Berlin, Germany
Email: gorlatch@cs.tu-berlin.de
2) Universitдt Passau, FMI, D-94030 Passau, Germany
Email: kehbel@fmi.uni-passau.de

Abstract: -*We describe VisPar - a new visual tool intended to support the programmer in the process of designing complex parallel applications. The novel features of the tool are as follows: support of both task and data parallelism and mixture thereof, use of analytical cost models for performance prediction, systematic program design by optimizing transformations, and visualization of the design process. We demonstrate the usage of VisPar on a relevant case study - the practically used Jpeg compression algorithm - and report on the current status of the tool implementation.*

Keywords: - *parallel computing, program design and optimization, visual support, Java*

INTRODUCTION

Recent advances in hardware make it possible to solve large application problems potentially much faster than before by exploiting several processors in parallel within one computer or by harnessing the power of many computers in a local network or over the Internet.

Many parallel applications have been designed with a specific target architecture mind. Unfortunately, this often leads to the situation that the wheel has to be reinvented for every new kind of parallel system. Especially the optimization of parallel performance by tuning the program to a particular machine is a complicated process that demands special knowledge of the user. Typical applications do not use parallel concepts

from the outset but are rather transformed step by step to fit into a parallel environment. This transformation process, when done *ad hoc*, is both error-prone and time-consuming.

Our approach is to support the design of parallel software using a visual tool that assists the user in systematically choosing the parallel software structure and estimating its quality. The approach is implemented in the new tool *VisPar* (*Vis*ual *Par*allelism), whose current implementation status is described in this paper.

The rest of the paper is structured as follows. In the next Section we discuss the *VisPar* model of parallelism and the supported optimization algorithms. Section three describes our case study - the Jpeg compression algorithm - which is being parallelized using the *VisPar* tool. We proceed in Section four with the implementation status report and future plans for *VisPar*,and conclude in the last Section with a brief discussion of the novelty of our approach.

MODELLING AND OPTIMIZING PARALLELISM IN VISPAR

The *VisPar* tool is designed to support two major kinds of parallelism:

• data parallelism - applying one operation (function, subroutine, etc.) simultaneously on different elements of a partitioned data structure;

• task parallelism - performing different operations (functions, subroutines, etc.) simultaneously.

The vast majority of parallel applications exhibit either one of these two kinds of parallelism or a mixed two-level structure, with task parallelism at the higher level and data parallelism within individual tasks.

Parallel algorithms are modelled in *VisPar* using the DAG (directed acyclic graph) model which is quite popular in the literature, see e.g.[1]. There are two levels of potential parallelism available in the graph: (1) every node of a graph represents a task that might be executed in parallel with some other tasks; (2) every task may have inherent data parallelism.

The ultimate goal of the design process is finding the optimal target program, and some cost model should be used to compare various design options with each other. We do not restrict ourselves to a particular cost model; similarly to [1], we only require the monotonicity in the maschine

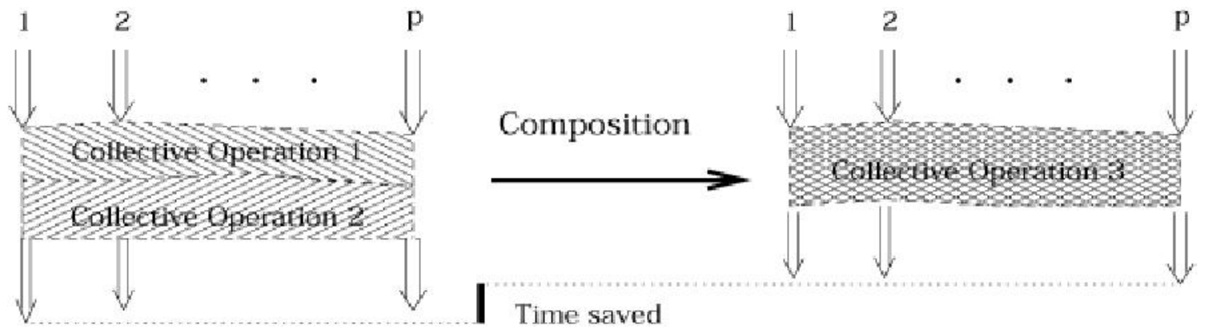

Fig. 1. ***The idea of optimizing data parallelism via composition in VisPar: Two collective operations on p processors are fused into one collective operation, thus saving synchronization costs and the total execution time***

parameters and the convexity in algorithm parameters. These, arguably rather reasonable constraints demand that, firstly, on a faster maschine the same algorithm will always require less time and, secondly, decreasing the costs of some part of an algorithm leads to decreasing total costs.

The *VisPar* tool expects the user to provide the cost annotations for particular parts of the program, which are then used by the optimization algorithms. Individual tasks are annotated with their maximal inherent parallelism and runtime costs. Potential parallelism among tasks, i.e. task parallelism, is expressed by means of the graph edges: they are the consequence of data dependencies and represent the data transfers between tasks. The edges may be labelled with the corresponding communication costs or the amount of transferred data.

We outline here the optimization algorithms that are being implemented in *VisPar* and can be called by the user during the design process. According to the structure of relevant parallel applications, these algorithms are divided in two groups: for optimizing data and for task parallelism, respectively.

Optimizing data parallelism. For the data-parallel part, we are going to use our own approach presented in [2]. Data-parallel tasks are described as sequences of parallel collective operations on groups of processors, which express either computations or communications or a mixture of both. Examples of operations that involve communications are scatter, reduction (involves also computations) and other collective operations of the current MPI (Message Passing Interface) standard.

It has been proved [2] that particular combinations of collective operations can be transformed into semantically equivalent but more efficient collective operations. An example of a composition transformation, which fuses two collective operations into one, is illustrated in Fig.1. Such transformations are formally proven to be semantics preserving. They are parameterized with respect to the basic operations performed on data, which enables their use in various applications. All available collective operations and their transformations with cost estimates are kept in an extendable library as part of *VisPar*.

Optimizing task parallelism. The transformation of task parallelism is usually based on graph-oriented models. We will briefly present two recent heuristic approaches, which are implemented in *VisPar*.

One of recent advances is the NoT (Network of Tasks) model [1]. It aims at optimizing the partitioning of the application into tasks and finding an optimal allocation of available processors to particular tasks, to minimize the total run time. The optimization makes use of two characteristics provided by the user - *m.a.p.* (maximal available parallelism) and the run time costs of a task. The NoT-graph has to be partitioned into layers, which is accomplished automatically. All tasks within a layer can be executed in parallel, therefore the available processors have to be partitioned among them. Intuitively, the idea is to reduce the number of processors allocated to the tasks with smaller costs, thereby slowing them down, and to increase the number processors of tasks with greater costs, thereby speeding them up. The allocation is obviously constrained by *m.a.p.* of a task. The described allocation adjustment will create a layer that consists of task that will have almost even runtime and will be work-balanced nearly optimally.

In the related approach from [3], the task graph is also layered, and the allocation and scheduling of each layer are optimized separately. This happens in two phases, bottom-up and top-down. Firstly, a layer is partitioned into sets, each consisting of two tasks.

For every pair of tasks, a decision is made whether the tasks should be executed after each other or rather simultaneously. When done for all

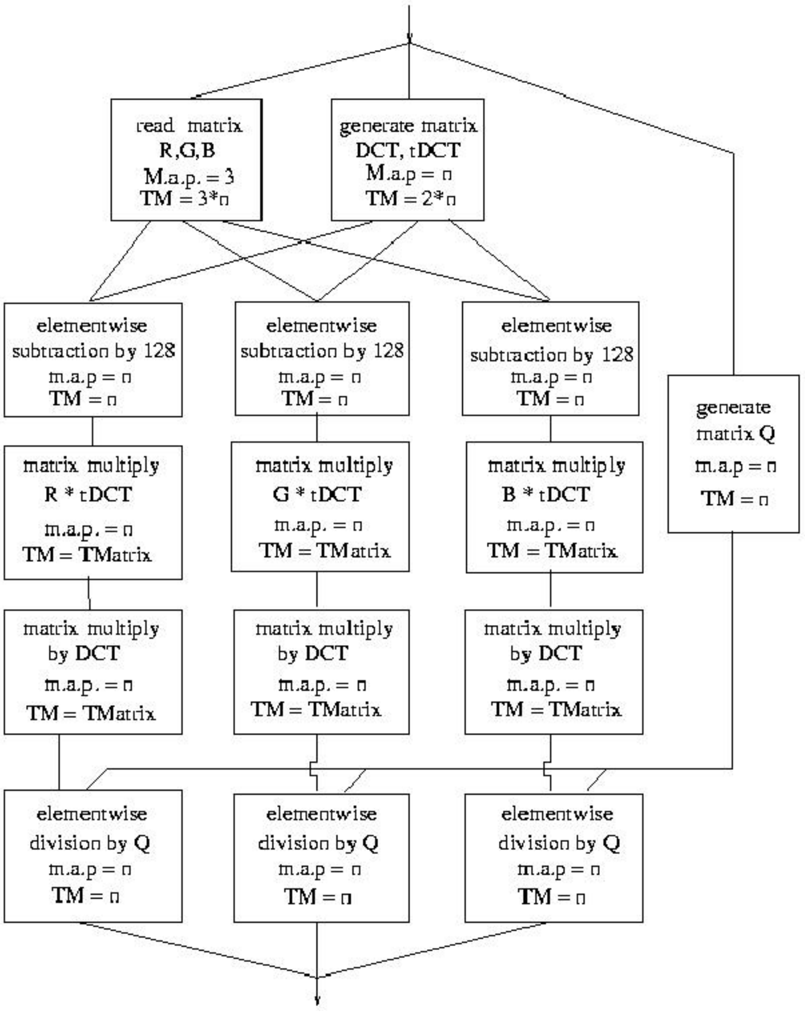
read matrix R,G,B M.a.p. = 3 TM = 3*n
generate matrix DCT, tDCT M.a.p = n TM = 2*n
elementwise subtraction by 128 m.a.p = n TM = n
generate matrix Q m.a.p = n TM = n
matrix multiply R * tDCT m.a.p. = n TM = TMatrix
matrix multiply G * tDCT m.a.p. = n TM = TMatrix
matrix multiply B * tDCT m.a.p. = n TM = TMatrix
matrix multiply by DCT m.a.p. = n TM = TMatrix
elementwise division by Q m.a.p = n TM = n

Fig. 2: ***Case Study: The initial representation of the Jpeg compression using DCT (Discrete Cosinus Transformation) on an RGB-coded (Red-Green-Blue) image. % consisting of the R,G,B 8x8 matrices with VisPar. The R, G and B matrices are sent to three different tasks that elementwise substract 128 and pass the result matrices to the next tasks. These multiply them with the transposed DCT-matrix (tDCT). After that the result matrices are multplied by the DCT-matrix. Matrix Q is built to reduce the resulting matrices to zero elements. The DCT-graph has been annotated with the maximal available parallelism m.a.p. and runtime TM.***

possible partitions into two-task sets, this creates a table showing sequential and parallel run times for every possible processor partition sizes. For every processor allocation, the minimal run time can be found and the respective pairs of tasks form new, combined tasks, whose number is a half of the initial task number. This procedure is repeated until we come to a task that combines all original tasks of the graph and for which the minimal run time is found. In the second, top-down phase, the processor allocation will be chosen using the information collected in the bottom-up phase. Starting with the root node of the combination tree, the available processors will be partitioned by searching for the minimal execution time. For every layer, the optimal scheduling and allocation are thus determined for the given number of available processors. It can be seen that the optimization algorithm of [3] can be rather time consuming for big applications.

TMatrix is a substitute for the runtime function

$$\frac{\left(\sqrt{n}\right)^3}{p} + 4 * \sqrt{p} * t_s + 4 * t_w * \frac{n}{\sqrt{p}} .$$

CASE STUDY: PARALLELIZING THE JPEG ALGORITHM

Our case study is the Jpeg algorithm for data compression. The essential part of the algorithm is the Dicrete Cosinus Transformation (DCT), which is a time-consuming operation amenable to parallelization. We take a version of Jpeg from [4].

The idea of using *VisPar* is that the user provides some initial version of the application struc-

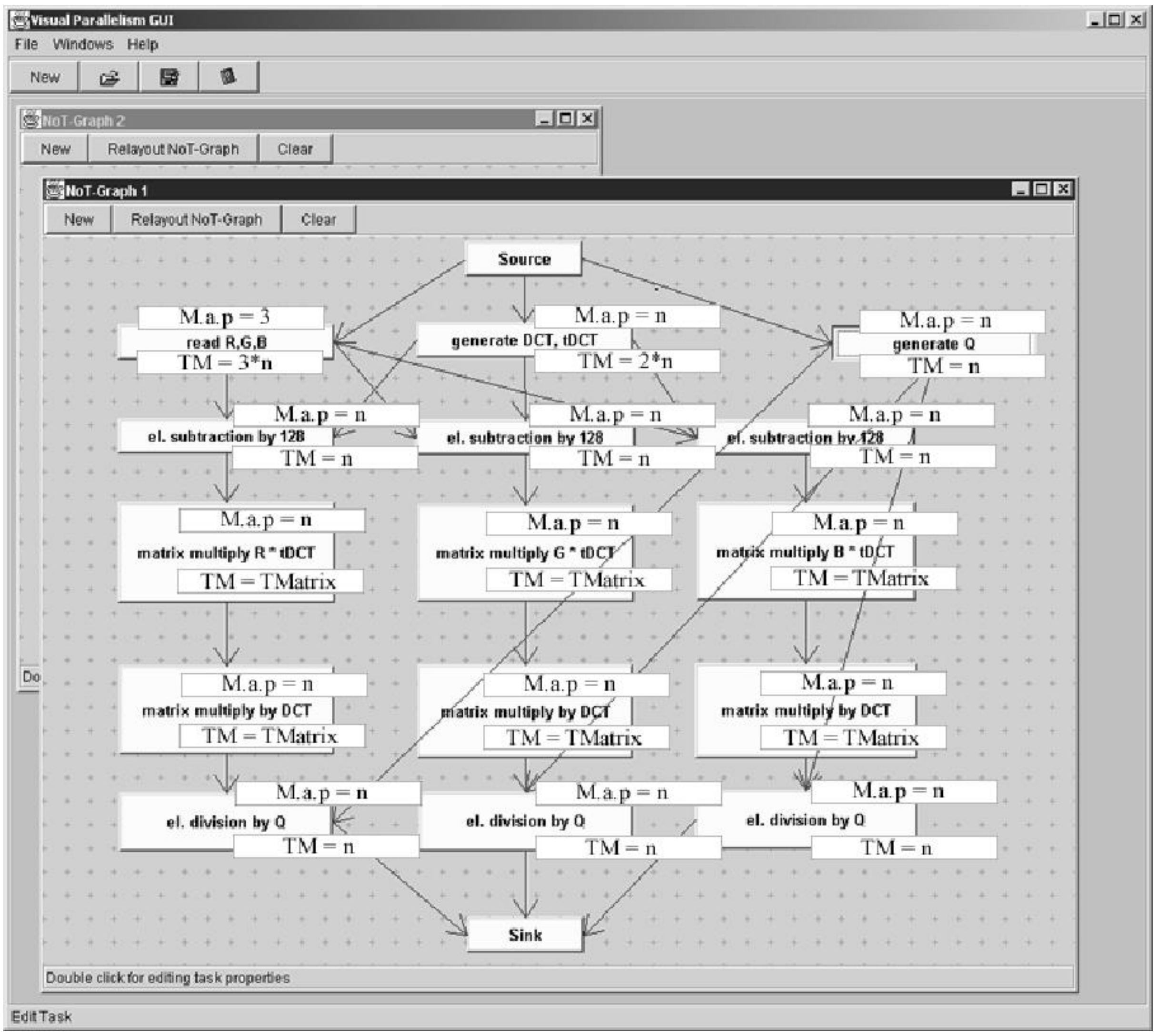

Fig. 3: ***The VisPar window with the canvas for the DCT-graph has been created with the maximal available parallelism m.a.p. and runtime TM. At this stage, no communication costs have still been added to the NoT-graph.***

ture in form of a task graph. The potential parallelism of the application is characterized by the *m.a.p.* and the run time costs of the tasks, and task interdependencies are specified by the graph edges. The *VisPar* tool supports the user in mapping these parallelism opportunities and parallelism constraints onto available processors.

The task graph in Fig.2 illustrates the overall structure of the case study. The DCT-compression algorithm works on a block of an RGB (Red-Green-Blue) colour picture consisting of the channel pictures: R,G and B each sized usually 8×8 pixels. So the typical matrix size of a channel picture is 64 Bytes, but in general case we will use parameter n. After generating the DCT and tDCT matrices and reading the chanel pictures, they have to be subtracted by 128 to gain colour values between -128 and +127. Then we use the transposed DCT matrix of size $\sqrt{n} \times \sqrt{n}$ to transform the colour values of the three channel pictures from the space domain to the frequency domain by performing the matrix multiplication with each channel picture matrix.

It remains to multiply the result matrices with the DCT matrix and divide them pointwise by matrix Q, so that after rounding most of the matrix values are zero and can be easily compressed by a Huffman-like algorithm.

The user of *VisPar* works with mouse-driven visual components, which enable to specify the described application structure graphically, see Fig.3. For each node of the graph, the estimates of the parallelism degree and of the run time are provided. Reading the three channel matrices with size n can be done in time 3*n. We assume that the channel matrices can be read in parallel, so that the *m.a.p.* is 3. Setting up the DCT and transposed DCT matrix will need 2*n steps and in both matrices the value can be generated independently, so the *m.a.p.* is n. Subtracting 128 from all matrix values costs n steps and can be done in parallel. The following matrix multiplications using the *Cannon/ Gentleman* algorithm on p processors requires time $\frac{(\sqrt{n})^3}{p} + 4 * \sqrt{p} * t_s + 4 * t_w * \frac{n}{\sqrt{p}}$ where t_s is the startup time and t_w ist the time needed to transfer a data word between two processors.

There may exist several graph windows, each being an implementation of the application or its parts; the user can merge and divide the graphs to improve the design. Besides annotation each task with a runtime function *TM* and the *m.a.p.*, the user can also open the second editor level to compose a task as a sequence of collective operations. In this case, the runtime costs can be computed automaticly from the costs of the single operations.

After creating the NoT-graph, the user can refine the application by modifying the graph or by adding alternative, semantically equivalent implementations of tasks. In particular, different data partitioning and communication pattern leads to bunch of possible implementations of one algorithm. After applying the optimization algorithms for both data and task parallelism

mentioned in first section , the automatic selection algorithm performed by *VisPar* chooses the best alternative, which may be nearly optimal because of rounding towards an integer value for the number of processors.

For illustration purposes, we demonstrate the task parallel optimization for the DCT example of Fig.2. To compute the processor allocation, equation (1) is used, where p_i is the number of allocated processors for task i, TM_{slow} is the runtime of the slowest task of the layer and TM_i is the runtime of task i of the layer:

$$p_i = \max\left(\left\lfloor \frac{m.a.p._i * TM_{slow}}{TM_i} \right\rfloor, 1\right) \qquad (1)$$

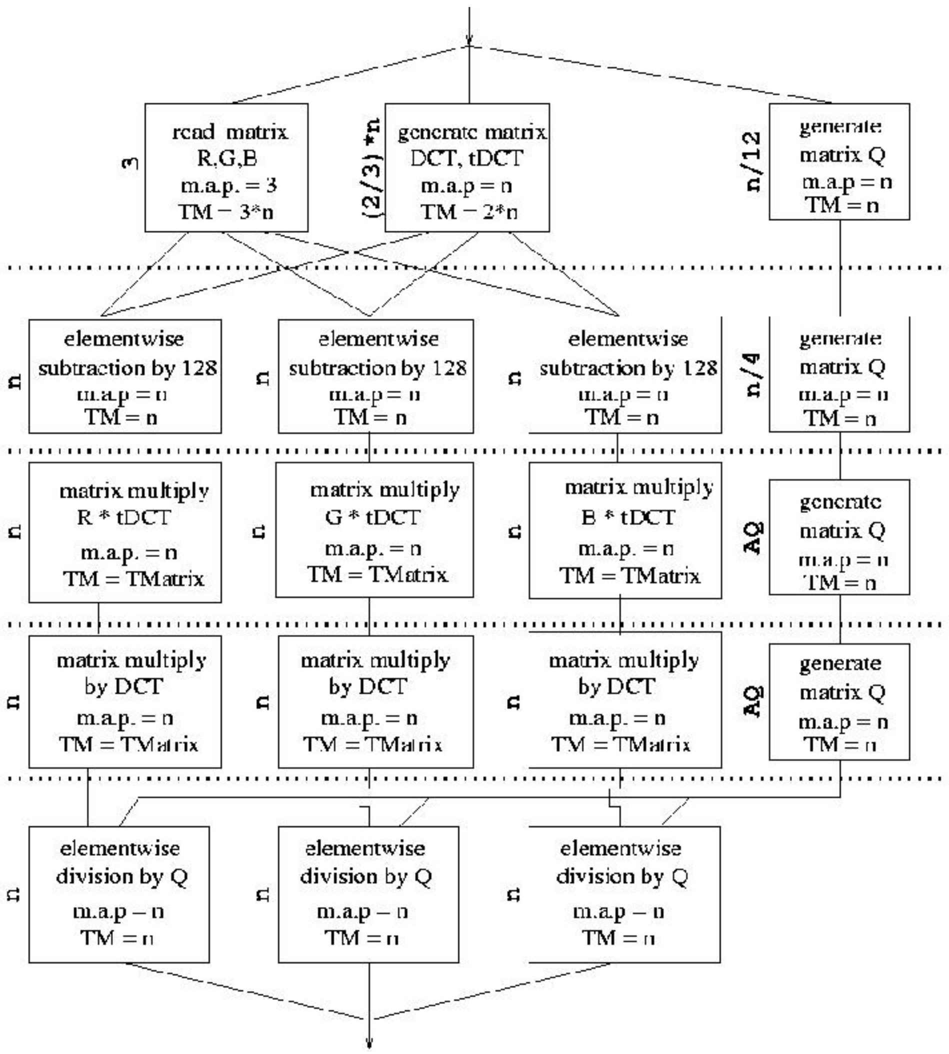
3
read matrix R,G,B m.a.p. = 3 TM = 3*n
(2/3)*n
generate matrix DCT, tDCT m.a.p = n TM = 2*n
n/12
generate matrix Q m.a.p = n TM = n
n
elementwise subtraction by 128 m.a.p = n TM = n
n
elementwise subtraction by 128 m.a.p = n TM = n
n
elementwise subtraction by 128 m.a.p = n TM = n
n/4
generate matrix Q m.a.p = n TM = n
n
matrix multiply R * tDCT m.a.p. = n TM = TMatrix
n
matrix multiply G * tDCT m.a.p. = n TM = TMatrix
n
matrix multiply B * tDCT m.a.p. = n TM = TMatrix
AQ
generate matrix Q m.a.p = n TM = n
n
matrix multiply by DCT m.a.p. = n TM = TMatrix
n
matrix multiply by DCT m.a.p. = n TM = TMatrix
n
matrix multiply by DCT m.a.p. = n TM = TMatrix
AQ
generate matrix Q m.a.p = n TM = n
n
elementwise division by Q m.a.p = n TM = n
n
elementwise division by Q m.a.p = n TM = n
n
elementwise division by Q m.a.p = n TM = n

Fig. 4: ***Optimized, layered graph of the Jpeg-algorithm, with nearly-optimal processor allocation (in bold font). All allocations are rounded to get an integer number of processors, which makes the numbers only nearly optimal.***

The resulting graph is presented in Fig.4, where

$$AQ = \frac{n^2}{4*\left(\frac{\left(\sqrt{n}\right)^3}{p} + 4*\sqrt{p}*t_s + 4*t_w*\frac{n}{\sqrt{p}}\right)} \qquad (2)$$

The optimization algorithm thus generates the allocation of processors, such that no task has to wait for other tasks of the same layer. Due to the rounding in (1), the allocation is only nearly optimal.

IMPLEMENTATION STATUS AND FUTURE WORK

The implementation of *VisPar* is being accomplished in *Java 1.3*. Java was our language of choice because of its portability and the extensive capabilities of the graphical components of its *Swing* class library. The design of the internal graph representation is based on the Model-Controller-View concept of Java that comprises two kinds of objects: the data model object which contains the user data and the visual object that provides the access to the information.

The class hierarchy of the current implementation is shown in Fig.5. The central class, *VisPanel*, is derived from the standard *JPanel* Java class. The editor functionality to manipulate the editor objects is realized as member methods of this class. The editor objects themselves are stored in the *VisPanelModel* class which provides methods to manipulate them, too. It holds the collection of *VisTask* and *VisEdge* objects that form the NoT-graph on the display and store parts of the graph data in the *VisTaskModel* and *VisEdgeModel* objects.

The innermost component of the editor is a special Java user class *EvalFunction*, responsible for parsing and evaluating cost functions; these are provided by the user as closed mathematical expressions. The tool detects input errors and produces a representation of a function as encapsulated postscript. Currently, all variables used in a cost function must be given a concrete value or value range before the function can be evaluated. The *EvalFunction* class contains *LRParser* and *LRGrammar*, which specify the parser and the grammar of mathematical expressions. These classes can be reused if a different grammar is desirable. The *LRParser* class contains an LR-Parser and is able to evaluate the values of the input string. If the cost function is not changed, it is possible to speed up the evaluation process by memorizing some values of the function. For the most important variables like problem size *n* and number of processors *p*, some often occuring values can be used to evaluate the cost function apriori.

The editor objects of *VisPar* export their data models as structured XML files for reusability purposes. Ultimately, the whole graph can be translated and exported in the XML format and then used by other software components of *VisPar*. In particular, the XML representation of the graph can be translated into parallel source code and compiled for a particular machine. Also a more detailed structure of the application can be inserted into an existing XML representation.

We try to keep our implementation as flexible as possible. All components are built as fully implemented Java objects, so that every further extension can be added by creating a derived class and adding further functionality. Especially graph layout functions can easily be added by subclassing the existing abstract objects of the editor.

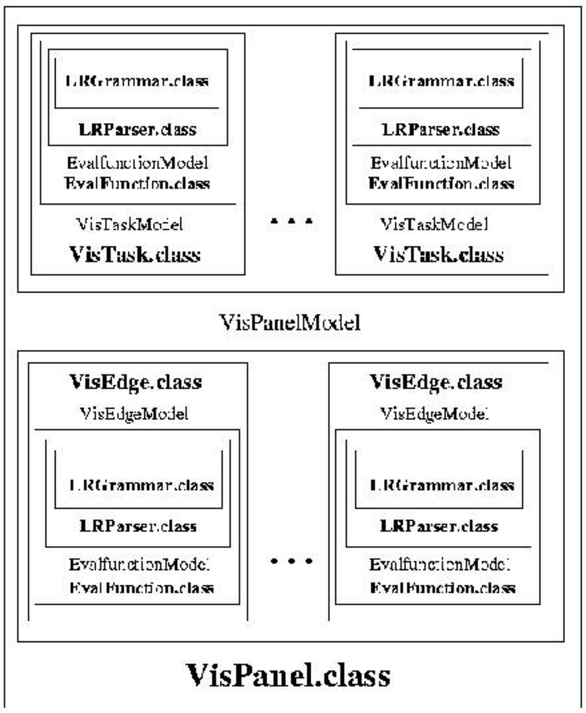

Fig. 5: ***Internal software structure of the VisPanel class that is the canvas for drawing NoT-graphs in the NoT-graph windows of the VisPar application.***

The main aspects that remain to be implemented are the optimization and selection procedures. For the latter, a simple method based on Dijkstra's shortest-path algorithm has been suggested elsewhere [1]. The next step will be the optimization on the data-parallel level. This will add a second layer of representation to the visual programming concept of *VisPar*. It will enable the user to construct a parallel program using exclusively visual components and obtain optimized intermediate parallel code that can be translated into different target parallel languages. Furthermore, this relieves the user of estimating the costs of tasks: these are supplied automatically by the data-parallel optimization methods.

We envisage the *VisPar* toolbar as having buttons to start the optimization and selection procedures. Every procedure will open a additional dialog that displays information referring to the optimization process. The results of optimization are displayed in a new graph-window showing the optimized NoT-graph. The optimization procedures can be performed on several NoT-graphs simultaniously. We also intend to add buttons for particular combinations of optimization algorithms.

Several usefull GUI enhancements are planned for the future work, like a batch processing mode to accommodate lengthy optimization processes like the algorithm from [3]. The next phase of extension could include adding pipelining and iteration into application graphs. Finally, additional Java modules should translate the DAG of a parallel program into a target language, e.g. MPI.

CONCLUSION

In our view, both the novelty and advantage of the *VisPar* tool is that it combines the sound foundation of parallelism modelling and optimization with the easy-to-use visual support for performance prediction and optimization in the design process.

The user starts with an intuitive description of potential parallelism and data dependencies in the application. Using the *VisPar* tool, optimization algorithms for data and task parallelism are applied, leading to an efficient implementation with a predictable performance on various configurations of an available parallel machine.

[1] D. Skillicorn. The Network Of Tasks Model. In Proceedings Int.Conf. on Parallel and Distributed Computing and Systems, Boston 1999.

[2] S.Gorlatch. "Towards formally-based design of message passing programs", IEEE Transactions on Software Engineering, 26(3): 276—288, 2000}.

[3] T. Rauber and G.Rьnger. Scheduling of dataparallel modules for scientific computing. In Proceedings Int. Conf. on Compilers for Parallel Computers. Aussois, 2000.

[4] V. Bhaskaran and K. Konstantinides. Image and video compression standards, algorithms and architectures, Kluwer, Boston 1995.

Sergei Gorlatch received his Master's degree in Computer Science from Kiev State University in 1979, and his PhD degree from Glushkov Institute of Cybernetics, Kiev, Ukraine in 1984.

From 1991 to 1992, he was a Humboldt Research Fellow at the Technical University of Munich.

From 1992 to 1999, Dr. Gorlatch worked as Assistant Professor at the University of Passau, Germany, where

he obtained his "Habilitation" (post-doctoral degree) in 1998. Since 2000 he has been Associate Professor at the Technical University of Berlin.

Sergei Gorlatch has actively participated in several projects on building novel parallel architectures and their software. His current research area includes parallel algorithms, programming methodology and formal methods for parallel and distributed systems, and performance evaluation.