Automatic Task Based Analysis and Parallelization in the Context of Equation Based Languages

[Work in Progress]

Mahder Gebremedhin Peter Fritzson

PELAB - Programming Environments Laboratory

Dept. of Computer and Information Science

Linköping University

SE-581 83 Linköping, Sweden

{mahder.gebremedhin, peter.fritzson}@liu.se

ABSTRACT

This paper presents an automatic parallelization approach for handling complex task systems with heavy dependencies, including methods of analyzing dependencies, representing them in a convenient way, and processing the resulting task graph representation. We present a library-based task system representation, clustering, profiling, and scheduling approach to simplify the otherwise tedious process of parallelizing complex task systems. We have implemented a flexible and robust task system handling library to manipulate and parallelize these complex task systems on shared memory multi-core and multi-processor systems. The implementation has been developed as part of the OpenModelica simulation environment. We demonstrate methods of extracting and utilizing parallelism in the context of mathematical modeling languages.

Keywords

Task Parallel, Multi-core, Modeling, Simulation, Parallel Simulation, Modelica, OpenModelica

1. INTRODUCTION

In this paper we present an approach of extracting parallelism from systems of equations, representation as task graphs, and a generic, flexible, and portable task system management library in C++ for representing, clustering, profiling, scheduling and executing complex task systems efficiently. The implementation is an extension to the OpenModelica [1] runtime system. The dependency analysis and extraction of parallelism is done automatically based on the Modelica model description with the help of the OpenModelica compiler.

2. PREVIOUS WORK

A lot of research work has been done on parallelization based on the OpenModelica compiler and the Modelica language, including the early modpar parallelization effort [2]. Later this work was extended to support pipelining at the solver level [3]. A recent effort at TU Dresden [4] implements an approach similar to our work.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).
EOOLT 2014, Oct 10-10 2014, Berlin, Germany
ACM 978-1-4503-2953-8/14/10.
<http://dx.doi.org/10.1145/2666202.2666210>

[5] has also presented a variation of a level scheduler implementation similar to one of the schedulers presented in this paper.

All four implementations are concerned with task parallel executions. The most important distinction between our work and those mentioned above is that the focus in this work is more on providing a standalone flexible task system handling implementation that is not tied to a specific environment. To this end we have opted for separate handling of the normal simulation process and task parallel related operations. This decision is made for improved flexibility, performance, and maintainability.

3. EQUATION SYSTEMS AND DEPENDENCY ANALYSIS

The sorting and casualization of equations by a Modelica compiler results in a system which we can be represented in an incidence matrix of equations and variables in block lower triangular (BLT) form. For example consider this acausal ODE equation system:

$$\begin{aligned} f_1(x_1, x_2, t) &= 0 \\ f_2(x_3, t) &= 0 \\ f_3(x_1, x_3, x_4, t) &= 0 \\ f_4(x_3, x_5, t) &= 0 \\ f_5(x_1, x_4, x_5, t) &= 0 \\ f_6(x_6, t) &= 0 \\ f_7(x_6, x_7, t) &= 0 \end{aligned}$$

One sorting and casualization of these equations can result in a system of equations shown below (different sorting output is possible, see [6])

$$\begin{aligned} x_6 &= g_6(t) \\ x_7 &= g_7(x_6, t) \\ x_3 &= g_2(t) \\ x_5 &= g_4(x_3, t) \\ g_3(x_1, x_3, x_4, t) &= 0 \\ g_5(x_1, x_4, x_5, t) &= 0 \\ x_2 &= g_1(x_1, t) \end{aligned}$$

The incidence matrix in BLT form is shown in Figure 1.

	x_6	x_7	x_3	x_5	x_1	x_4	x_2
g_6	1						
g_7	1						
g_2			1				
g_4			1				
g_3				1			
g_5				1			
g_1					1		

Figure 1 Original Incidence Matrix in BLT form

Five of the equations in the system are causalized to assignment statements of the form:

$$x = f(v, t)$$

where t represents time and v is the vector of variables involved in the computation of x .

Equations g_3 and g_5 , however, are strongly connected and form an algebraic loop that needs to be solved together for the variables x_1 and x_4 . These blocks of equations, forming linear or nonlinear systems, can be composed of tens or hundreds of simple equations. These blocks are usually the most expensive parts of a simulation in terms of computation time since they often involve multiple assignments, complex linear algebra operations as well as possible iterations within the block. This gives rise to potential data parallelism within the blocks since most linear algebra operations can be parallelized. However, here we ignore this potential and treat these complex blocks as atomic units of computation that need to be executed non-preemptively as a whole by a single thread.

	x_6	x_7	x_3	x_5	x_1, x_4	x_2
g_6						
g_7						
g_2						
g_4						
g_{35}						
g_1						

Figure 2 Incidence Matrix in proper BLT form

We can represent the system of equations formed by these multiple equations as

$$x = f(v, t)$$

where x is now the vector of variables being updated by the system of equations. In the above example we can treat g_3 and g_5 as one single equation of the form:

$$\{x_1, x_4\} = g_{35}(x_3, x_5, t)$$

resulting in the incidence matrix of proper BLT form shown in Figure 2, where we observe that the system contains two sets of connected components. The sets $\{g_6, g_7\}$ and $\{g_2, g_4, g_{35}, g_1\}$ form two completely independent subsystems. Having these kinds of equation systems with multiple decoupled system gives some potential for parallelism. The decoupling of systems can be further improved by a modeling technique called Transmission Line Modeling (TLM). In previous work we have investigated this approach and implemented a parallelization approach that is based on balancing these completely independent systems and executing them in parallel [7]. Here we have decided to improve the previous implementation to analyze not just connected components but the whole system and extract more parallelism.

To this end we have implemented a task graph based approach to better represent the system and enable a more convenient analysis. We start by converting the incidence matrix into an adjacency list representing a directed acyclic graph (DAG). A directed acyclic graph DAG, where each node represents an equation block and each directed edge from block i to block j represents a dependency of block j on the variable defined (assigned) in block i , can be built using the information provided in the incidence matrix.

An alternative approach of extracting more parallelism by further analyzing the BLT incidence matrix is proposed by Casella [8]. From now on we will refer to equation blocks (both simple and complex) as tasks and treat both block types as atomic tasks.

4. THE TASK SYSTEMS LIBRARY

The task systems library implemented in this work is a generic task representation, clustering, profiling and execution library written in C++. It uses C++ templates heavily and is built on top of the Boost Graph and Intel Thread Building Blocks Libraries. The Boost Graph is used to represent the underlying graph primitives for the task systems and TBB for the parallelization primitives.

The library can be used for the whole task representation and parallelization process including clustering, profiling and execution. However it is also possible to only use it for a few of these purposes. For example it can be used to just perform a specific set of clustering algorithms on the task system and use the resulting system for other purposes. The idea is to provide a framework that is easy to extend, for example by adding a new custom clustering algorithm or a new scheduling algorithm, while the rest of the system representation stays intact.

The library uses an adjacency list to represent a directed acyclic graph (DAG) that models a given task system. The task system can be represented by the tuple

$$G = (V, E, c)$$

V is the set of vertices, E is the set of directed edges and c is the execution cost of each vertex. Each vertex in the graph corresponds to one Task and each directed edge represents a data dependency between the source vertex/Task and the destination vertex/Task. Currently the library only supports shared memory systems.

4.1 Tasks

A user defined task is represented as a C++ class which must inherit from the abstract class **Task** provided by the library. The abstract base class provides the necessary information for the clustering and scheduling algorithms. For example a modified version of an equation task for the OpenModelica compiler is shown in Listing 1.

```

struct Equation
    : public Task
{
    Equation();
    long index;
    std::set<long> depends;
    std::set<long> updates;
    virtual bool depends_on(const Task&) const;
    virtual void execute();
};

```

Listing 1 A custom class represents an equation task

Construction of a task system starts with creation and addition of Tasks. The library can read the necessary information from an XML specification and create the system automatically. Tasks can also be added manually directly one by one into the task system.

Creation of edges, i.e., dependency representation, can be handled in two ways. The first method involves the user directly creating edges between vertexes. Another option is to let the library handle the analysis to some extent. When a task is added to the system the library traverses each existing task and determines if there is a dependency between the existing task and the new task.

A node in the internal task graph representation is not a task but rather a **cluster of tasks**. Each cluster initially consists of a single task, i.e., when the library is asked to create a new task it creates a new cluster with a single task. Later the different clustering algorithms rewrite the system graph by moving tasks from one cluster to another. If a cluster becomes empty it is deleted.

Clusters are responsible for the profiling and execution of their tasks. Tasks in a single cluster are always **executed sequentially in the order they are added** to the cluster.

4.2 Clustering Algorithms

The library also provides some algorithms for clustering the system graph as well as mechanisms for writing customized clustering algorithms. Once the task system is created an optional clustering phase can be applied. The available clustering algorithms can be applied in any combination and order.

Currently there are four available clustering algorithms in the library: Merge Single Parent (MSP), Merge Level Parents (MLP), Merge Children Recursive (MCR) and Merge Level for Cost (MLC). These algorithms traverse the task system and move tasks from one cluster to another (merge tasks) depending on specific criteria.

The first two algorithms are cost-oblivious, ignoring the cost of the child task, the parent task, or the merged task while applying the merging rules. They are useful for improving temporal locality so that tasks that operate on the same portion of data are executed as closely as possible. The last two algorithms (MLP and MCR) are cost based algorithms traversing the task system and cluster tasks until a user-specified algorithm specific target cost cutoff is achieved. Cutoff cost should be selected to avoid unnecessary clustering since this will limit the parallelization available for the schedulers.

Custom clustering algorithms can be easily added to the library. The library provides methods for traversing, marking, collecting, sorting, merging tasks as well as dumping graphs of systems for debugging purposes so that developers can focus solely on their algorithms.

4.3 Profiling and Cost Estimation

There are two ways of handling cost estimation for tasks in the library. The first method is static cost estimation, which relies on user provided cost values for the whole operation of clustering and scheduling. Users have to set costs of tasks at task creation time or later but before applying any clustering rules. Static cost estimation is mandatory when the library is used just for offline scheduling. Since there is no execution of tasks in this case, all cost information should be provided by the user.

Static cost estimation is suitable for handling tasks that are executed only once or very few times per program or whole system execution. For such systems tasks costs have to be estimated manually at compile time since there will be no opportunity to actually measure and store execution times of the tasks at run time. This can be done by analyzing the internal representation of tasks when possible, e.g. by traversing abstract syntax trees.

Modeling and simulation environments on the other hand involve variations of iterations over a certain set of tasks to solve the given problem. This provides the opportunity to measure and store execution costs of tasks dynamically at runtime and use this information to perform more effective clustering and scheduling.

In dynamic cost estimation mode the library always assumes the first request to evaluate the whole task system as a profiling stage where all tasks are executed sequentially and in profiling mode. Execution times are recorded and stored in each task. Once the whole system has been executed the library will schedule the system. In all subsequent steps or calls for evaluation of the system, parallel execution is performed using the existing schedule unless rescheduling is requested explicitly. Periodic rescheduling might be useful since task costs might change significantly between evaluations of the task system due to state changes in the system.

4.4 Schedulers

Schedulers in the library are implemented as standalone C++ template classes responsible for executing clusters in parallel and synchronizations in their own way. Actual clustering (if needed) is provided by the clustering classes explained in section 4.2. These clustering algorithms are passed as template parameters for the schedulers and applied in the order they are passed.

Currently two simple scheduler implementations are available in the library. The first is a level based scheduler being lock step or wave front based. Level schedulers incur very low overhead in managing tasks and synchronization since the only requirement is that tasks at one level are all finished before starting any task at the next level. This synchronization and thread management facility is provided by a core scheduler class *StepSync*. The actual level scheduler implementation is a specialization of this core algorithm with specific clustering algorithms. Variations of this level scheduling approach can be created simply by specializing the core *StepSync* class with a different set of clustering algorithms in different orders. For example the Level Scheduler class used mainly in the current OpenModelica parallelization implementation is shown in Listing 2.

```
template<typename TaskType>
struct LevelScheduler :
    StepSync < TaskType
        ,MCR
        ,MLC
    > {};
```

Listing 2 Level Scheduler Class for OpenModelica

The second scheduler is built on top of the Intel Thread Building Blocks' flow graph implementation. This scheduler uses the Intel TBB's Flow Graph (TBBFL) implementation as core scheduler [9], which implements its own a message based work stealing algorithm to dynamically execute tasks in the graph. The scheduler is implemented as a wrapper for the TBB Flow Graph. It incorporates clustering and hides the details of TBB related primitives from the user. Similar to the Level Scheduler explained above this class is responsible for the profiling, cost estimation, and execution of tasks. It automatically creates the additional flow graph once profiled cost values are obtained and specified clustering rules are applied to the task system.

5. PERFORMANCE

To evaluate the performance of the task scheduling implementation and schedulers we have performed tests using models from the Modelica Standard Library. We have selected and presented two selected models with satisfactory performance improvements here.

All measurements have been done on a 64-bit Intel(R) Xeon(R) W3565 CPU with 4 cores at 3.2 GHz (3.46

GHz turbo) frequency. The machine is running Windows 7 professional Edition. Simulations have been performed from time 0 to 1 second with step of 0.002 seconds, using the default OMC solver, currently DASSL. Only ODE systems of models are currently parallelized.

The time results presented do not include model compilation time, only simulation executable run times. However, all parallelization related execution times are included. This includes all the extra overhead from task system creation, clustering and scheduling plus a sequential first step computation performed to collect cost information. Speedups presented are ratios of sequential to parallel execution times.

For each model we have presented the estimated speedup with level scheduling and actual achieved speedup with level scheduling as well as the Intel flow graph based scheduler. The estimated speedup for the level scheduler is the ratio of the sequential cost to parallel cost. Sequential cost is obtained by adding the cost of each individual task in the system. The parallel cost is obtained by summing the costs of the largest task at each level in the clustered task graph.

Figure 3 Speedup for CauerLowPassSC model

Figure 4 Speedup for BranchingDynamicPipes model

The first test model presented is the fifth order low-pass-filter model (CauerLowPassSC) from the Electrical Analog domain, with speedup curves in Figure 3. The second model is the Branching Dynamic Pipes model from Fluid domain demonstrating the use of a distributed pipe models with dynamic energy, mass, and momentum balances, with speedup curves in Figure 4.

For the CauerLowPassSC model the level scheduler implementation outperforms the dynamic flow graph scheduler for both 2-threaded and 4-threaded executions. On the other hand for the BranchingDynamicPipes model the flow graph based scheduler outperforms the level scheduler on both runs. One reason for this behavior can be the different nature of equation system composition in the two models. BranchingDynamicPipes model results in an ODE task system with 48 nonlinear systems while the CauerLowPassSC has no nonlinear systems. parts of the simulation executable (i.e., large tasks). Lots of large

tasks gives the dynamic flow graph scheduler a higher parallelization to overhead ratio since threads spend most of their time working on these large tasks.

On both test cases the level scheduler shows very promising estimated speedups. Although it is not practically possible to achieve this ideal speedup the implementation can be improved to achieve close to estimated speedups.

6. CONCLUSION

We have shown that the task graph based parallelization implementation already provides significant performance improvements over sequential execution. We have provided a flexible and efficient task system management library. The library provides multiple task system handling functionalities which simplifies usage and extensions.

There is a lot of room for further improvement. More clustering and scheduling algorithms should be implemented to give users a wider range of options. Better clustering algorithms lead to even better data temporal locality which should improve performance by increasing cache usage efficiency in CPU systems. Moreover, enhancements are possible of the user friendliness and flexibility of the library as well as the performance of the current scheduling algorithms.

7. REFERENCES

- [1] "OpenModelica": <https://www.openmodelica.org/>.
- [2] P. Aronsson, "Automatic Parallelization of Equation-Based Simulation Programs," 2006 www.diva-portal.org/smash/get/diva2:22444/FULLTEXT01.pdf.
- [3] H. Lundvall, K. Stavåker, P. Fritzson and C. Kessler, "Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms," in *ACM SIGARCH Computer Architecture News*, 2008.
- [4] M. Walther , V. Waurich , C. Schubert and I. Gubsch, "Equation based parallelization of Modelica models," in *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, 2014.
- [5] H. Elmqvist, S. E. Mattsson and H. Olsson, "Parallel Model Execution on Many Cores," in *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, 2014.
- [6] F. E. Cellier and E. Kofman, *Continuous System Simulation*, Springer, 2006.
- [7] M. Sjölund, M. Gebremedhin and P. Fritzson, "Parallelizing Equation-Based Models for Simulation on Multi-Core Platforms by Utilizing Model Structure," in *17th Workshop on Compilers for Parallel Computing*, Lyon, France.
- [8] F. Casella, "A Strategy for Parallel Simulation of Declarative Object-Oriented Models of Generalized Physical Networks," in *5th EOOLT workshop* Nottingham, UK, 2013.
- [9] Intel Corporation , "TBB Flow Graph," [Online]. http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm. [Accessed 4 7 2014].